Heterogeneous Cloud Systems Based on Broadband Embedded Computing

Richard Neill

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2013

©2013
Richard Neill
All Rights Reserved

ABSTRACT

Heterogeneous Cloud Systems Based on Broadband Embedded Computing

Richard Neill

Computing systems continue to evolve from homogeneous systems of commodity-based servers within a single data-center towards modern Cloud systems that consist of numerous data-center clusters virtualized at the infrastructure and application layers to provide scalable, cost-effective and elastic services to devices connected over the Internet. There is an emerging trend towards heterogeneous Cloud systems driven from growth in wired as well as wireless devices that incorporate the potential of millions, and soon billions, of embedded devices enabling new forms of computation and service delivery. Service providers such as broadband cable operators continue to contribute towards this expansion with growing Cloud system infrastructures combined with deployments of increasingly powerful embedded devices across broadband networks. Broadband networks enable access to service provider Cloud data-centers and the Internet from numerous devices. These include home computers, smart-phones, tablets, game-consoles, sensor-networks, and set-top box devices.

With these trends in mind, I propose the concept of broadband embedded computing as the utilization of a broadband network of embedded devices for collective computation in conjunction with centralized Cloud infrastructures. I claim that this form of distributed computing results in a new class of heterogeneous Cloud systems, service delivery and application enablement. To support these claims, I present a collection of research contributions in adapting distributed software platforms that include MPI and MapReduce to support simultaneous application execution across centralized data-center blade servers and resource-constrained embedded devices. Leveraging these contributions, I develop two complete prototype system implementations to demonstrate an architecture for heterogeneous Cloud systems based on broadband embedded computing. Each system is validated by executing experiments with applications taken from bioinformatics and image

processing as well as communication and computational benchmarks.

This vision, however, is not without challenges. The questions on how to adapt standard distributed computing paradigms such as MPI and MapReduce for implementation on potentially resource-constrained embedded devices, and how to adapt cluster computing runtime environments to enable heterogeneous process execution across millions of devices remain open-ended.

This dissertation presents methods to begin addressing these open-ended questions through the development and testing of both experimental broadband embedded computing systems and in-depth characterization of broadband network behavior. I present experimental results and comparative analysis that offer potential solutions for optimal scalability and performance for constructing broadband embedded computing systems. I also present a number of contributions enabling practical implementation of both heterogeneous Cloud systems and novel application services based on broadband embedded computing.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many individuals have helped me through my long journey to arrive at this point. First, I would like to thank my advisor, Luca Carloni, for his inspiration, guidance, and support, especially early on when I first started at Columbia. With his seemingly unlimited energy, confidence, ideas and skillful direction, I have learned how to conduct research for which I will be forever grateful. I would also like to thank my committee members, Stephen Edwards, Martha Kim, and Yemini Yechiam for their valuable suggestions, and perspective, challenging me to recognize important open problems, so that I may focus my research goals.

I would like to thank Cheng-Hong Li for his patience during my early years and Rebecca Collins for her help and assistance throughout. Many thanks to the staff at CSL, especially Daisy Nguyen for her assistance in facilitating the construction of my research infrastructure.

A number of individuals have been instrumental in helping me during my research. In particular, I would like to thank and acknowledge YoungHoon Jung who I have worked closely with as a co-developer of a heterogeneous MapReduce computing system presented in my dissertation. YoungHoon has been an exceptional research partner and I thank him for his ideas and contributions to my research. I would like to also thank Marcin Szczodrak for helping me with his systems expertise and more recently with the introduction of new ideas for future work. I look forward to continued collaborations with both YoungHoon and Marcin. I would like to thank Young Jin Yoon for his support and never-ending reassurance during my journey.

This research effort would not be possible without the support of Cablevision systems and my industry colleagues. I would like to thank Satish Marada, Dwarak Gummadikayala, Alex Shabarshin, Serguei Tcherepanov and Valeriy Sigaev for all their support and effort in the implementation of the prototype lab systems.

Finally, and most of all, I am grateful to God and for my family. My wife, Peggy for patiently spending days and nights with me, always by my side, providing me with never-ending encourage-

ment and support of all my endeavors. To my father who would have been very happy for this moment, and my mother who is perhaps my biggest fan and supporter, even though she is not sure what I have been working on all this time. Lastly, my brothers Bob and Tom, and sister Nancy, for their lifelong encouragement.

Dedicated to Peggy and my parents

Chapter 1

Introduction

A number of evolutionary trends in Information Technology and the emergence of pervasive embedded systems across both wired and wireless networks are impacting the computing landscape in two ways. On one hand, computation is moving away from traditional desktop and centralized computing centers towards an infrastructural core that consists of many large and distributed data-centers with high-performance computer servers and data storage devices. Following the *Cloud computing* paradigm, these large-scale data-centers provide the delivery of numerous computational, storage and application services to a multiplicity of peripheral clients, through various interconnection networks. On the other hand, the increasing majority of these clients consist of a growing variety of embedded and consumer electronic devices. Such devices are mobile smart phones, tablets, video-game consoles, television set-top boxes (STB), and a multiplicity of intelligent sensors whose capabilities continue to improve. For example, both modern data-center class blade servers and embedded devices share increasingly powerful multi-core systems-on-chip (SoC) and graphics processor unit (GPU) implementations. This trend is confirmed by the recent announcement that STMicroelectronics and ARM have teamed up to deliver high-performance ARM Cortex-A9 MPCore processors to the STB industry [34]. The ARM Cortex-A9 features from 1 to 4 cores, L1 and L2 caches, with each core also containing either a media processing unit, or FPU with support for both single and double precision floating-point operations [54].

The performance gap between data-center blade servers and embedded device processors also continues to decrease as processor power dissipation and clock frequencies approach their upper limit. As a consequence, Cloud-system platforms are becoming increasingly heterogeneous, with

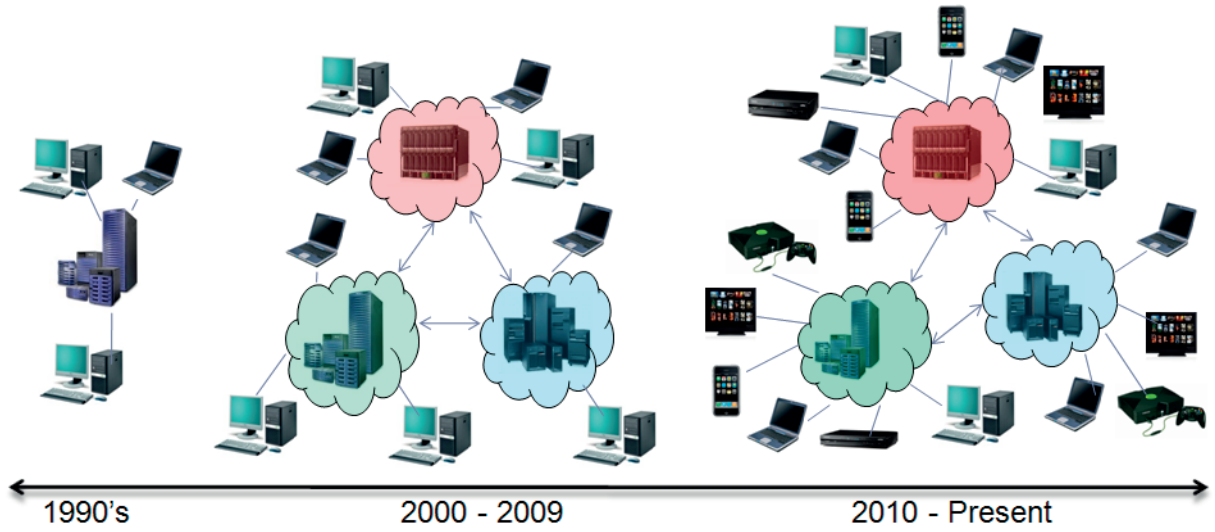


Figure 1.1: Evolution towards heterogeneous Cloud computing.

continued performance improvements achieved through growth in the number of Cloud-system computational nodes, corresponding server processor cores, and software paradigms that exploit patterns of distributed computation wherever possible.

Evolution towards heterogeneous Cloud systems. Service providers such as Google, Amazon, Yahoo, and cable system operators (also referred to as Multiple Service Operators or MSOs) are driving the trend in heterogeneous computational systems. Fig. 1.1 illustrates this trend over roughly the past 15 years; where highly centralized data-center computing systems have evolved towards a *heterogeneous Cloud system* model that is both highly distributed and heterogeneous in device composition.

An early definition for heterogeneous Cloud systems is given by Crago, *et al.* [24] who extends traditional Cloud computing infrastructures of homogeneous processors to a heterogeneous processor architecture consisting of GPUs and data-center blade servers utilizing general purpose CPUs. The introduction of heterogeneity allows Clouds to be competitive with traditional distributed computing systems at a larger scale and at comparatively lower capital expenditures [24]. In the heterogeneous Cloud systems model, distributed Cloud data-centers have expanded to include the integration of networks of distributed embedded systems as shown in Fig. 1.1. Emerging systems for *Embedded and Mobile Cloud Computing* have extended the interpretation of heterogeneous Cloud systems to include support for dynamic (on-demand without user interaction) device-capability aug-

mentation from data-center Cloud computing facilities to billions of wired and wireless embedded devices across diverse application domains [29; 64].

Similar to the previous definition of heterogeneous Cloud systems, a recent paper by Bonomi, *et al.* describes the notion of *Fog Computing* as a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud computing data-centers located at the edge of the network [16]. The definition of Fog Computing includes data-center class systems and networked embedded devices that are highly location independent, such as connected vehicles, and wireless sensors/actuator networks enabling the implementation of Smart Grid applications [16].

MSOs are examples of service providers that are driving the evolution of large-scale Cloud systems and growth in deployment of large numbers of increasingly-powerful embedded processors across broadband networks. As a result, there has been an explosion of both mobile and stationary devices that access the Internet through broadband connectivity. In turn, this has led to rapid increases in the number of consumer applications accessible through broadband networks. Since 2006, MSOs have accelerated the purchase and deployment of next-generation embedded *set-top box (STB)* devices to support the deployment of digital interactive services and content. According to a recent market research [65], worldwide STB shipments has a projected growth of over 150 million units in 2010, rising to nearly 201 million units by 2013. Growth of mobile Internet computing has outpaced similar desktop Internet adoption. For instance, during the first two years since launch, Apple had acquired over 57 million iPhone and iTouch subscribers. This is more than eight times the number of AOL users for a similar period during the emergence of desktop Internet [68].

1.1 Broadband Embedded Computing and Heterogeneous Cloud Systems Taxonomy

In order to meet the growing computation and communication demands, MSOs are rapidly assembling complex Cloud system infrastructures which are highly heterogeneous and distributed over massive broadband networks. The MSO infrastructure primarily consists of geo-diverse Cloud data-centers that efficiently deliver video and data over broadband networks to large populations of embedded devices. These systems are unique in that they are centrally managed by the MSO

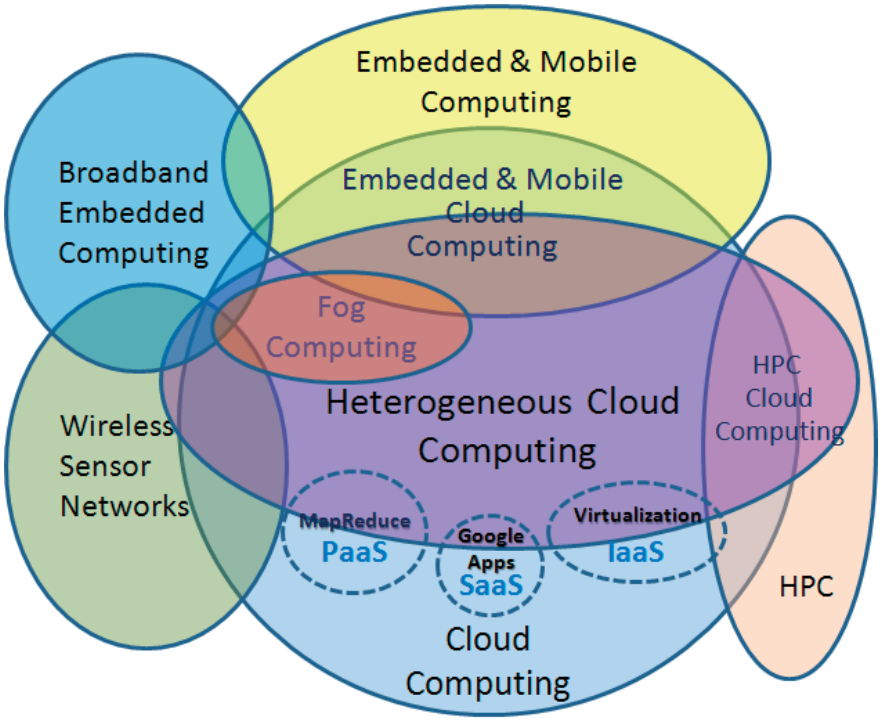


Figure 1.2: Heterogeneous Cloud taxonomy.

and must scale to reliably execute concurrent application processes across millions of devices. I claim that *the ability to centrally manage the execution of application processes across millions of broadband-networked embedded devices enables the utilization of service provider embedded systems for a new class of distributed computation that I refer to as broadband embedded computing [79; 98; 99].* Broadband embedded computing is the utilization of a broadband network of embedded devices for collective computation in conjunction with centralized Cloud data-center infrastructures.

Heterogeneous Cloud systems taxonomy. The evolution towards heterogeneous Cloud systems has led to numerous Cloud computing system variants depending on the composition of devices and platform services required. Fig. 1.2 illustrates a taxonomy for a number of heterogeneous Cloud computing system types based on the integration of various distributed embedded systems and the traditional data-center centric Cloud computing model. The addition of broadband embedded computing is shown on the left-hand side of the figure as an intersection among broadband connected networks of wireless sensor, embedded and mobile computing devices. Referring to Fig. 1.2, heterogeneous Cloud computing extends the definition and service-delivery model

of traditional Cloud computing. For example, heterogeneous Cloud computing systems implement standard Cloud service delivery methods such as: PaaS (Platform as-a Service), SaaS (Software as-a Service), and IaaS (Infrastructure as-a Service). Therefore, the definition of Heterogeneous Cloud systems from a service delivery perspective is consistent and derives from existing formal definitions of Cloud computing [92]. In comparison to traditional Cloud systems, however, its taxonomy expands beyond an existing traditional data-center centric view to include emerging classes. Fig. 1.2 illustrates a number of emerging heterogeneous Cloud systems including: 1) mobile embedded, 2) broadband embedded, 3) wireless-sensor network based, and 4) high-performance Cloud computing systems. At the intersection of each technology, a different class of heterogeneous Cloud system is defined. For example, the intersection of data-center centric Cloud computing and broadband embedded computing results in heterogeneous Cloud service delivery platforms that utilize both data-center and broadband embedded computing system resources.

The integration of broadband embedded computation with traditional data-center cluster systems and Cloud infrastructure technologies such as processor virtualization, distributed programming models, and application services, results in a new class of heterogeneous Cloud systems based on broadband embedded computing.

The taxonomy illustrated in Fig. 1.2 is representative, but not exhaustive. New classes of heterogeneous Cloud systems will likely be developed in the future to further extend this taxonomy.

1.2 Dissertation Scope and Contributions

The scope of the dissertation is as follows. First, the dissertation develops and experimentally confirms heterogeneous system computation utilizing broadband embedded devices through the integration of traditional commodity blade server clusters with a real-world broadband embedded computing system implementation. The broadband embedded computing system is based on common set-top boxes that exist worldwide in large numbers across multiple service provider systems. A major research effort entails the investigation and development of new distributed software technologies based on two existing distributed computing paradigms, MPI and MapReduce, that are optimized to enable heterogeneous system implementation. Specifically, the dissertation presents the optimization of both MPI and MapReduce based software platforms for distributed embedded

systems and their integration into traditional data-center cluster computing environments.

Second, the dissertation includes an in-depth analysis of broadband communications cost and performance for the development of a scalable runtime environment system as well as validation in the form of simulated application model behavior. Experimental results are utilized to support the proposed design of a scalable communications model within broadband network environments.

Finally, an architecture for large-scale heterogeneous Cloud systems utilizing broadband embedded computing is proposed. The dissertation concludes with the presentation of four proposed application scenarios leveraging the systems developed throughout the dissertation. Open areas of research include optimization of distributed middleware to improve communication performance as well as addressing fault-tolerance in heterogeneous Cloud system environments.

The following outlines a detailed description of the dissertation contributions by chapter:

Dissertation contributions by chapter. The introduction in **Chapter 1** defines the concept of heterogeneous Cloud systems along with the fundamental thesis claim that the Cloud system model taxonomy can be extended by utilizing a new class of broadband embedded computation across networks managed by multiple service providers. Key to this claim is the concept of broadband embedded computing defined as the utilization of a broadband network of embedded devices for distributed computation. **Chapter 2** provides background for the remainder of the dissertation. The feasibility of broadband embedded computing is confirmed through experimental studies and analysis in **Chapter 3**. The study includes the implementation of a data-acquisition system that obtains measures of embedded set-top device CPU, memory, and uptime over an extended period of time, confirming available resources for broadband embedded computation.

Next, three different broadband embedded computing systems are implemented. The first-generation system for broadband embedded computing based on MPI is implemented and experimentally evaluated in **Chapter 4**. These research results were presented at the 2010 ACM International Conference on Computing Frontiers [99]. This is the first published work where an integration between an embedded set-top cluster and Linux cluster is built to gain insight into practical system requirements for heterogeneous system computing leveraging broadband embedded devices. Experimental results show that there are challenges regarding the scalability of traditional runtime environments for broadband embedded computing, suitability of standard MPI implementations on resource constrained embedded devices, and broadband communications performance.

Based on the lessons learnt in Chapter 4, a second-generation broadband embedded computing system is developed in **Chapter 5** addressing key open issues from the first-generation system, leading to several new contributions which were published in a paper presented at the 12th IEEE/ACM International Conference on Grid Computing [98]. First, a new virtualization model is described that maps embedded devices into the runtime space of traditional compute cluster runtime management systems. Second, this virtualization model is implemented as a new runtime environment called the *Open Embedded Runtime Environment or OERTE*. The OERTE system executes on Linux blade servers within a centralized cluster representing a typical Cloud data-center infrastructure. OERTE provides transparent launch and execution of distributed application processes across both traditional computing clusters and broadband embedded computing clusters based on the standard *Open Runtime Environment* process execution model. OERTE also provides services that offload operations from resource-constrained embedded devices to the virtual runtime environment. Third, to support complete interoperability across traditional MPI Linux clusters and broadband embedded computing clusters, an optimized version of the MPI software environment is developed for the embedded device environment. The new embedded MPI library implementation takes advantage of the OERTE virtual runtime system and is fully compatible with standard MPI system implementations. The second-generation system is subjected to a number of experiments including the execution of two applications (multiple sequence alignment, and image rendering) and the IMB benchmark suite [19] to validate consistency across both cluster environments, compare embedded versus blade processor performance, and evaluate broadband communication performance of collective operations.

Cloud systems are increasingly being utilized to process large data-sets and solve so called *Big Data* problems via MapReduce computation. **Chapter 6** describes a new heterogeneous Cloud system for MapReduce processing using broadband embedded computing. This work was published at the 4th IEEE International Conference on Cloud Computing Technology [79]. Contributions described in Chapter 6 include methods for porting the popular Hadoop system to the embedded environment using several back-porting software techniques. The system is evaluated using common MapReduce benchmarks to compare the performance of the traditional blade cluster and embedded cluster environments.

Chapter 7 provides the first in-depth simulation study on broadband network performance

characteristics when scaling the number of nodes to 8000 devices. A key contribution is formally addressing the communication challenges associated with managing the communication costs associated to launching and executing an application process across millions of broadband devices. Extensive experiments are described and carried out to evaluate broadband communications for typical client-server interactions. Two communication methods, multicast and unicast are evaluated under various lab and simulation experimental scenarios to compare broadband communication performance in both cases and perform a scalability analysis. Experimental results confirm the validity of the proposed network design for a highly-scalable runtime system for managing execution across a heterogeneous Cloud system that includes broadband embedded devices.

Based on lessons learnt with the three broadband embedded computing system implementations and the broadband network experimental results, **Chapter 8** describes a complete heterogeneous Cloud system architecture that utilizes both data-center server clusters and broadband embedded devices. The final section of Chapter 8 discusses four proposed heterogeneous Cloud system application scenarios based around broadband embedded computation. Some of the contributions of this chapter were published at the 54th IEEE International Midwest Symposium on Circuits and Systems [97]. I conclude the dissertation by presenting ideas for future work in **Chapter 9**.

Chapter 2

Background

2.1 Overview

In this chapter background information covering a diverse range of topics is presented that is used throughout the dissertation. Specific topics include: distributed computing based on MPI, Map-Reduce, Volunteer, and Cloud computing service-infrastructure models, coverage of broadband cable networking standards and technologies; cable service provider system architecture; and finally a survey of embedded device technologies commonly deployed over broadband systems. While any one of these topics entails volumes of books and deserves a detailed study in itself, the primary purpose of the following sections is to set the context for the rest of the dissertation.

2.2 Distributed Computing

Distributed computing systems consist of autonomous computers that communicate through a computer network while executing a distributed program to solve a computational problem [10]. In distributed computing, a problem is divided into many tasks each of which is solved by one or more computing nodes. Distributed computing systems consist of computational nodes that each contain their own local memory and processor, and communicate with one another using message-passing primitives [10]. An example of a distributed computing network model is shown in Fig. 2.1. There are virtually an unlimited number of possible distributed computing network topologies. However, in practice distributed computing networks typically follow a finite set of topologies essentially

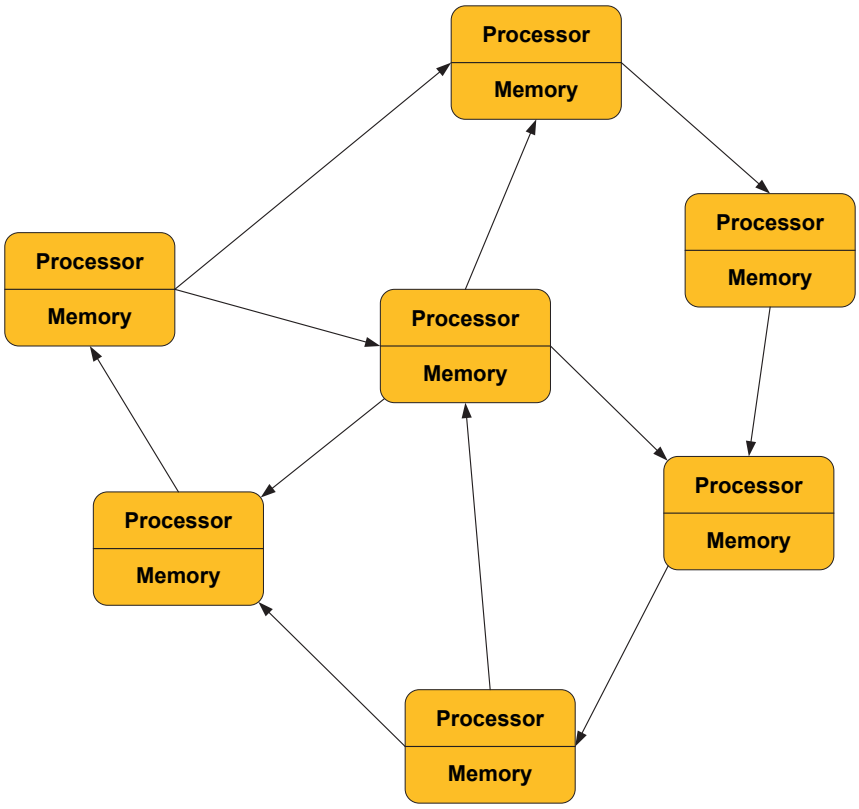


Figure 2.1: Distributed model of computation.

determined by the underlying computational communication pattern required to solve the given computational problem. In support of solving computational problems using the message passing model of computation, various programming models for distributed programming have emerged over the last two decades. The Message Passing Interface (MPI) and MapReduce are two leading examples that are in use today. A third model for distributed computing, which harvests Internet-based computing resources from personal computers is called *volunteer computing* and is described in Section 2.2.3.

2.2.1 Distributed Computing Based on MPI and the Open MPI Environment

The MPI environment consists of a de facto standard library and runtime system for the development, implementation and execution of both distributed and parallel computing applications based

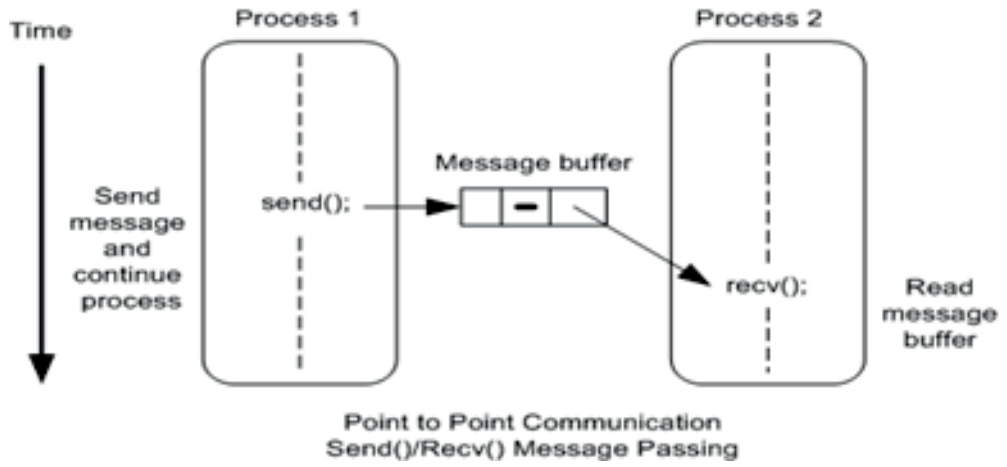


Figure 2.2: MPI send and receive operations.

on the message passing programming model. Within the MPI framework, a library of over 120 functions is available for all common operating systems. The MPI library is divided up into multiple categories of communication primitives; however the point-to-point and collective libraries, in both synchronous (blocking) and asynchronous (non-blocking) varieties, are the most common operations used by the majority of MPI applications. The basic point-to-point MPI operations between two host processes are *send* and *receive*. These are illustrated in Fig. 2.2. The more complex *collective* communication primitives involve groups of host nodes whose processes are all communicating within a single organized domain referred to as a *communicator*. Collective operations may be built using the fundamental point-to-point send and receive message operations as building blocks. In more optimized cases, however, the collective library implementation may utilize broadcast and multicast capabilities of the underlying communication network.

As an example of three common collective operations, Fig. 2.3 and Fig. 2.4 illustrate the MPI *broadcast* and *scatter-gather* operations. The broadcast operation is used for situations where a single host process wishes to transfer a single message to all host processes within a communicator. The scatter operation is utilized in situations where an array of messages must be distributed among a group of host processes from a single-source host process (each element of the local host process array is sent to a different remote host MPI process). The gather function is essentially the reverse operation, wherein a single host process constructs an array of messages from the group of host processes in the communicator (each element of the array is received from a different remote host

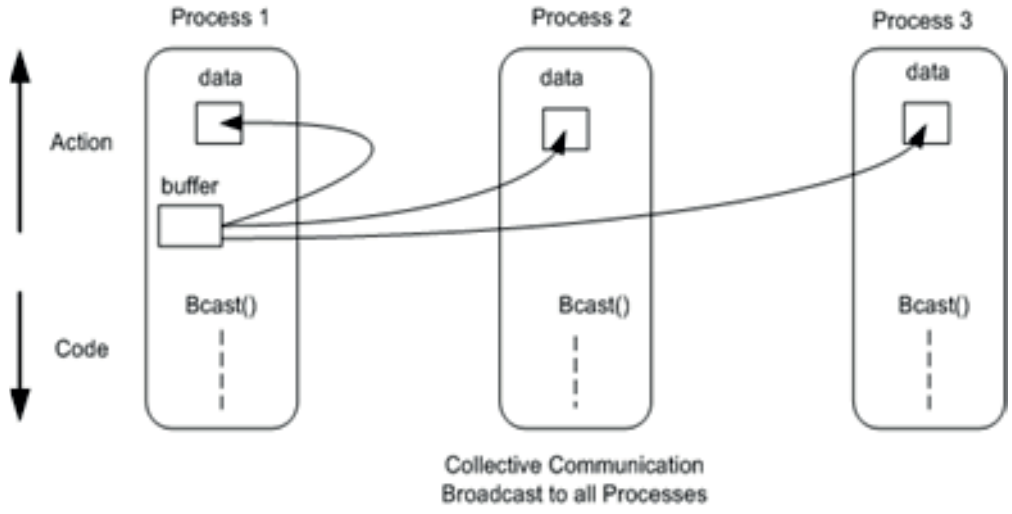


Figure 2.3: MPI collective broadcast operation.

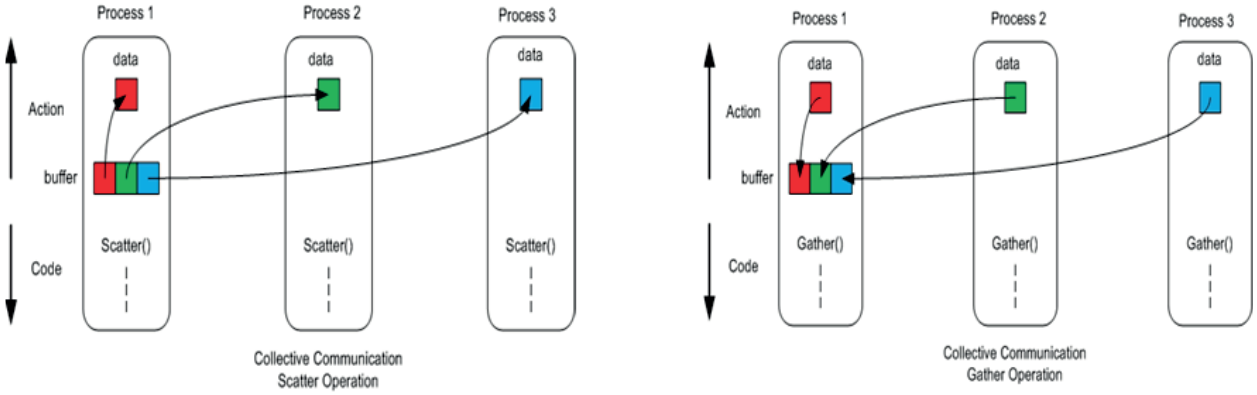


Figure 2.4: MPI collective scatter-gather operation.

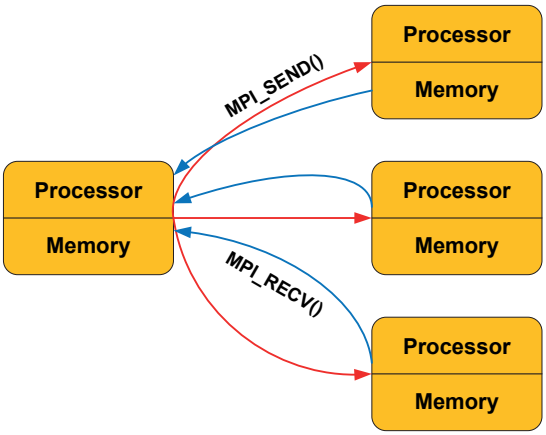


Figure 2.5: Basic MPI master-worker computational pattern.

MPI process). The MPI point-to-point and collective libraries provide a rich set of primitives, but they still represent a small subset of those available. The interested reader may find the complete set of library functions defined within the MPI standard specification [52].

A simple model for distributed computing is called the *master-worker* pattern and is represented in Fig. 2.5. The master-worker pattern is a typical MPI application communication pattern, where a single central system, or MPI host process often called the *master*, orchestrates the execution of a distributed application by sending both program and data messages to one or more MPI host processes called *workers*. When the workers have completed their task, results are sent back to the master MPI host process. In this scenario, where each MPI process executes and communicates independently of all other MPI processes, the MPI application is called *embarrassingly parallel*. Bioinformatic sequence alignment or Monte Carlo simulations are examples of embarrassingly parallel applications. However, more complex computational models often require message operations which involve data and communication *inter-dependence* among MPI host processes. In these cases, the application is not embarrassingly parallel; here distributed computation is among processor nodes and there is typically extensive use of MPI collective operations. An example is N-Body Simulation which solves for a dynamic system of particles under the influence of physical forces [49]; here multiple MPI processes communicate among one another in a nearest neighbor or

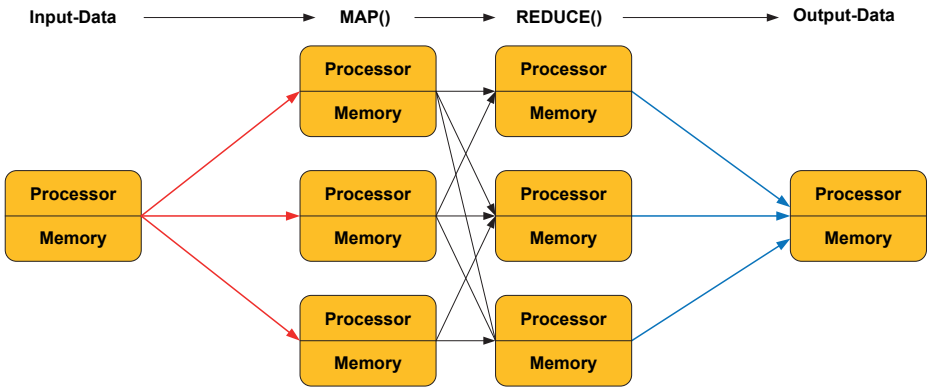


Figure 2.6: Basic MapReduce computational pattern.

lattice topology organization as they represent particles within the simulation domain [49].

Open MPI. The execution and lifecycle management of MPI parallel and distributed applications are handled through the MPI *runtime environment*. The leading Message Passing Interface library and runtime environment implementation is called Open MPI [100]. All Open MPI implementations include a library of communication routines that support point-to-point message buffer send and receive operations, as well as collective communication and computation operations that facilitate group-wide data movement and computation, respectively. In addition, Open MPI includes a runtime environment called Open Runtime Environment or ORTE, which enables concurrent execution of MPI based programs across all MPI processing nodes. Open MPI was developed over the last decade for distributed cluster computing and is now a de facto standard MPI implementation with a large base of software available for execution on most, if not all, distributed-memory, parallel computing architectures. Open MPI [42; 100] is highly modular, architected for inter-operability, portability, extensibility and scalability, and incorporates the best features of a number of earlier MPI implementations, thus making it an excellent choice for distributed memory computing platforms. Further details on the architecture and design of Open MPI will be given in the context of the experimental systems presented in Chapters 4 and 5.

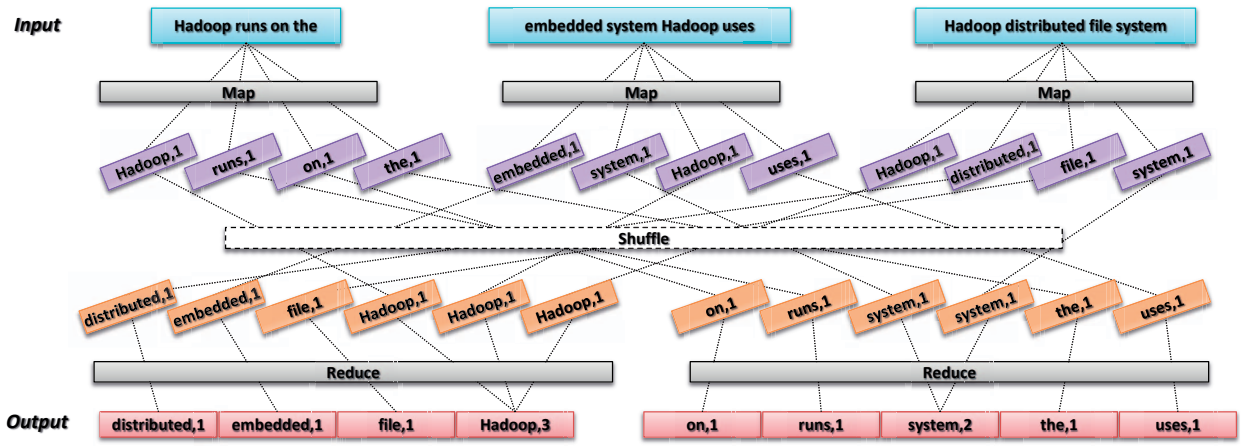


Figure 2.7: Example of MapReduce algorithm flow

2.2.2 Distributed Computing Based on MapReduce and the Hadoop Environment

While the MPI programming model and the Open MPI environment have gained de facto standard status with ubiquitous usage within the High-Performance Computing community, in recent years the MapReduce programming model, originally developed at Google [28] has also gained widespread acceptance, particularly for large-scale, data-intensive computing, analysis and search problems [82]. In particular, an open source version, called Hadoop, is seeing increased adoption in both industry and academia [82]. MapReduce is essentially a programming model where distributed computing algorithms are expressed in terms of two important operations: *Map*, and *Reduce*.

Computational problems expressed in terms of a distributed MapReduce algorithm follow a communication pattern similar to what is shown in Fig. 2.6. Here, a single master node controls and initiates the execution of map operations on a set of worker nodes, which compute input data into an intermediate format. Next, the output of the mapper node is passed to a second set of nodes that execute a reduction operation before computing and delivering a final output result. Programmatically, the MapReduce programming model is inspired by functional languages, but was developed into a large-scale parallel system by Google and, later, into a Cloud service by Yahoo. The MapReduce algorithm executes both Map and Reduce operations in two phases. Application-specific input-data is processed by the Map operation phase whose output is a set of intermediate <key,value> pairs. Next, a Reduce operation phase is applied to all intermediate pairs with the

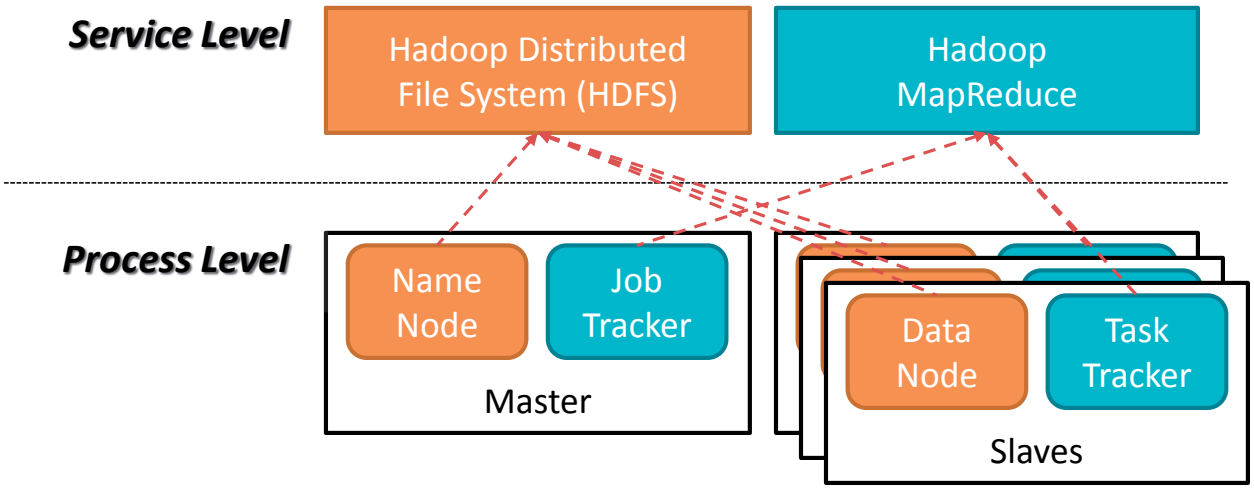


Figure 2.8: Hadoop MapReduce system architecture.

same key. The Reduce phase typically performs some kind of merging operation and produces zero or more output pairs [28]. Finally, the output pairs are sorted by their key value. As an example, Fig. 2.7 illustrates the MapReduce `wordcount` program, which produces as output, the number of occurrences of each word in an input text string. It operates as follows: first, during the Map phase, input text is tokenized into words and key-value pairs, where each key is a word from the input and the value is 1. For example the first four key-value pairs computed from the Map phase in Fig. 2.7 are *(Hadoop,1) (runs,1) (on,1) (the,1)*. Second, during the Reduce phase, all keys are grouped together and the values for similar keys are added. In the example shown, the first four key-values are grouped and summed, producing: *(Hadoop,3) (runs,1) (on,1) (the,1)*.

An architecture for the Hadoop MapReduce system is shown in Fig. 2.8. MapReduce systems are scalable due to the master-worker parallel architecture. Google [28] and Yahoo [117] utilize MapReduce extensively in providing Cloud-based services for both internal applications and to Internet-based client applications accessible through Web Services [75]. A key advantage of the MapReduce architecture lies in its loose coupling of components that makes them suitable for Virtual Machine implementation, thus leading to better scalability and fault-tolerance for some applications than traditional parallel computing models such as MPI [77].

Hadoop is an open source Java implementation of MapReduce that additionally includes a runtime environment for the execution of MapReduce programs written in the Java programming

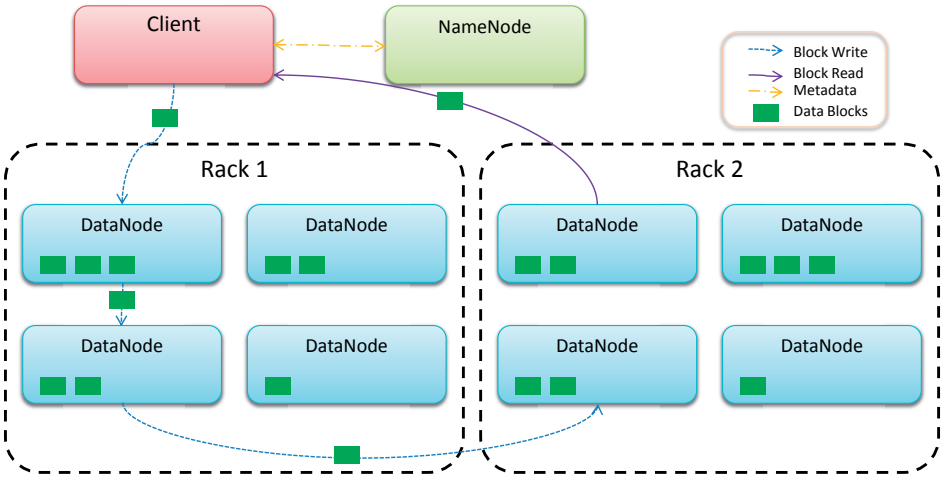


Figure 2.9: HDFS architecture.

language [51]. The Hadoop implementation of MapReduce makes extensive use of a distributed file system called the *Hadoop Distributed File System (HDFS)* as its underlying foundation. In comparison, the Google MapReduce implementation is based on a distributed file system providing similar capabilities referred to as the Google File System (GFS) [77]. The Hadoop system is based on a distributed master-slave architecture; is both highly-scalable as well as fault-tolerant through its use of task and data replication. The Hadoop core implementation is divided into two fundamental layers, HDFS [77] and the Hadoop MapReduce Execution system, that work together to execute MapReduce applications.

Hadoop Distributed File System. HDFS follows a master-worker architecture and is illustrated in Fig. 2.9 [17]. A single master node called the *NameNode* contains all metadata describing the HDFS filesystem and namespace along with the location of all blocks of data making up the filesystem. The blocks themselves reside on one or more worker systems called *DataNodes* which are responsible for managing data-block storage. The mapping of blocks in the filesystem namespace to DataNodes is determined by the NameNode. To store a file in this architecture, HDFS splits the file into fixed-sized blocks (typically 64MB) and stores them on the workers (DataNodes) [77]. The NameNode maintains the namespace metadata associated with the distributed filesystem, and all information regarding the location of input splits/blocks in all DataNodes. The NameNode is also responsible for executing all filesystem namespace operations like opening, closing and renaming files and directories [17]. Block replication and optimized block placement along with heartbeat

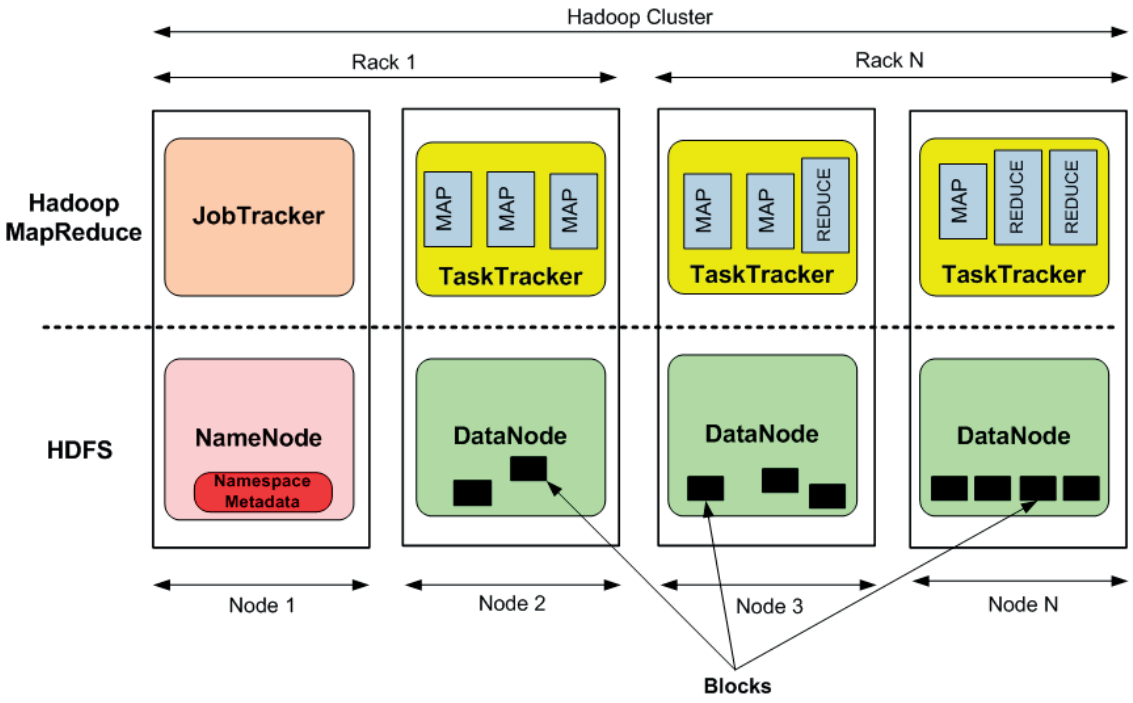


Figure 2.10: Hadoop HDFS and MapReduce architecture.

messaging are key features of the HDFS architecture. With block replication and placement, HDFS reliability is enhanced by storing multiple replica copies of a given block across multiple DataNodes. The blocks themselves are placed intelligently within the Hadoop cluster to minimize communication costs and latency. The Hadoop HDFS system is made fault-tolerant through the use of heartbeat messages between the NameNode and DataNode to assure DataNodes are functioning properly. In the expected case of DataNode failures, a DataNode rebalancing process is initiated. This automatically migrates and reorganizes the cluster to guarantee a minimum number of DataNode replicas to maintain Hadoop system stability. As shown in Fig. 2.9, to make use of the HDFS for MapReduce applications, clients read or write the HDFS by first requesting file access from the NameNode, which performs the appropriate filesystem operations (open, close, create, rename) on behalf of the client and responds with a list of DataNodes that contain the required block for the corresponding read or write access. The client then may perform read or write operations directly on the DataNode. Besides serving read or write requests from clients, DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode [17].

Hadoop MapReduce Engine. The MapReduce engine executes the actual Map and Reduce

functions of the client application, or *job*, and runs on top of HDFS as its data storage manager [77]. This is illustrated in Fig. 2.10. Similar to HDFS, the MapReduce engine also has a master-worker architecture consisting of a single *JobTracker* as the master and a number of *TaskTrackers* as the workers [77]. The JobTracker is responsible for monitoring and managing the MapReduce job across its execution over the Hadoop cluster and for assigning tasks to the TaskTrackers. Each TaskTracker manages the execution of the actual Map and Reduce tasks on a single computational node. A Hadoop cluster may consist of as little as one JobTracker and TaskTracker nodes, but in a typical cluster hundreds of TaskTrackers may exist. Each TaskTracker node has a number of simultaneous execution slots, each executing either a Map or a Reduce task [77] with a corresponding DataNode to provide block locality to minimize communication. In the example architecture shown in Fig. 2.10, a small two rack Hadoop cluster consists of four hosts that are typically commodity Linux-based x86 servers. Each Hadoop cluster must include a JobTracker and NameNode as shown in Node 1. The remaining nodes consist of three TaskTrackers for executing Map and Reduce tasks, along with three DataNodes.

2.2.3 Distributed Computing based on Volunteer Computing and BOINC

During the last decade a new parallel and distributed computing model called *volunteer computing* (also known as *public-resource computing*) has emerged for utilizing the large installed base of hundreds of millions of PC and gaming systems to solve computationally complex scientific problems. The key concept is to leverage the enormous number of idle processor cycles available throughout the Internet by users willing to donate unused personal system resources. As a form of Grid Computing, volunteer computing aggregates computational power in an organized fashion to solve a common problem [8]. However, unlike traditional Grid Computing, the computational resources of the public resource computing platform often exceed the computational power of most privately owned or government-sponsored grid supercomputer infrastructures [8].

There are many examples of volunteer computing initiatives, but perhaps the best known are SETI@Home [53] and Folding@Home [50], which data back to 1999 and 2001, respectively. Started at Stanford University in conjunction with Google, Folding@Home aims to advance research on *protein folding*, a very important problem in the field of computational biology. The SETI@Home project seeks to process large data-sets produced from radio-telescope signals in search of extrater-

restrial intelligence.

The implementation of volunteer computing systems centers around specialized software that is downloaded by consumers to participating PC systems or game consoles such as the PlayStation 3 [115]. For a given volunteer computing initiative, units of work or computational tasks that are required to solve a targeted problem are made available to users typically from within a screen saver or background agent running on the consumer device. In this model of computation, when the screen saver or agent is available it receives tasks assigned by a master server. These tasks are then processed by the consumer device on a best effort basis. Results are sent back to the master server when the task is completed. Compared to other types of high-performance computing, volunteer computing has a high degree of diversity [9]. The volunteered computers vary widely in terms of software and hardware type, speed, availability, reliability, and network connectivity [9]. While early volunteer effort relied on custom software implementations, most volunteer computing initiatives today have standardized around the BOINC (Berkeley Open Infrastructure for Network Computing) open source platform developed at U.C. Berkeley Space Sciences Laboratory [47].

BOINC Volunteer Computing Platform. The BOINC model involves *projects* and *volunteers* [9]. A BOINC project corresponds to an organization or research group that does public-resource computing [8]. It is identified by a single master URL, which is the home page of its web site and also serves as a directory of scheduling servers [8]. After downloading a special BOINC client to their personal system, participants register with projects that can involve one or more applications, which may change over time. A BOINC server architecture is illustrated in Fig. 2.11. The BOINC database is the central BOINC platform component and stores descriptions of applications, platforms, versions, work units, results, accounts, teams, and so on [8]. Server functions are performed by a set of web services and daemon processes. Scheduling servers handle RPCs from clients: they issue work and handle reports of completed results [8]. Data Servers handle file uploads using a certificate-based mechanism to ensure that legitimate results are transferred between the clients and the BOINC servers. All file transfers occur over HTTP. The BOINC system platform includes tools for creating, starting, stopping, and querying projects; adding new applications, platforms, and application versions, creating work-units and monitoring server performance [8]. Participants may join a BOINC-based project by visiting the projects web site, filling out registration form, and downloading the BOINC client [8]. The BOINC client may operate in one of several modes: as a

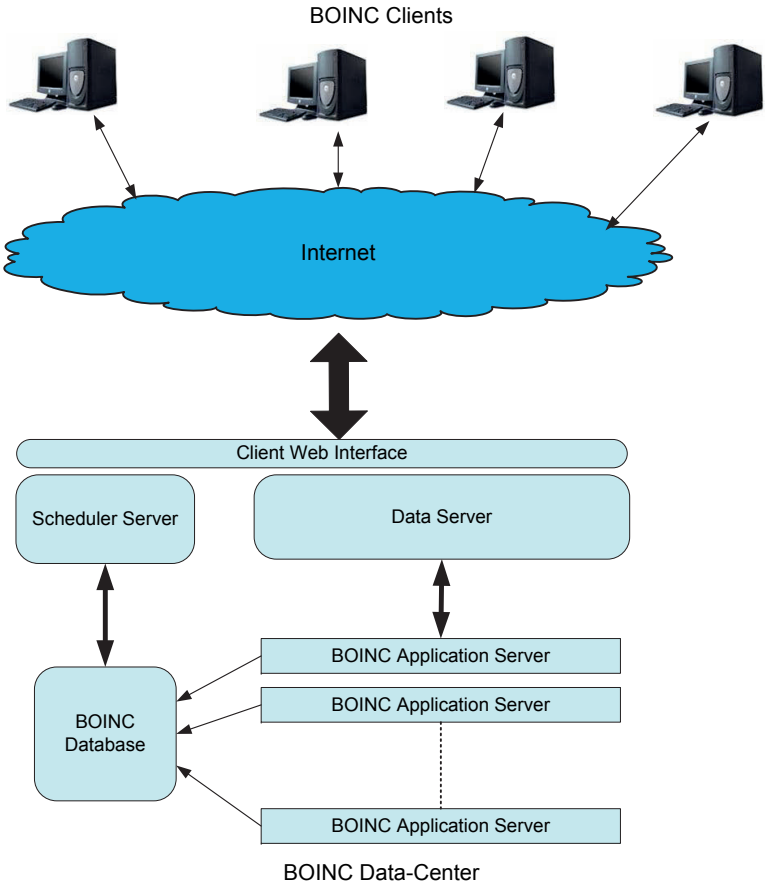


Figure 2.11: BOINC platform architecture.

screen saver that shows graphics of the running application; as a windows service which operates even when no users are logged in; and as a UNIX command-line program [8].

The BOINC platform has been adopted by a number of volunteer computing organizations that are supporting over 50 projects including Grid Republic [63] and the IBM sponsored World Community Grid [9]. As of 2011, over 6.2 million machines running the BOINC platform are members of the World Community Grid, which exceeded, in terms of teraflops, the fastest supercomputer (Tianhe-1A) in 2011 [103]. Similarly, the Folding@Home project has reported 5PFlops performance over 350,000 machines, as of March 7, 2009 [103].

While volunteer computing systems offer scalability on the order of the number of PCs or game consoles active on the Internet at any one time, in practice participation is not predictable and is

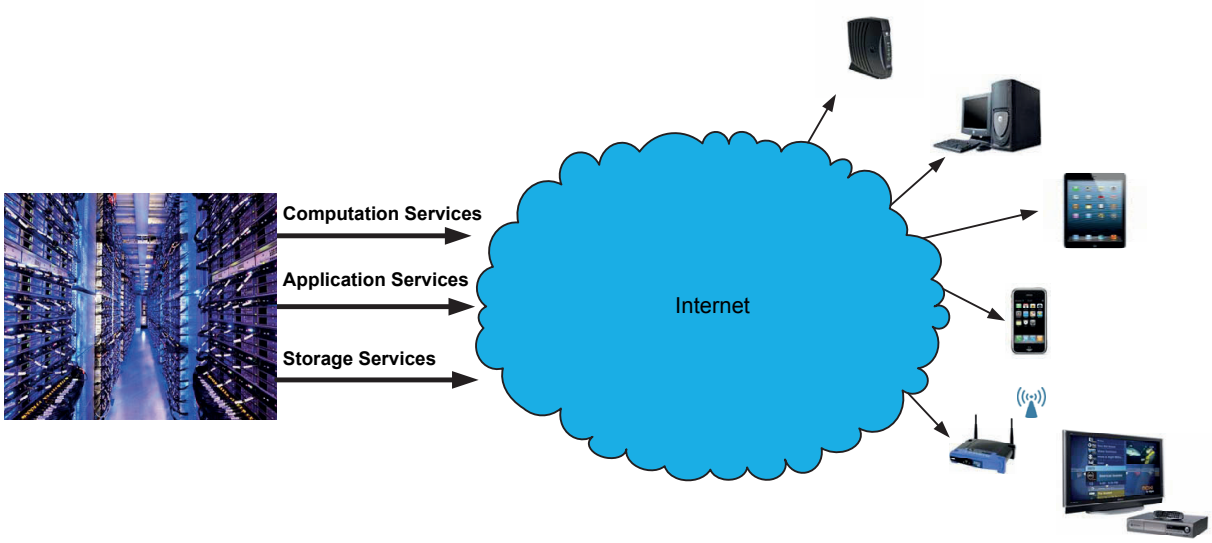


Figure 2.12: The Cloud model.

ad-hoc. This presents challenges in terms of offering a service level agreement (SLAs) that offer the same quality of service that Cloud-service providers now commonly guarantee. Despite these issues, distributed Internet computing has shown great success and highlights the potential of harnessing Internet connected devices in solving previously infeasible scientific problems and experiments [50; 53; 63].

2.3 Cloud Computing

Cloud computing can be thought of as a specialized form of distributed computing with the addition of service abstraction and virtualization of processor, storage, network, and applications that hide the underlying computational infrastructure to the extent required by external users of the cloud. A Cloud offers a large pool of easily usable, and accessible virtualized resources [116]. These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing for optimum resource utilization [116]. The pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the infrastructure provider by means of customized SLAs [116].

The term *Cloud computing* originally evolved as a concept where large aggregations of compu-

tational systems within one or more data-centers and other application specific software services are made available ubiquitously to users over a network, commonly the Internet [105]. This is illustrated in Fig. 2.12 [105]. Similarly, in the Berkeley definition [11], Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data-centers that provide those services [11]. In this definition, the data-center hardware and systems software is an instance of a Cloud [11]. The services themselves have long been referred to as *Software as a Service (SaaS)* [11], where SaaS is any software service available from a network accessible Cloud to users on-demand for a metered period of time. While the Berkeley definition is representative, there are in fact many definitions of the "Cloud" as well as of the term "Cloud Computing". However, the most commonly accepted formal definition is by NIST [92]:

"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

However, neither the NIST formulation, nor the interpretation of what it means, are universally accepted. A pragmatic approach is to understand common attributes of typical Cloud solutions [103]. NIST defines Cloud Computing by the following essential characteristics [92]:

- **On-Demand self-service.** A consumer can provision computing capabilities or services automatically without human intervention.
- **Ubiquitous network access.** Capabilities are available over the network and accessed through standard mechanisms.
- **Resource pooling.** The service provider computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to demand. Resources and services are generally transparent to the consumer.
- **Rapid elasticity.** Capabilities can be elastically provisioned and released. Resources appear unlimited to service consumers.
- **Measured service.** Cloud systems automatically control and optimize resource use by leveraging a metering capability as some level of abstraction appropriate to the type of service.

While these essential characteristics have evolved over time, the general evolution of Cloud Computing has been progressive with the emergence of new Cloud service and delivery models as described next.

2.3.1 Taxonomy of Cloud Services Delivery

The Cloud and its associated computing models are generally classified by their service delivery types and deployment scenarios. The most basic Cloud services delivery model follows a layered approached called the SPI model [92] or *Infrastructure as a Service (IaaS), Platform as a Service (PaaS)*, and previously defined *Software as a Service (SaaS).*

Infrastructure as a Service. Representing the lowest level of Cloud services, IaaS providers manage large sets of computational resources, such as compute nodes and storage systems. Through virtualization, these system elements may be dynamically managed and resized to build ad-hoc systems as demanded by customers [116]. Consumers utilize the IaaS resources by deploying and running arbitrary software which can include operating systems and applications [92]. The consumer does not manage or control the underlying Cloud infrastructure, but has control over operating systems and deployed applications [92]. Typically an IaaS provider supplies virtual machine images or different operating system flavors [103]. These images can be tailored by the consumer to run any given custom or packaged application [103]. Compute, storage and network bandwidth are consumable commodities in an IaaS environment and charged by CPU time, gigabytes-per-month, and bandwidth transit into and out of the Cloud system, respectively [103]. The Amazon Elastic Compute Cloud (EC2) is a classic example of IaaS [77].

Platform as a Service. Cloud systems can offer an additional abstraction beyond supplying a virtualized infrastructure. They can provide the software platform where consumers deploy applications based on a set of software services that include programming languages, libraries, application services and framework tools provided by the Cloud provider [116; 92]. The primary difference between IaaS and PaaS is the level of interaction required with the hardware and operating system platform. In PaaS, there is no interfacing or administration of virtual machines or infrastructure. With PaaS, the sizing of the hardware resources demanded by the execution of the services is made fully transparent [116]. Instead, the platform is abstracted, thus enabling the consumer to focus on application development. The trade-off, however, comes at the cost of less flexibility and

the requirement to develop applications in the specific environment support by the PaaS provider. Examples of PaaS providers are Google Apps Engine and Microsoft Azure [77].

Software as a Service. SaaS refers to services and applications that are available on an on-demand basis accessed through a network or the Internet. SaaS provides an alternative to locally run services and applications that in many cases cannot scale computationally or are otherwise infeasible. According to NIST [92], the SaaS capability enables the consumers to use the providers applications running on a cloud infrastructure [92]. The applications are accessible from various client devices through either a thin client interface, such as a web browser or tablet, or a program interface [92]. The underlying SaaS Cloud infrastructure is fully transparent to the consumer, hence SaaS provides the highest level of Cloud services. Salesforce.com is one of the best known examples of SaaS [77].

Since the number of Cloud service delivery models is ever increasing, the term *Everything as a Service* or *XaaS* is used to define the Universe of all Cloud service delivery models: it is a generic term to denote all service deliveries that exist and that might be created in the future [35; 107]. Esteves has offered one possible Cloud service delivery taxonomy [35]. As shown in Fig. 2.13, the root of the taxonomy is represented by XaaS since it encompasses all possible service deliveries. Each leaf corresponds to a subset of service deliveries defined by NIST including for completeness, *Business Process as a Service* which corresponds to the notion of outsourcing business or enterprise processes to the Cloud.

The Cloud model architecture can be illustrated as in Fig. 2.14: it consists of concentric layers with the physical infrastructure at the center, followed by progressively higher-levels of virtualization and abstractions built on top of one another as we move outward. The systems infrastructure or datacenter layer shown in Fig. 2.14 represents the physical computational and storage systems that implement the cloud hardware and software systems. This infrastructure may follow any number of computational models including: shared memory (typical of large multi-core symmetric multiprocessing servers), distributed memory, message passing based compute clusters (typically based on commodity Linux based server blades), or a hybrid combination of both. The programming model or software environment is generally determined by the service type and the Cloud provider. Common development models include anything from a consumer-defined software environment provided as IaaS in the form of a configurable, generic operating system, within a virtual machine instance,

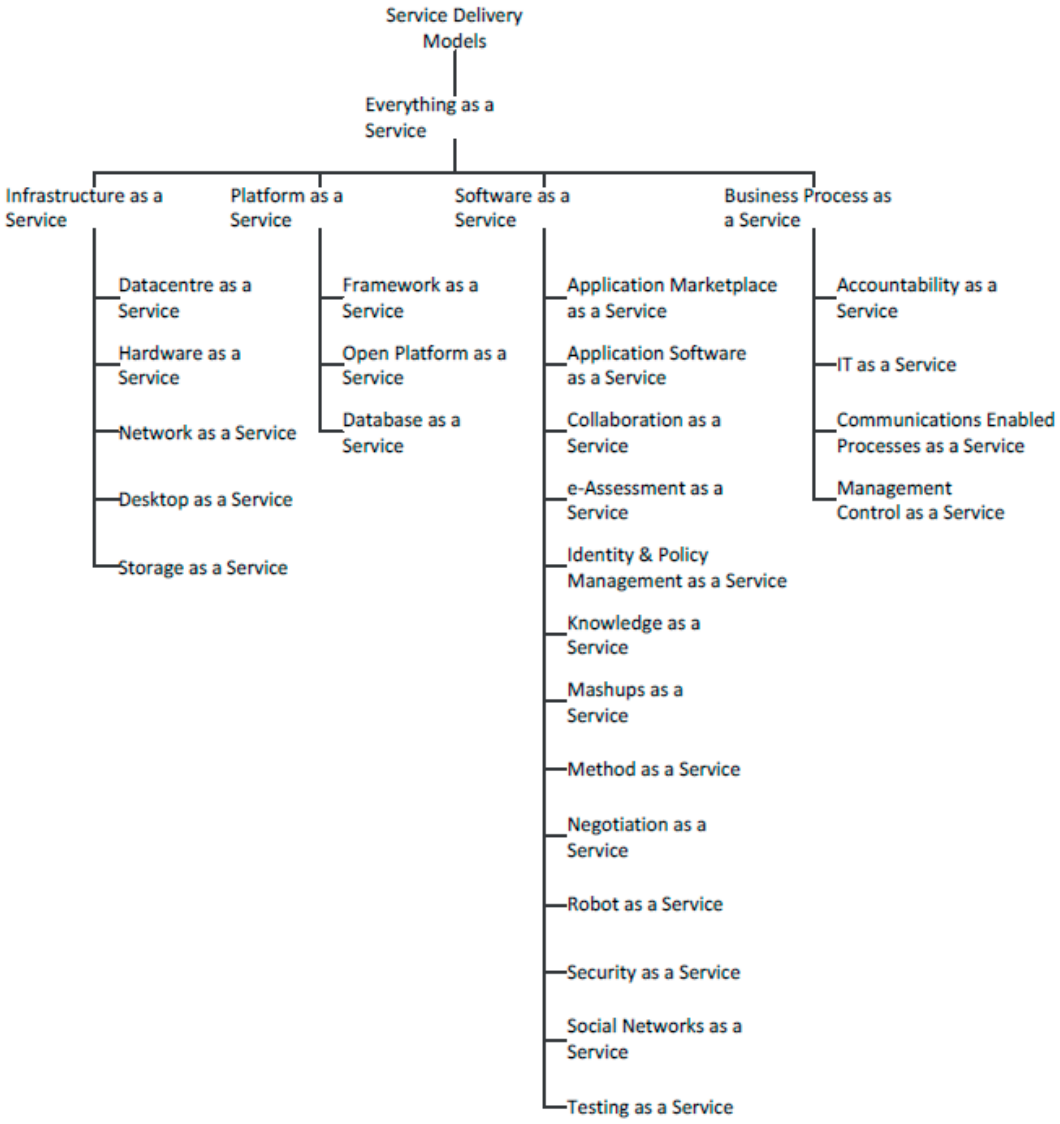


Figure 2.13: Cloud services delivery taxonomy[35].

to service provider instrumented platforms built around web and database services typically offered as PaaS. For larger scale or HPC applications, distributed and parallel programming models are available in any of PaaS, SaaS or XaaS. For example, Service Providers of data-intensive or computation-intensive Cloud services and applications support an infrastructure based on cluster computing leveraging MPI or Map-Reduce as described in Section 2.2. This form of Cloud service is illustrated in the example of Fig. 2.14. *An important point to note is that each Cloud ser-*

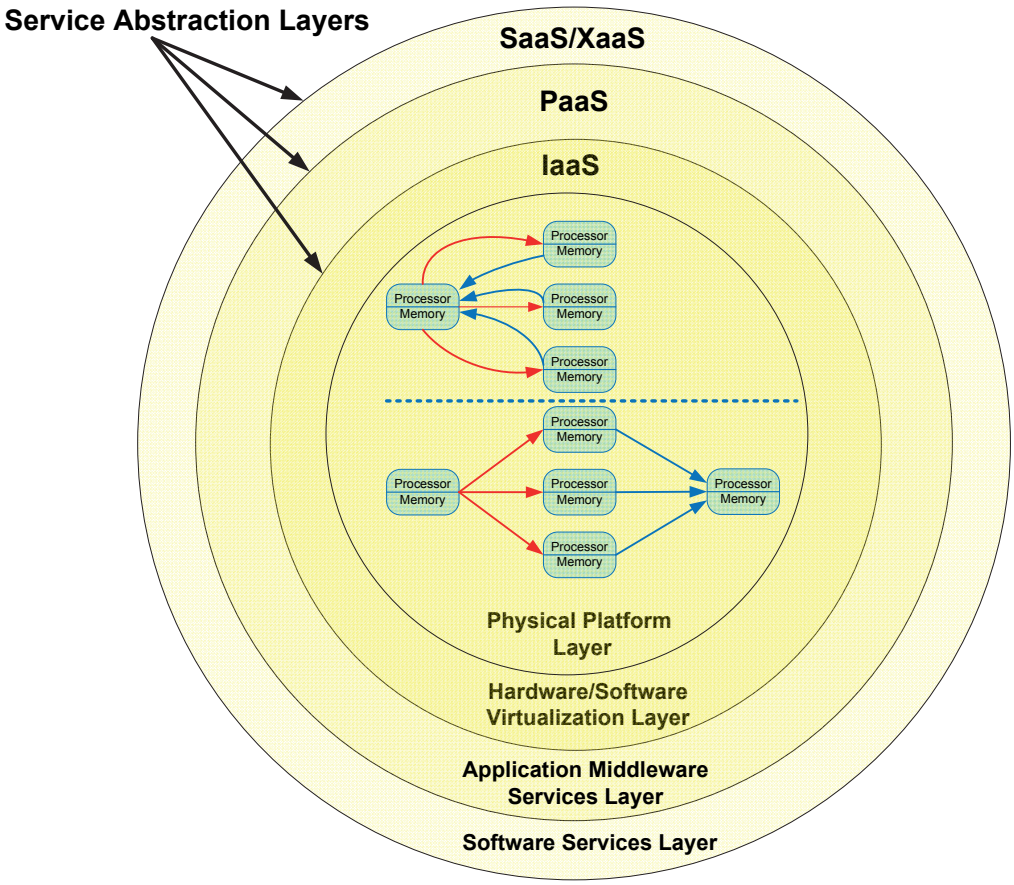


Figure 2.14: Cloud model as a layered architecture.

vice delivery method discussed (IaaS, PaaS, SaaS, or XaaS), has an implementation that is totally transparent to the consumer. Underlying an essential characteristic of the Cloud computing model, the consumers do not know if their Cloud application service requires a single processor core or a distributed system of tens of thousands of processor cores. Moreover, there is full location independence to the consumer as execution may be centralized or geographically dispersed among numerous computational systems.

2.3.2 Cloud System Deployment Models

In addition to service delivery types, access to Cloud services is defined by a set of Cloud deployment models which describe the manner and access scope provided by Cloud service providers. There are

four primary deployment models [92] defined as follows; 1) Public or external Clouds, 2) Private or Internal Clouds, 3) Community Clouds, and 4) Hybrid Clouds. Each is briefly described next.

Public Cloud. This Cloud infrastructure is provisioned for open use by the general public [92]. Many public Clouds are available, including Google App Engine (GAE), Amazon Web Services (AWS), Microsoft Azure, IBM Blue Cloud, and Salesforce.com's Force.com [77]. These providers are commercial providers and offer publicly accessible remote interfaces for accessing their respective Cloud services.

Private Cloud. In this deployment model, the Cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple business units [92]. It may be owned, managed, and operated by the organization or a third party [92]. It may be on premise or external.

Community Cloud. A Community Cloud infrastructure is provisioned for the exclusive use of a specific community of consumers from organizations that have shared concerns [92]. It may be owned and operated by one or more of the community members, or a third party.

Hybrid Cloud. The Hybrid Cloud deployment model is simply a composition of one or more of the other deployment models. It is composed of two or more distinct Cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technologies [92].

2.4 Broadband Cable Service Provider Systems

Among the largest organizations that utilize and operate massive computational and network infrastructures are broadband cable service providers, also known as Multiple Service Operators (MSO). Broadband cable service providers offer Internet access, voice-over-IP, and interactive digital video content across centrally managed broadband networks and systems to over 50 million consumer-electronic embedded devices at subscriber homes within the United States alone [67]. Compared to the United States, global broadband penetration is accelerating, and is expected to reach 600 million subscribers accessing over 1 billion embedded devices. China is leading the growth trend with an overall broadband cable adoption reaching 152 million subscribers [72].

Fig. 2.15 illustrates the distribution of the leading ten MSOs operating within the United States [67]. Table 2.1 lists the top ten number of subscribers associated to each of the MSOs pub-

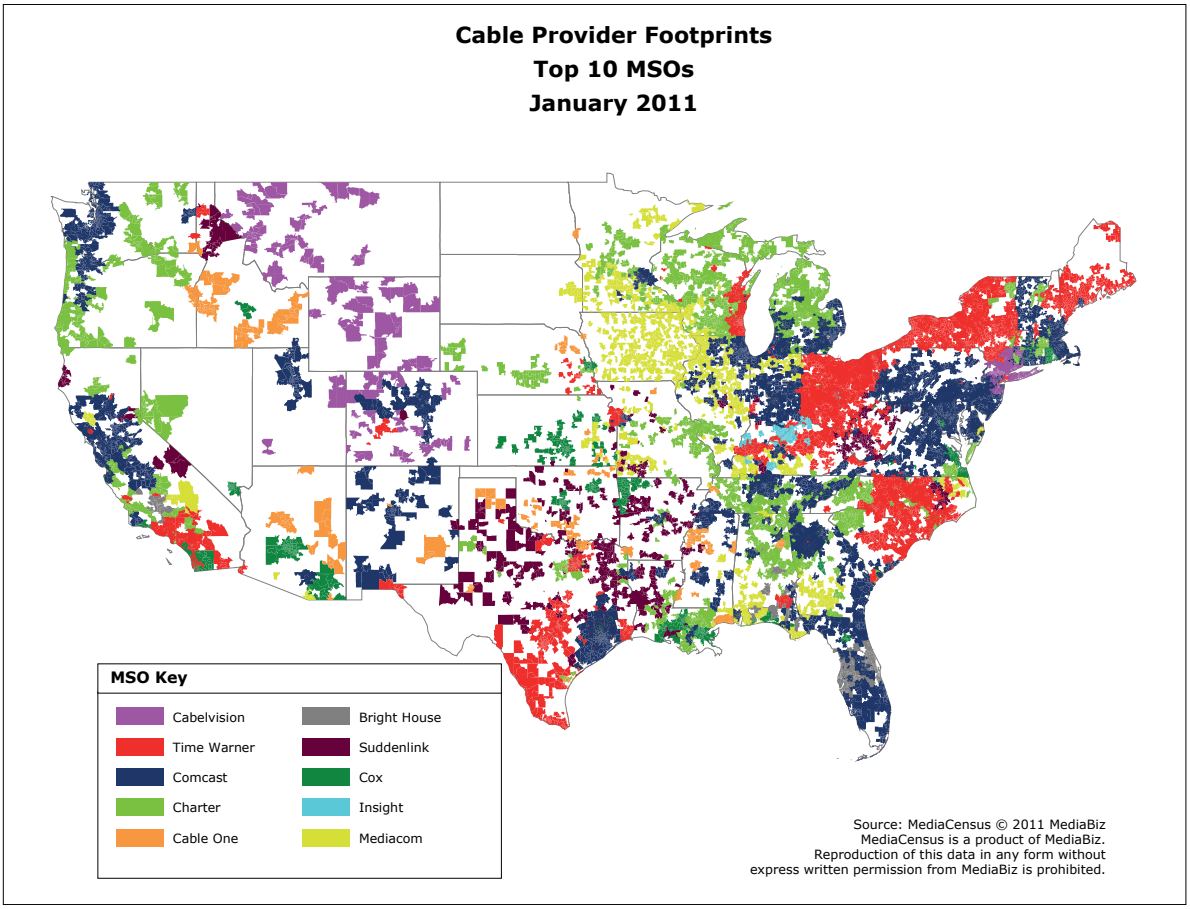


Figure 2.15: Distribution of top 10 broadband cable service providers in the U.S.

lished by the National Cable TV Association [70]. To deliver a wide range of interactive content and network services to a large population of managed embedded devices, MSOs have developed massive, distributed network and computational system infrastructures whose software and service delivery models are moving towards the Cloud system environment described in Section 2.2. In contrast, traditional High Performance Computing (HPC) environments have relied on largely homogeneous computing systems with emphasis on peak performance and throughput, optimized for complex scientific calculations in the fields of computational physics, chemistry and biology. Indeed, MSOs have implemented a heterogeneous system composed of multiple geo-diverse managed data-centers connected through broadband and fiber based network technologies that provide services to millions of distributed intelligent embedded devices. The remaining sections of this

Top 10 Broadband Cable Service Providers	
Comcast	22,360,000
Time Warner	12,109,000
Cox	4,789,000
Charter	4,371,000
Cablevision	3,264,000
Bright House	2,109,000
Suddenlink	1,268,000
Mediacom	1,100,000
Insight	670,000
CableOne	628,000

Table 2.1: Number of subscribers for top 10 cable service providers as of Sept. 2011

chapter describe the primary system components that comprise a cable service provider system, including the broadband network and the cloud infrastructure also referred to as *digital head-end*. Section 2.6.1 describes embedded devices and processor technologies typically managed or attached to cable service provider systems along with their evolution into the foreseeable future.

2.4.1 Broadband Cable Service Provider Network Technologies

Broadband cable service providers deliver multimedia and data services, including Internet access through a two-way Hybrid-Fiber RF cable (HFC) infrastructure. As shown in Fig. 2.16, all data communication occurs between a cable provider head-end or data-center and typically thousands of Cable Modem (CM) attached devices, which may include personal computers, WiFi access points for mobile and tablet devices, or other embedded devices such as *set-top boxes (STB)*. A STB provides for secure reception of premium video content and interactive applications such as program guides, search and video on-demand. At the head-end, a Cable Modem Termination System (CMTS) connects the cable modem network to the centralized Cloud. The cable model network is organized as a tree and branch structure (which is essentially a bus topology) with the rest of the cable service provider network. For Internet access and connections to any outside networks, both CMTS and CMs act as forwarding agents to transport the data transparently across the cable modem network.

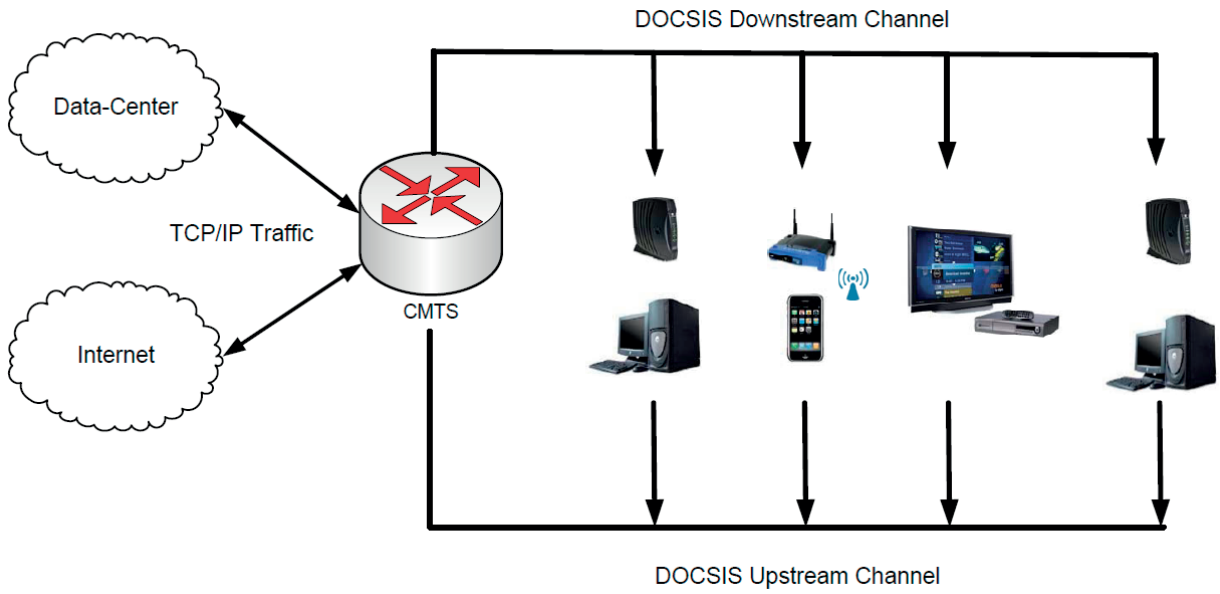


Figure 2.16: Broadband DOCSIS network architecture.

The CMTS interface to the cable modem network is through independent downstream and upstream communication channels, where each channel occupies a different RF frequency range (typically within a 6Mhz spectrum allocation) for its operation. The downstream channel is a one-to-many communication between the CMTS to all CMs, whereas the upstream channel is shared among all CMs and thus requires the CMTS to allocate and manage all CM transmission activities including bandwidth allocations and prioritization of network data. All management of network bandwidth in both downstream and upstream channels is through a Media Access Control (MAC) layer as defined by the Data over Cable System Interface RF Specification(DOCSIS) [57; 90].

The DOCSIS interface defines both the Media Access Control (MAC) layer as well as the physical RF communications layer. The original DOCSIS MAC layer interface (DOCSIS 1.0) provides a single best-effort service with simple prioritization using a contention-based request scheme [109]. The DOCSIS 1.0 physical layer supports a maximum downstream data rate of roughly 30.34 Mbps. Upstream channels of 3.2Mhz offer a maximum data rate up to 10.3Mbps that is shared by all CM devices using a TDMA scheme (Time Division Multiplex Access) [27]. DOCSIS 1.1, which is currently deployed by cable service providers, enhances the DOCSIS 1.0 MAC layer protocol with a set of quality-of-service (QoS) extensions enabling support for real-time and

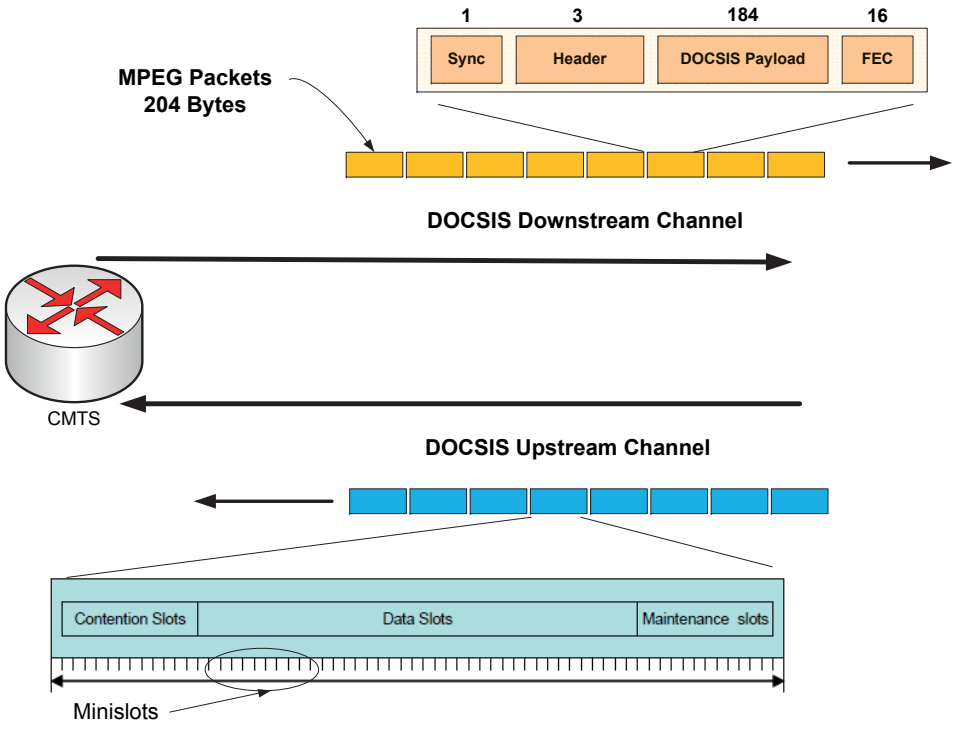


Figure 2.17: Structure of DOCSIS upstream and downstream packets.

constant bit rate services such as voice over IP (VoIP) and real-time video distribution. DOCSIS 2.0 supports all DOCSIS 1.1 capabilities, but increases the upstream capacity to 30Mbps through more advanced modulation techniques and by increasing the RF upstream channel allocation to 6.4Mhz [109].

2.4.1.1 Overview of DOCSIS Protocol

The structure of the DOCSIS downstream and upstream packets also referred to as *frames* is illustrated in Fig. 2.17. The downstream data is sent as a continuous stream of fixed size units (204 bytes) carefully chosen so that MPEG video frames and DOCSIS data frames can coexist within a single RF channel [27]. The DOCSIS data frame area is further broken into a 1 byte sync byte and a 3 byte MAC header that determines what type of payload is contained in the 184 byte DOCSIS

payload area. For example, the payload may carry CMTS management, timing synchronization messages, or standard TCP/UDP/IP data. The DOCSIS data frame ends with a 16 byte forward error correction code computed by DOCSIS transmission logic at the sender. In contrast, the upstream data is modeled as a stream of transmission opportunities or slots in the time domain, also called minislots. A single interval of upstream communication is broken into minislots that are a multiple of 6.25 microseconds long [27]. Depending on the system configuration, each minislot period may contain between 8 and 32 bytes of data. To manage the allocation of bandwidth to each CM, DOCSIS has a reservation-based, CMTS-centric TDMA approach for allocating bandwidth on the upstream channel, with a dynamic mix of contention and reservation-based transmission opportunities [1]. Using TDMA, multiple devices are granted data transmission opportunities to a shared communication channel through carefully controlled transmission time-slots. The CMTS and the CM network maintain accurate time synchronization relative to one another through periodic MAC layer ranging and timing messages [27; 1].

The lower portion of Fig. 2.17 illustrates the allocation of minislots during a typical upstream interval. CMs may use the contention minislots as defined by the CMTS for transmitting their requests, and the CMTS will allocate transmission opportunities for the CMs in the next interval frame, if capacity is available [90]. This contention model is referred to as a best-effort service, and is the only upstream allocation method defined in DOCSIS 1.0. For CMTS bandwidth management purposes, each CM has a unique MAC address that is mapped within the DOCSIS network to a Service Identifier (SID) [1]. There are four types of SIDs: 1) broadcast for identifying active upstream allocations that are available to all CMs; 2) multicast for a subset of CMs; 3) unicast for a specific CM; and 4) null, which is a special case meaning no intended CM's are targeted [109; 1]. Each SID instance is unique and identifies zero, one or more [1] CM devices and associates them to bandwidth allocations and QoS policies. DOCSIS CM upstream allocation methods along with QoS are described further in Sections 2.4.1.2 and 2.4.1.3, respectively.

The CMTS manages CM upstream transmission opportunities by periodically transmitting a MAC layer protocol bandwidth allocation message (called an allocation MAP message) on the downstream to all CMs. The allocation MAP is a MAC management message that defines three types of minislot allocations for upstream intervals: 1) minislots as data grants for particular CMs

[1]CM targets may be any of broadcast, multicast, unicast, or the special null case.

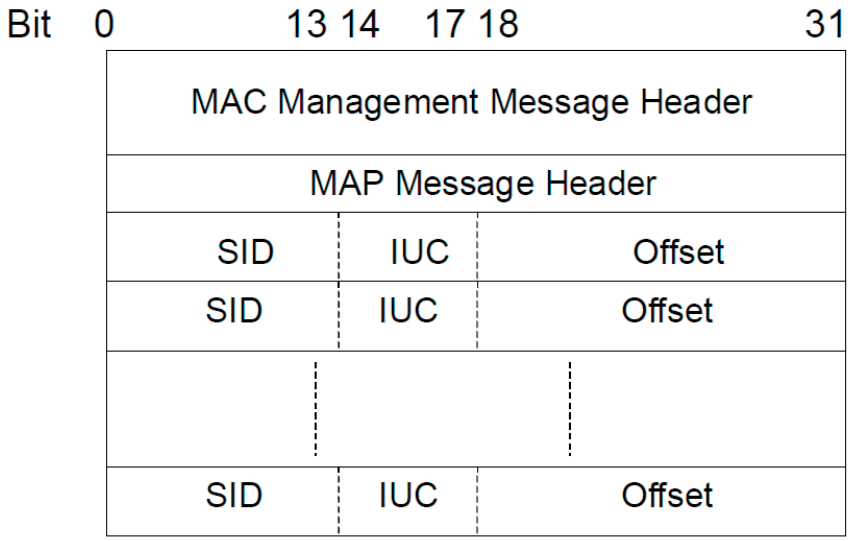


Figure 2.18: MAP message payload structure.

to transmit data, 2) minislots for contention request transmission (CMs requesting a data grant), and 3) minislots as an opportunity for new CMs to join the network.

Fig. 2.18 illustrates the layout of a MAP message payload [27; 1]. Each entry represents an allocation of minislots to targeted CMs (associated to SID) for a given upstream transmission interval. The allocation MAP message is a variable-length MAC message made up of a fixed MAC and MAP message header and a variable number of Information Elements (IE). Each IE defines the allowed usage for a range of minislots for a unique upstream interval [1]. The IE consists of a 14-bit SID, a 4-bit IUC type code and a 14-bit starting offset. The 14-bit SID specifies the targeted set of CM devices mapped to a unique service class. The 4-bit type code defines the type of traffic carried during the interval, and the 14-bit offset, the starting time for the interval.

A more detailed structure of the IE is shown in Table. 2.2. The IUC field determines the IE type as shown in the first column of Table. 2.2. A *Request* IE provides intervals in which bandwidth-requests may be made for upstream transmission. The *Request/Data* IE provides an interval for either bandwidth requests or short data packets. The *Initial Maintenance* IE provides intervals in which new stations may join the DOCSIS network, where *Station Maintenance* IEs provide opportunities for existing CM maintenance operations. Data grants are provided to CMs using the *Short and Long Data Grant* IE. These grants provide an opportunity for a CM to transmit one or more upstream DOCSIS frames (recall the DOCSIS frame contains a MAC header followed by a

IE Name	Interval Usage Code (IUC) (4 bits)	SID (14 bits)	Mini-slot Offset (14 bits)
Request	1	any	starting offset of REQ region
Request/Data	2	multicast	starting offset of IMMEDIATE Data region
Initial Maintenance	3	broadcast/multicast	starting offset of MAINT region
Station Maintenance	4	unicast	starting offset of MAINT region
Short Data Grant	5	unicast	starting offset of Data Grant assignment
Long Data Grant	6	unicast	starting offset of Data Grant assignment
Null IE	7	zero	ending offset of the previous grant. Used to bound the length of the last actual interval allocation
Data Ack	8	unicast	CMTS sets to map length
Reserved	9-14	any	reserved
Expansion	15	expanded IUC	# of additional 32-bit words in this IE

Table 2.2: Information Element (IE) structure.

MAC Ethernet frame). Finally a *Null* IE is used to terminate the list of IEs. Within each IE the 14-bit offset field defines the specific mini-slot assignment in terms of the time from the beginning of the minislot interval for a given IE entry [27; 90; 1].

2.4.1.2 DOCSIS Upstream Bandwidth Allocation

All DOCSIS network devices acquire upstream bandwidth allocations and transmission opportunities from the CMTS through MAP messages as described in Section 2.4.1.1. The actual CM requests to the CMTS occur through a number of explicit request mechanisms that include any of the following three kinds [20]:

Unicast Requests Polls. The CMTS may send periodic request opportunities as means of real-time polls regardless of network congestion to allow CMs to transmit data without requiring a contention request process. These transmission opportunities are included in the downstream MAP messages [20].

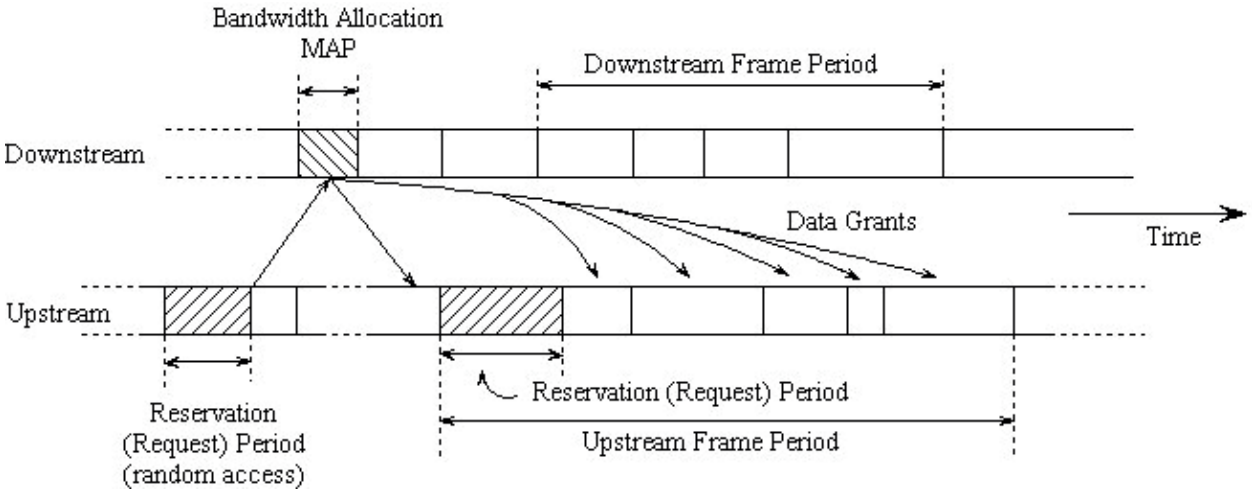


Figure 2.19: Contention reservation request and MAP grants[20].

Piggybacking Requests. Piggybacking is a method defined by DOCSIS where CM requests for additional bandwidth are sent within scheduled upstream data transmissions. Piggybacking reduces contention, since the CM bandwidth requests are transmitted within the data packets [20].

Contention Based Requests. Within each upstream transmission interval, a portion of minislots are allocated for contention-based requests that may collide with one another, but are available to all CMs. Contentions are resolved by a Contention Resolution algorithm defined by the DOCSIS standard. The CMTS controls assignments on the upstream channel through the MAP MAC message and determines which minislots are subject to collision. Contention resolution is based on a Truncated Binary Exponential Back-off, with the initial back-off window and maximum back-off window controlled by the CMTS [27; 20]. These values are specified as part of the MAP message. When a CM has information to send and wants to enter the contention resolution process, it sets its internal back-off window equal to the back-off start defined in the MAP currently in effect. The CM then randomly selects a value within its back-off window. This random value indicates the number of contention minislots the CM must wait before transmitting [20]. After a contention transmission, the CM must wait for a *Data Grant* (Data Grant Pending) or *Data Ack* in a subsequent MAP, which completes the contention resolution process. The amount of data the CM may then transmit, after the Data Grant is received, is defined by the request IEs or Request/Data IEs contained in the MAP message. Fig. 2.19 illustrates the relationship between MAP and Data Grants for contention based bandwidth requests. In the event a CM contention message is lost due

to a collision, the CM determines that it was lost when it finds a MAP without a Data Grant or Data Ack for it. The CM must now increase its back-off window by a factor of two (as long as it is less than the maximum back-off window) [20]. The CM then selects a new back-off value within this window and retries the process described above. This process may occur up to a maximum number of retries (16 as defined in the DOCSIS specification). After this the upstream request must be discarded [20].

2.4.1.3 DOCSIS Quality-of-Service (QoS)

A single QoS service class is supported in DOCSIS 1.0 using service identifiers or SIDs (see Section 2.4.1.1). The CMTS assigns SIDs to CMs as part of managing upstream communication, in conjunction with best-effort data transmission through the contention arbitration described previously. In DOCSIS 1.0 networks, each CM is assigned only one SID for both upstream and downstream directions, creating a one-to-one correspondence between a CM and its SID. This limits the flexibility of DOCSIS 1.0 networks, since all traffic is treated the same, independently of the service level and communication bandwidth requirements [27; 60].

DOCSIS 1.1 (as well as DOCSIS 2.0 and higher versions) enhances DOCSIS 1.0 with the concept of service flows and introduces several new QoS concepts including: 1) Packet Classification and Flow identification; 2) Service Flow QoS Scheduling; and 3) Fragmentation and Concatenation. The multiple QoS service flow classes now apply independently to both downstream and upstream communication channels, thus adding more flexibility in managing DOCSIS network bandwidth and services [60]. The DOCSIS 1.1 QoS framework is based on the following objects: Service Flows, Service Class and Packet Classifiers.

Service Flows. A service flow is a MAC layer transport service that provides unidirectional transport of either upstream frames transmitted by the CM or downstream frames transmitted by the CMTS. A Service Flow is characterized by a set of QoS parameters such as latency, jitter, and throughput assurances. In order to standardize operation between the CM and CMTS, these attributes include details on how the CM requests upstream minislots and the expected behavior of the CMTS upstream scheduler [20; 60]. DOCSIS 1.1 service flows are identified and assigned to CMs using a Service Flow Identifier (SFID) which extends the DOCSIS 1.0 Service Identifier or SID functionality specifically for QoS management. Note, however, that DOCSIS 1.1 continues to

use the term SID, but it only applies exclusively to upstream service flows. Every CM establishes a primary service flow for the upstream and downstream directions with separate SFIDs for the purpose of maintaining connectivity with the CMTS for DOCSIS MAC and management messaging [60]. DOCSIS 1.1 also allows for multiple service flows assignments that can be created either statically or dynamically to meet the needs of on-demand traffic. Multiple service flows enable CMs to support a combination of data, voice, and video traffic concurrently while optimizing bandwidth utilization [60].

Service Class. A service class defines a collection of settings maintained by the CMTS that provide a specific QoS service tier (such as maximum bandwidth or traffic priority) to a CM that has been assigned a given service flow associated with that service class [1; 60]. The DOCSIS 1.1 service class also defines the MAC layer scheduling type for the service flow. The schedule type defines the type of data requests that the CM can make, how often it can make those requests, as well as the priority the CMTS determines for granting transmission opportunities to all CM devices [60]. A CM can also be assigned multiple service flows, allowing it to have multiple traffic flows that use different service classes [60].

The following schedule types are supported:

- **Best-Effort (BE).** A CM competes with the other CM devices in making bandwidth requests and must wait for the CMTS to grant those requests before transmitting data. This service flow is similar to the default method in DOCSIS 1.0 networks for upstream bandwidth allocation [60].

- **Real-time polling service (rtPS).** A CM is given a periodic time-slot in which it can make bandwidth requests without competing with other CM devices. This class-of-service allows real-time transmissions with data bursts of varying length [60].

- **Non-real-time polling service (nrtPS).** A CM is given regular opportunities to make bandwidth requests for data bursts or varying size. This flow is similar to rtPS except the CMTS can vary the time between polling of the CMs, depending on the current traffic and network congestion [60].

- **Unsolicited grant service (UGS).** In the UGS service, CMs can transmit fixed data bursts at a guaranteed minimum data rate and with a guaranteed maximum level of jitter. This

type of service flow is suitable for traffic that requires a Committed Information Rate (CIR), such as Voice-over-IP (VoIP) [60].

- **Unsolicited grant service with activity detection (UGS-AD).** The UGS-AD service class is similar to UGS. Here, however, the CMTS monitors the traffic to detect when the CM is not using the service flow, in which case the CMTS switches the service flow to a rtPS type. When CMs begin using the flow again, the CMTS switches the flow back to UGS class of service. The UGS-AD class allows the CMTS to manage unused bandwidth more effectively during time-periods when the UGS service flow is not required [60].

Packet Classification. In DOCSIS 1.1 networks, CM devices can use multiple service flows, each with a different level of service. To quickly assign upstream and downstream packets to their proper service flows, the CMTS uses the concept of *Packet Classification* and *Packet Classifiers*. Packet classification describes a process executed at the CMTS, whereby a set of packet-header fields are used by the CMTS network processor to classify packets onto service flows, establishing the appropriate data-transmission priority for the given service flow. Each packet classifier specifies one or more packet header attributes such as source MAC address, destination IP address, or protocol type [60].

Fig. 2.20 illustrates the logical structure for DOCSIS packet classifiers and their relation to upstream and downstream service flows. When the CMTS receives downstream and upstream packets, it compares each packets headers to the contents of each packet classifier. When the CMTS matches the packet to a classifier, the CMTS then assigns the proper SFID to the packet and transmits the packet to or from the CM. If no matching classifier is found, the packet is forwarded to the primary default service flow. Packet classification ensures that all packets are assigned to their proper service flows and thus proper QoS characteristics [27; 60].

Fragmentation and Concatenation. DOCSIS 1.1 introduced two other features to support better QoS operation; fragmentation and concatenation. Fragmentation splits large data packets so they can fit into smaller time slots and is enabled on a per service flow basis. When enabled for a service flow, fragmentation is initiated by the CMTS when it grants bandwidth to a CM with a grant size that is smaller than the corresponding bandwidth request from the CM. Concatenation enables a CM to combine multiple upstream packets into one larger MAC data frame, allowing the

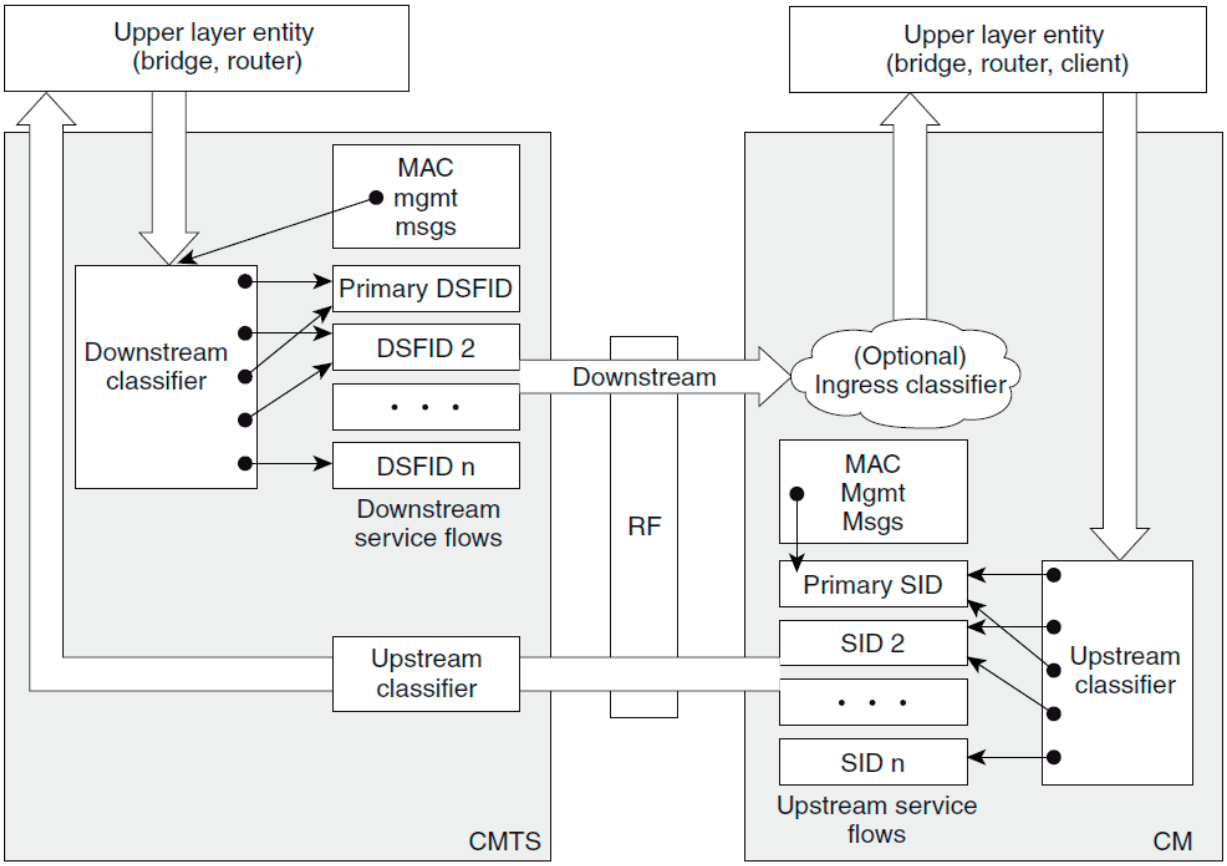


Figure 2.20: DOCSIS packet flow classification[59].

CM to make only one time-slot request for the entire concatenated frame [27; 60]. Fragmentation and concatenation together, enable better use of upstream resources and improve throughput.

2.4.2 Broadband Cable Service Provider Network Architecture

Fig. 2.21 shows a high-level architecture of a typical broadband cable system network that supports approximately 8 million consumer electronic (CE) embedded devices. Three regional data centers, or *head-end facilities* are shown, each containing all necessary infrastructure to operate a complete regional system that typically spans hundreds of miles and services 1 to 3 million subscriber devices. A vast fiber interconnection network is designed to transport both digital video and TCP/IP data traffic between the data-centers and consumer devices, as well as network peering points including: content service providers, third-party clouds that offer a variety of services, and the Internet at

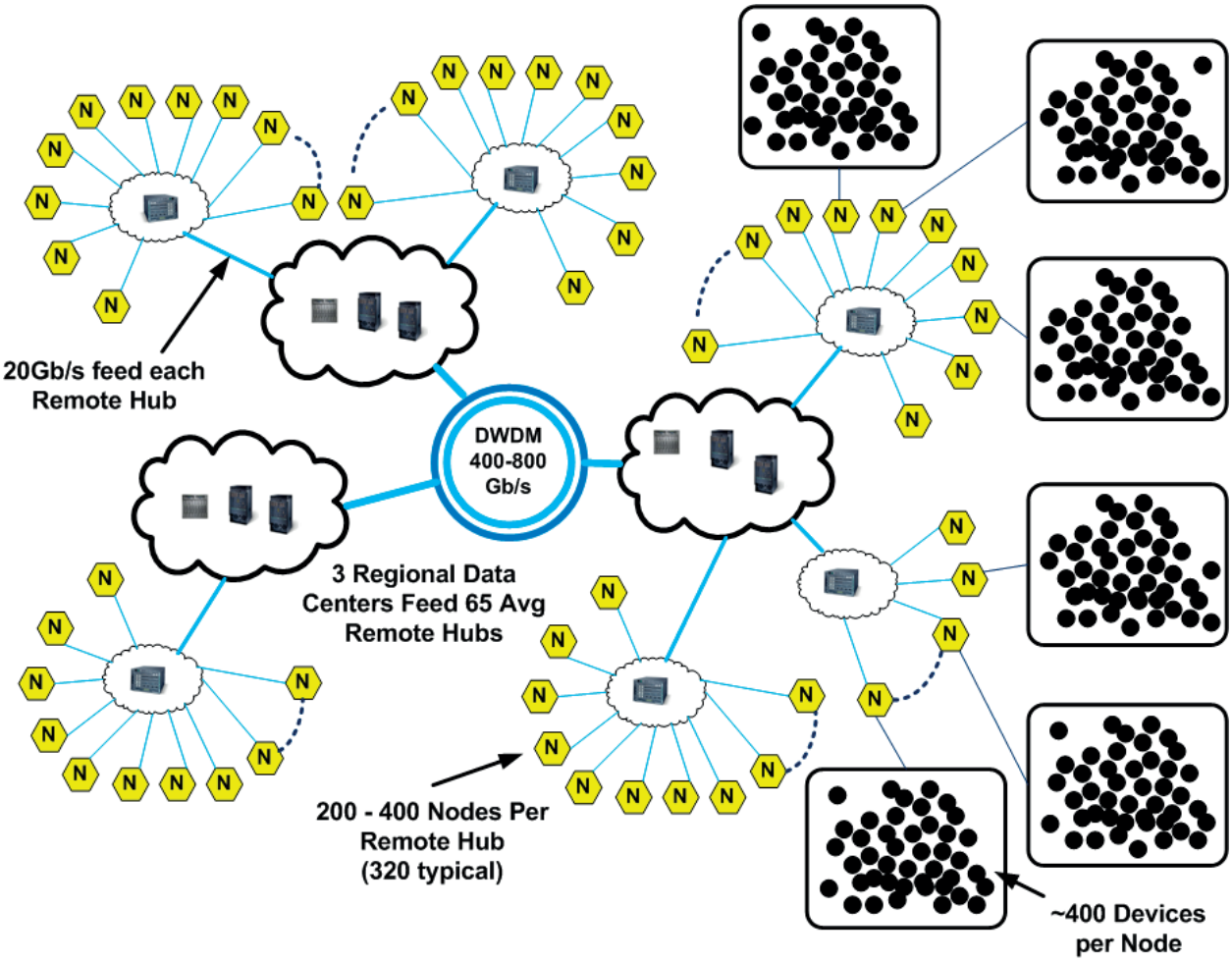


Figure 2.21: Example of a network architecture of a broadband cable service provider.

large. The regional head-ends are typically interconnected using a dense wavelength-division multiplexed (DWDM) fiber 400 Gb/s transport network capable of expansion up to 800Gb/s bandwidth, enabling video and data distribution. The distribution network provides for movement between regional head-ends with very low-latency transmission delay typically on the order of a few milliseconds. The DWDM network is configured as a full mesh-topology, providing a fault-tolerant environment, where loss of communication on any path will not result in a loss of communication services.

In order to scale video and data distributions to millions of devices, each regional head-end data-center feeds approximately 65-75 smaller *remote-hubs* that are geographically dispersed. Each remote-hub is typically fed with 20Gb/s bandwidth via DWDM communication links. Some remote-

hubs support smaller device populations, and therefore require less network bandwidth and fiber capacity. The remote-hub is primarily designed to segment or partition the total device population into smaller device population clouds to ensure reasonable network bandwidth and traffic scalability. Each remote-hub drives 200 to 400 physical-layer translation devices or *nodes*. A node is a physical-layer distribution element containing fiber-to-RF, RF-to-fiber converters and RF amplifiers, where devices such as set-top boxes, cable-modems, or even WiFi access nodes are physically interconnected in a tree topology resembling a bus interconnection network. It is important to note that all embedded devices on a cable service provider system are connected to nodes via RF coax cables. The nodes themselves connect back to the remote-hubs over fiber. Nodes do not provide any layer 2 switching or routing functions, but due to their role in converting between fiber and RF, the term *hybrid fiber/RF network* is used when describing a cable network system. In the example shown in Fig. 2.21, each hub feeds approximately 320 nodes, with each node in turn feeding approximately 400 subscriber homes.

Remote-hub Architecture. Each of the remote-hubs illustrated in Fig. 2.21 is interconnected using a fiber-based optical network and supports physical-layer communication for approximately 80,000 to 160,000 subscriber devices. The remote-hub provides segmentation or partitioning of physical-layer communication services for both data and video content. Interconnection between the regional head-ends and remote-hubs is via long distance, single-mode fiber optic cabling using DWDM technologies. DWDM enables the multiplexing and demultiplexing of multiple optical carriers or channels over a single fiber using multiple light wavelengths. DWDM enables distribution of multiple data and video communication flows over distances exceeding 100km [58].

The internal architecture of a remote-hub is illustrated in Fig. 2.22. A DWDM fiber switch connecting the remote-hub with the central head-ends demultiplexes both video signals and data traffic. The video signals are converted into RF signals and all data traffic is sent to a Gigabit Ethernet network switch. The network switch supports multiple Gigabit Ethernet networks each of which, supports one or more DOCSIS routers. The output of each DOCSIS router consists of one or more RF signals, which are re-combined with all the video RF signals from the DWDM switch, producing a single combined RF output signal. This combined RF signal is converted back to a pair of optical signals for bidirectional optical fiber communication to and from fiber/RF node devices. The fiber/RF node device converts the optical signal back into RF, feeding a bus like RF

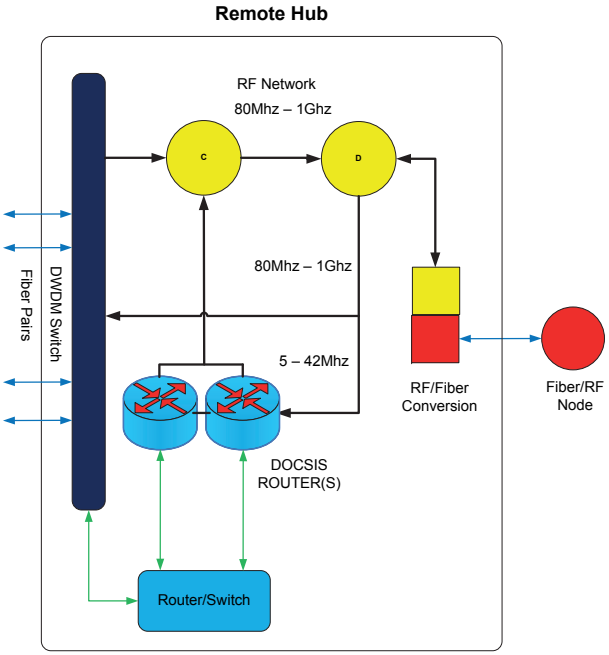


Figure 2.22: Remote-hub architecture.

cable network topology where each subscriber premise is attached.

Fig. 2.23 illustrates the layout of a typical internal data network within a remote-hub. The internal data network is typically supplied with 20Gb/s aggregate communication link bandwidth from the larger regional head-end data centers via the DWDM switch. Since each remote-hub may contain on average 5 to 10 DOCSIS routers, a router/switch is utilized to aggregate all traffic from multiple networks between the DWDM fiber switch and the DOCSIS router devices as shown in Fig. 2.23. The network feeding each DOCSIS router is actually formed by multiple 1Gb/s Ethernet networks that are bonded together. For example in Fig. 2.23, each DOCSIS router is supplied 4Gb/s overall bandwidth. This ensures adequate bandwidth on the DOCSIS RF network side of the system. The specific number of DOCSIS routers within a given remote-hub is determined by the number of nodes fed and associated bandwidth requirements for the given region. In general, the trend is towards much higher bandwidth to subscribers, therefore remote-hub data networks are moving to newer 10Gb/s Ethernet links and consequently the input to the DOCSIS router is

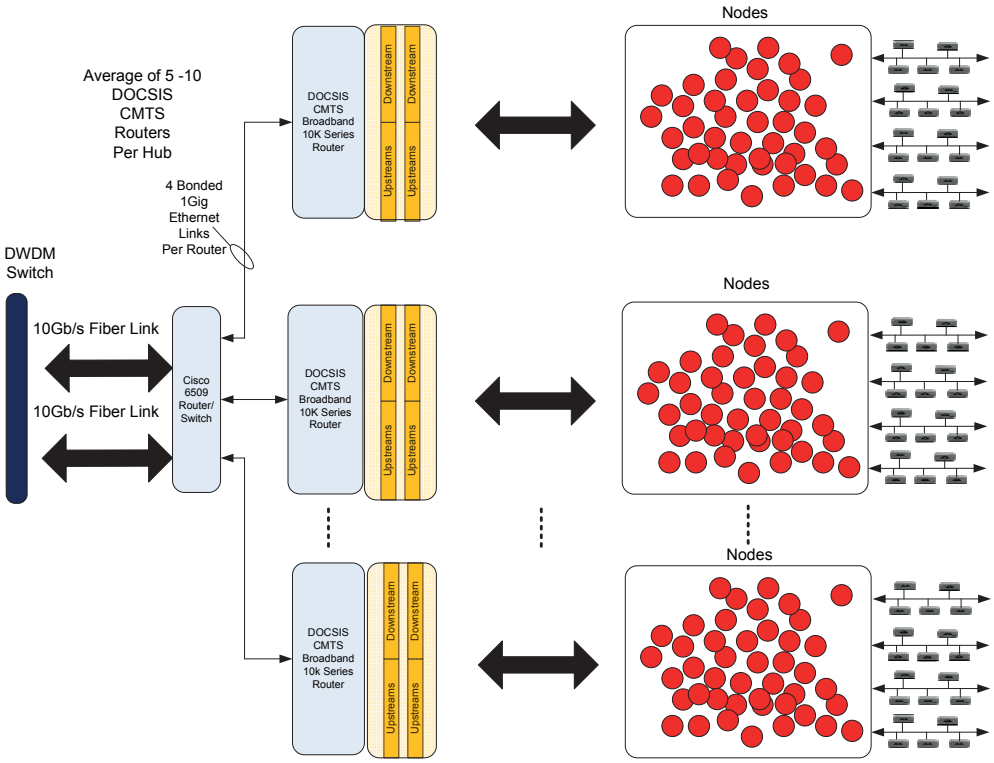


Figure 2.23: Remote-hub data network architecture.

also moving to 10Gb/s interfaces. The broadband RF network terminates into the DOCSIS router devices, where it is then converted to standard Ethernet TCP/IP protocols. The broadband RF network is based on the DOCSIS MAC layer protocols as described further in Section 2.4.1.1.

Each DOCSIS router contains a number of DOCSIS network line cards that drive the hybrid fiber/RF service provider cable plant using RF modulation techniques across fiber interconnects. Fig. 2.24 illustrates a representative line card. In this example, each line card supports 5 downstream and 20 upstream interfaces that feed multiple hybrid fiber/RF node devices. Assuming an average of 2 line cards per router, approximately 40 nodes are supported for each CMTS router enabling DOCSIS communication support for up to 20,000 broadband embedded devices. Given 5 to 10 DOCSIS routers per remote-hub, approximately 100,000 to 200,000 subscriber devices may be supported per remote-hub.

The DOCSIS physical layer network driven by each of the hybrid fiber/RF nodes, supports 400 to 500 DOCSIS CM embedded devices over a RF bus topology. Fig. 2.25 illustrates the connection

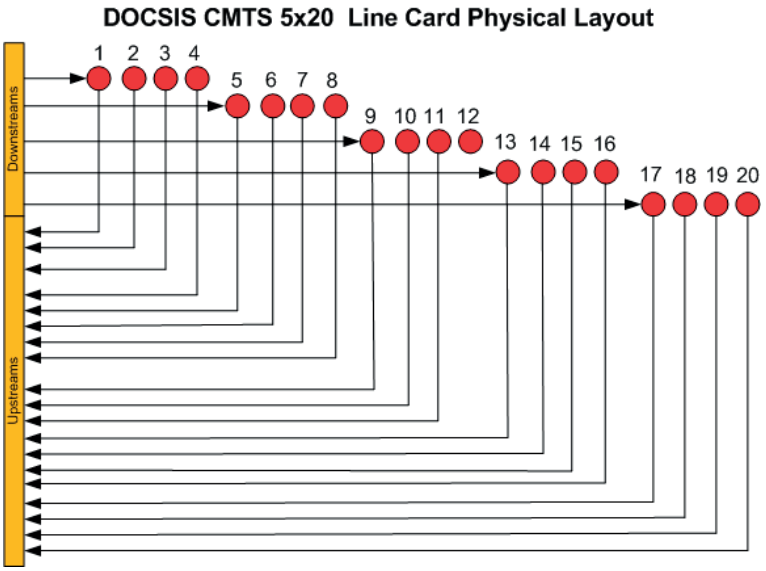


Figure 2.24: DOCSIS router line card organization.

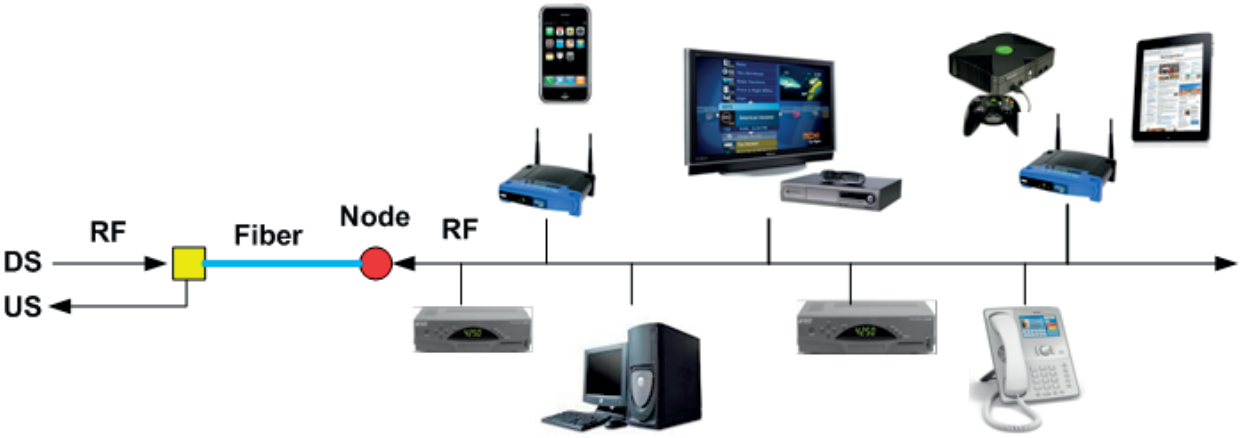


Figure 2.25: Hybrid fiber/RF node device

of multiple embedded device attachments to a DOCSIS based cable service provider system. As discussed in Section 2.4.1.1, DOCSIS CM devices may include any of set-top boxes, VoIP phones, and WiFi access points (for both residential and commercial applications) that attach personal computers, tablets, smart phones, game consoles, and connected *Smart-TV's*.

2.4.3 Broadband Cable Service Provider System Architecture

Fig. 2.26 illustrates the internal functional components of the remote-hub, broadband network, head-end, and typical regional data-center. A head-end can be partitioned into four functional sub-systems; 1) a digital content and RF distribution system including content protection services, 2) command and control system for embedded devices such as set-top boxes, 3) one or more compute and data-center server clusters, and 4) Internet and cloud system network peering infrastructures. The regional data-centers or head-end clouds are all inter-connected utilizing the same DWDM fiber network for connecting remote-hubs. Therefore each regional data-center or remote-hub may access content or data located anywhere on the system.

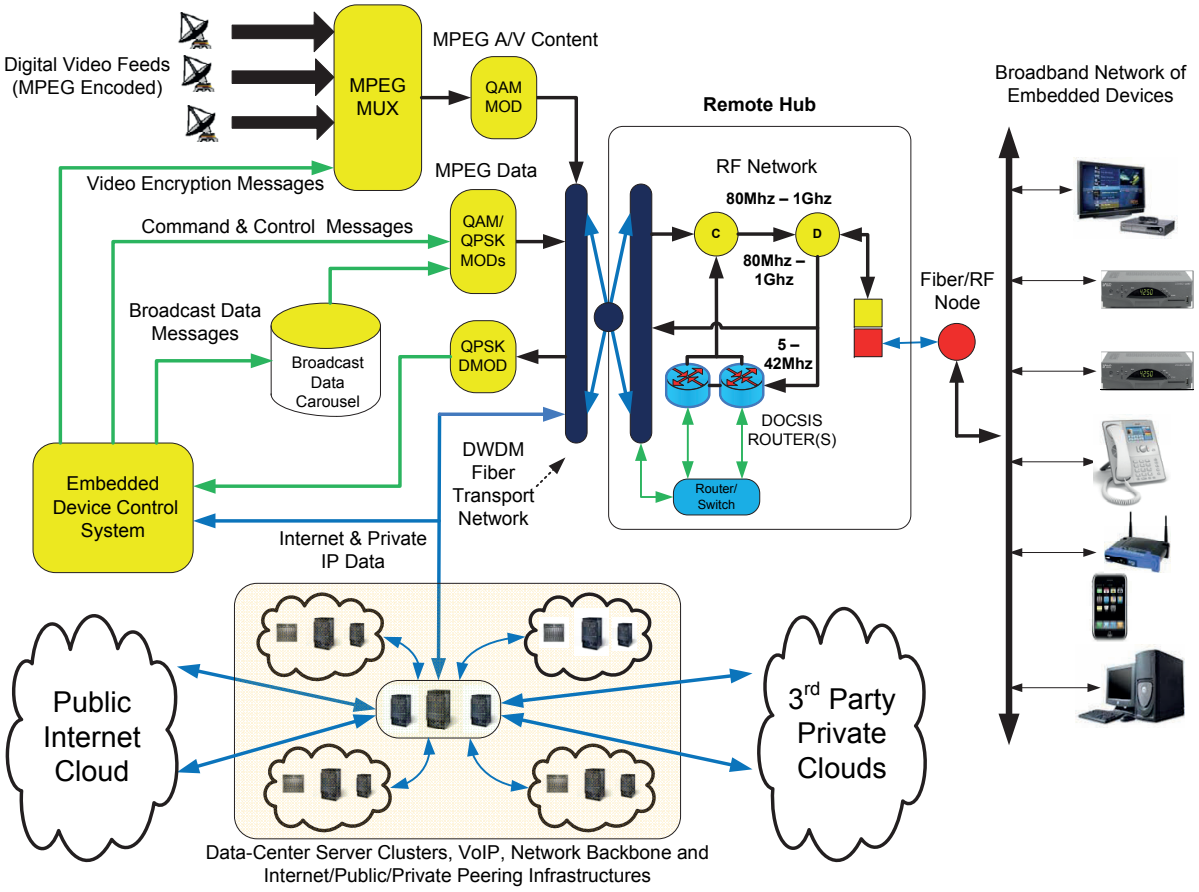


Figure 2.26: Broadband cable service provider system architecture.

Digital content and RF distribution system. The digital content and RF distribution system is illustrated in the upper left and center of Fig. 2.26. A modern broadband cable system is designed to carry audio/video content in the form of MPEG-2 or MPEG-4 signals. These MPEG signals, also referred to as digital audio/video *feeds*, are delivered either from local content sources (for example video footage captured on a digital video camera) or, more commonly, content providers that syndicate their content throughout countries typically over satellite delivery. In both cases, digital content feeds are multiplexed together along with encryption information using MPEG multiplexing hardware whose outputs are baseband frequencies that are upconverted to RF frequencies within a 6Mhz band using QAM (Quadrature Amplitude Modulation) modulator devices. The broadband RF signals are then distributed to remote-hubs using the DWDM fiber transport network as described in Section 2.4.2 and illustrated in the center of Fig. 2.26. The QAM devices at the head-end in conjunction with the remote-hub RF network and fiber/RF node comprise the RF distribution system. The remote-hub and fiber/RF node physical RF distribution system are interconnected via fiber up to 50km in distance. The fiber/RF node converts optical signals back to RF frequencies which are then amplified before driving the RF cable network where embedded devices attach, as shown on the right of Fig. 2.26. To support communication with legacy RF devices, which do not support QAM based signaling, a second RF modulation scheme referred to as QPSK (Quadrature Phase Shift Keying) is present as part of the RF distribution system. The QPSK modulator and demodulator elements in Fig. 2.26 are primarily utilized by the command and control system described next. The DOCSIS network is also considered part of the RF distribution system, and consists of two subsystems: 1) a two-way IP network interconnect between the DWDM switching network, data-center servers and cloud, within the regional head-ends, and 2) the DOCSIS network and router system that resides in each remote-hub.

Command and control systems. Embedded devices compatible with broadband cable networks are managed by a specialized embedded control system that orchestrates device registration, operating system boot and application processes lifecycle management for set-top boxes, cable modems and cable system WiFi access points. The embedded system control system shown in Fig. 2.26 sends and receives command and control messaging to/from embedded devices through the QAM and QPSK RF distribution system. The command and control system is also responsible for generating encryption messages which are delivered as part of the MPEG video and data

stream for securing premium content services to devices capable of receiving video streams. For distribution of data to all devices, a broadcast data carousel is managed by the command and control system. The broadcast data carousel provides a mechanism for cyclical data distribution (data represented as a filesystem is repeatedly broadcast) over both the QAM and QPSK RF networks at specified delivery frequencies that devices such as set-top boxes may listen to. In this manner, scalable data delivery can occur in a unidirectional manner to a large number of devices eliminating the need for two-way communication.

Compute and data-center server clusters. In addition to RF specific devices such as QAM and QPSK described earlier, DOCSIS network components, and embedded control systems, multiple data-center server complexes provide a wide range of network, content, and application layer services over TCP/UDP/IP protocols to the overall system. The bottom center of Fig. 2.26 illustrates multiple clusters of server clouds each interconnected through a central network backbone of core routers and switches. The server clusters consist of typical rack mounted Linux multi-core blade servers available from companies such as Cisco, IBM, Dell or HP. Each individual cluster performs a specific set of services to the embedded device network that may include network services such as DHCP or DNS, or more application-oriented services such as interactive user-interfaces for electronic programming guides (EPG) and video-on-demand (VOD). In recent years, newer embedded devices such as smart tablets and intelligent televisions have become prevalent in terms of their operation on cable broadband systems. Therefore, new compute intensive service clusters have been established to provide these devices with IP based services such as streaming video, search and recommendation-aware applications, to name a few. The multiple server clusters communicate with one another within a regional data-center over a centralized 10Gb/s Ethernet backbone that is switched, whereas bidirectional communication across data-centers (among different regional data-centers or head-ends) and the DOCSIS based embedded device network is through the DWDM transport network. This is indicated in Fig. 2.26 as the connection between the regional data-center DWDM switch and the core routing/switching network shown in the center of the server cluster clouds.

Internet and cloud system network peering. High-speed Internet access as well as third-party private cloud services to both internal server clusters and the millions of embedded devices is provided through direct peering connections from Internet backbone providers and respective

third-party cloud providers. Interconnection is based on multiple fiber connections that aggregate multiple 10Gb/s links over fiber, to each of the core routing/switching network infrastructures within each regional data-center. This is illustrated as the public and private network clouds interconnected to the core routing/switching network in the system architecture shown in Fig. 2.26. It should be noted that the integration to public and private network clouds is a key aspect of the ability of broadband service providers to support the production of new cloud services as well as their consumption.

2.5 Embedded Device Processors

Consumer electronic devices that attach to broadband cable networks are built utilizing increasingly powerful embedded processors from companies such as IBM, Cisco, ARM and Broadcom corporations. In this section I briefly describe a number of embedded processors that are now common in many DOCSIS cable modems, set-top boxes, game consoles, smart tablets and TV devices in-use within broadband subscriber homes. Section 2.6 illustrates the application of embedded processors to the implementation of various highly integrated consumer embedded devices.

IBM Cell Processor. The IBM Cell processor was developed by IBM, SONY and Toshiba for the Playstation 3 game-console and later also for high-performance applications including server products [81]. The Cell processor is a high-performance, multicore, 64-bit based architecture operating at clock rates exceeding 3GHz, consisting of the following primary subsystems: 1) a 64-bit SIMD PowerPC core (PPE) with L1 and L2 caches that acts as a management processor; 2) 8 SIMD compute cores referred to as Synergistic Processing Elements, or SPEs, each containing 256KB of local memory and high-speed DMA for data movement. The SPEs are essentially 128bit vector processors that can operate on multiple pieces of data at once; 3) the element Interconnect bus (EIB) to support communication between subsystems; 4) two Rambus XDR memory controller; and 5) two Rambus FlexI/O controllers providing dual input/output interface channels.

Cisco STB SoC. While the current trend in embedded devices is towards the use of high-performance, licensable multi-core processor macros, as opposed to proprietary custom processor technologies, many STB devices in use today make use of STB vendor designed *System-on-Chip (SoC)* technologies. For example, a typical Cisco STB SoC available during 2007 includes the

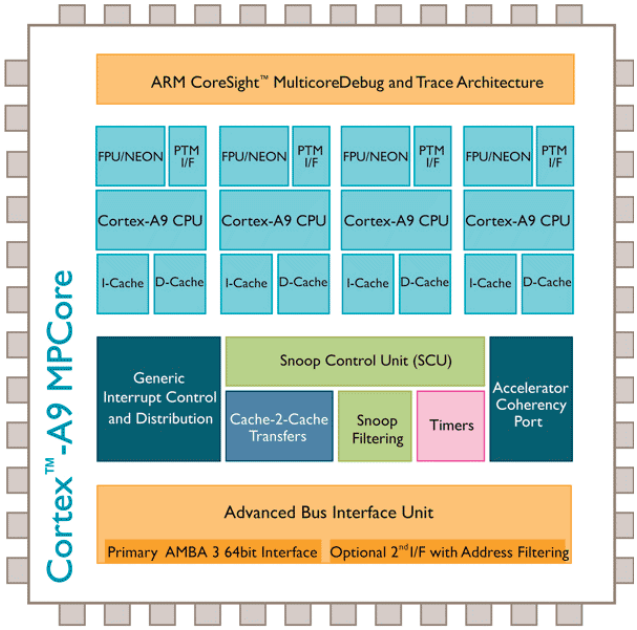


Figure 2.27: ARM Cortex-A9 MP processor. [Source: www.arm.com]

following functional components [59]: 1) a CPU core based on a 700MHz 32bit MIPS instruction set architecture, dual 16-bit DDR memory for 32-bit external memory interfacing, with hardware graphics support for 2-D drawing and blitter operations; 2) basic I/O operations such as USB and Infrared (IR) input; 3) support for wideband RF demodulation of both audio and video signals; and 4) decryption circuitry, as well as two-way DOCSIS (QAM) or DAVIC (QPSK) communication as described in Section 2.4.1.1. The remainder of the SoC functionality integrates the various video and audio signals from the MPEG and graphics accelerator hardware into output signals required for television receivers.

ARM Multicore Processor. Representative of a more recent embedded processor is the ARM Cortex-A9 MP 32-bit multicore processor. The ARM family of processors is found in numerous consumer electronic embedded devices, including smart tablets such as the Apple IPAD, mobile phones, mobile computing, as well as next generation set-top boxes and smart TV devices. The chip may operate from 800MHz to 2GHz, and is available in a number of configurations to optimize cost, performance and power [54]. The key features of the ARM Cortex-A9 MP are illustrated in Fig. 2.27 [54] and include the ability to support anywhere from 1 to 4 high-performance Cortex-A9 CPU cores, each with 32K L1 instruction and data caches, providing 2.5DMIPS/MHz/core

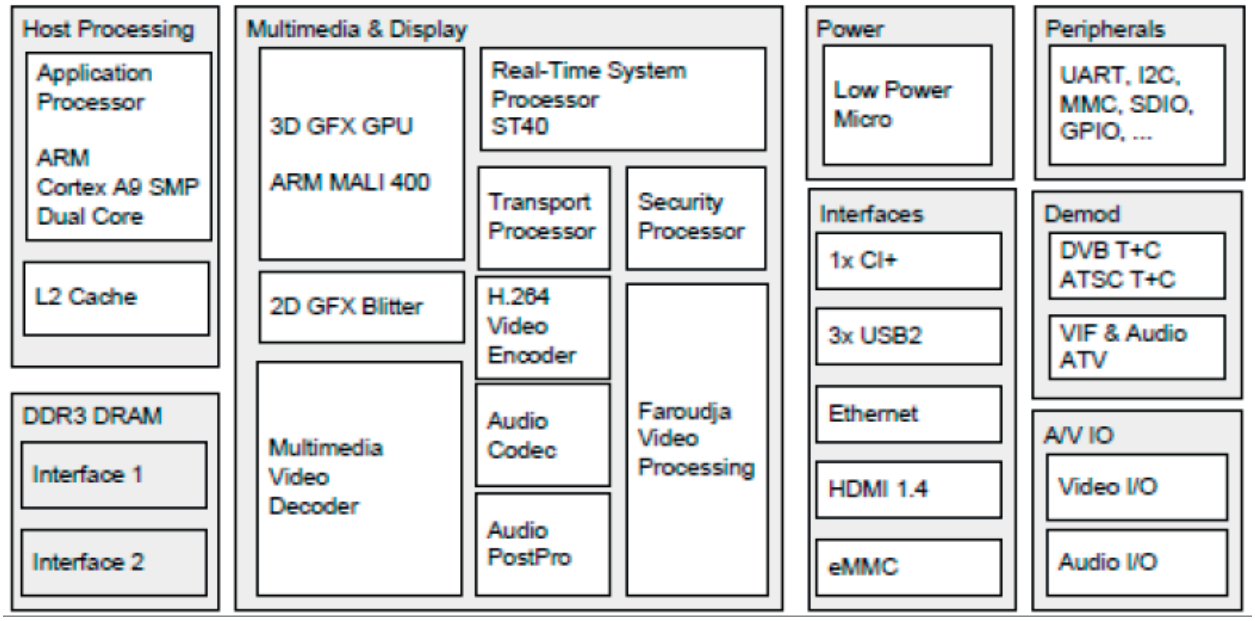


Figure 2.28: ST Smart-TV processor. [Source: www.st.com]

performance [54]. Optionally, FPU/NEON processing units may be added for high-performance single and double precision FPU operations, including a 128-bit SIMD instruction set addition by the NEON execution unit [54]. A snoop control unit provides cache coherency in a multicore processor environment. To further improve performance, an optional external L2 cache memory control unit enables integration with up to 8MB of L2 cache memory [54]. The ARM family of embedded processors can scale across a variety of consumer electronic markets or applications. The processors are generally licensed as synthesizable or hard-macro implementations [54].

ST Newman Smart TV SoC. Emerging digital televisions now contain powerful processors enabling interactive applications such as Netflix and Internet access. As a final example of the use of embedded processor technologies for consumer device applications, the block diagram of the STMicroelectronics Newman Smart TV FLI7680 SoC is shown in Fig. 2.28 [73]. The ST FLI7680 contains a dual-core ARM Cortex A9 processor with 1MB cache and 3D graphics processor based on the ARM MALI 400 GPU. The ST FLI7680 integrates dedicated audio and video decoders, cryptographic processor, video composition engine. Video and audio processing for exceptional front-end screen experience is provided by 14-bit color and contrast processing with hardware functions for video color sharpness and noise reduction [73]. Multiple output interfaces including

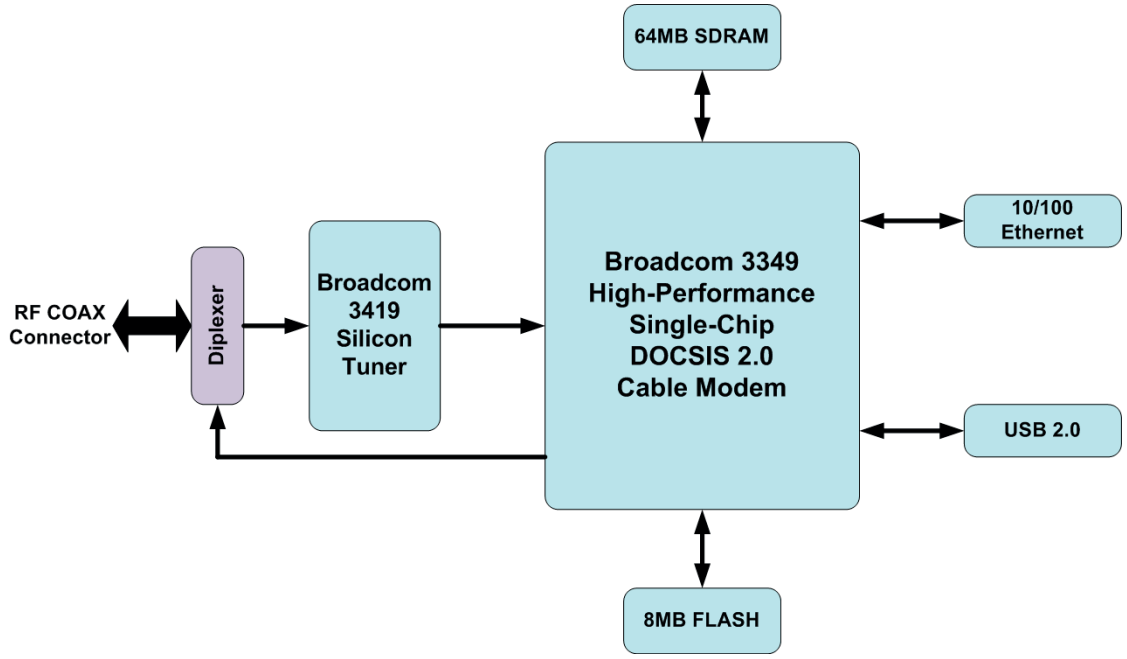


Figure 2.29: Generic DOCSIS cable modem block diagram.

HDMI and network access such as Ethernet are fully integrated.

2.6 Broadband Embedded Devices

Section 2.5 described a small cross-section of embedded processors found in consumer electronic devices. This section explores how these chips are used to implement embedded devices deployed on cable broadband networks and in particular, the evolution of broadband cable set-top box devices.

DOCSIS Cable Modem. The block diagram of a typical DOCSIS cablemodem is shown in Fig. 2.29 [74]. These embedded devices provide local IP network connectivity to a broadband network over DOCSIS protocols as described in Section 2.4.1.1. These devices are highly integrated, cost sensitive systems, found within millions of set-top boxes, home cablemodem routers, voice-over-IP (VoIP) phones, and WiFi access point devices. Referring to Fig. 2.29, the DOCSIS cablemodem is fully implemented using two Broadcom chips [55], relying only on external components for RF conversion, SDRAM Memory and Flash, as well as physical cable/LAN ports. Common to cablemodem devices is support for VoIP services. This requires the addition of DSP and voice CODEC processor chips as well as physical phone ports.

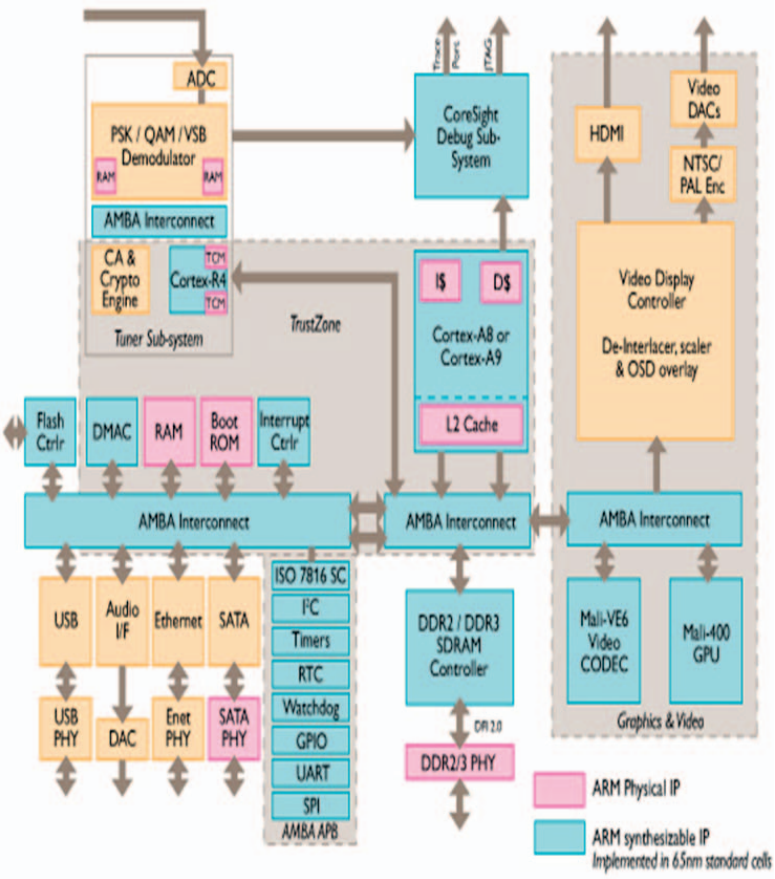


Figure 2.30: ARM based STB block diagram. [Source: www.arm.com]

Cisco STB. Similar to cablemodem devices, STB devices make use of highly integrated and often proprietary SoC technology. The Cisco SoC introduced in Section 2.5 is utilized to implement a complete Cisco model 4650 STB. The 4650 STB implementation features a single SoC that contains a MIPS based CPU core, Media processor, complete DOCSIS cablemodem, along with configurable static and flash memory modules. All remaining components include tuner and various analog discrete blocks necessary for interfacing with external I/O ports, network interfaces, connectors and indicators. The Cisco 4650 STB is representative of the technology available for STBs produced between 2005 and 2009.

ARM Based STB. The recent trend in STB embedded devices, as in most consumer electronic devices, is towards more powerful processors and memory configurations. Recently STB chip producer STMicroelectronics has licensed ARM technology to develop next generation STB devices [34]. Similar to the Cisco STB, Fig. 2.30 illustrates the block diagram of a highly inte-

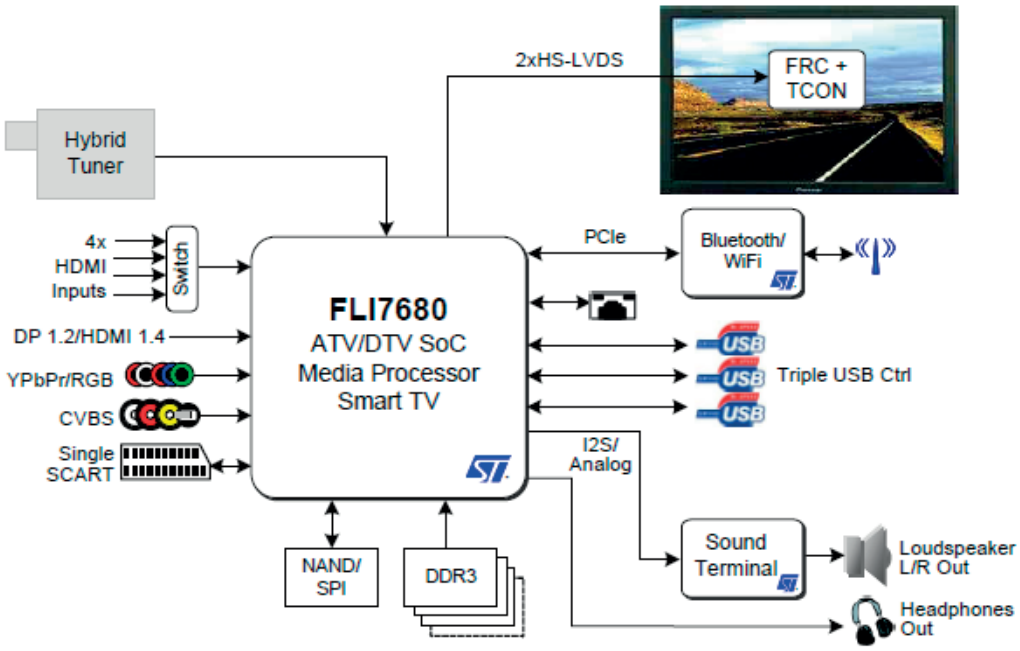


Figure 2.31: ST based Smart-TV block diagram. [Source: www.st.com]

grated, high-performance multi-core ARM based STB. In this implementation, the CPU is based on the Cortex-A9 processor, with an additional MALI-400 GPU. Memory and I/O functions are fully integrated and based on ARM soft or hard macro cell libraries. Not shown in Fig. 2.30 are additional components such as memory, flash, and external physical interface ports.

Next Generation Smart-TV. The block diagram of a current generation Smart-TV is shown in Fig. 2.31. Based on the ST FLI7680 Newman chip described previously, a high performance, multicore Cortex-A9 system implements an intelligent TV capable of 3D graphics applications and Internet access with nothing more than external memory. An optional WiFi-support module and physical interfaces are the minimum required components for a complete Smart-TV system. The Smart-TV system block diagram example demonstrates the degree of system integration and high-performance processor technologies available in modern consumer-electronic embedded devices currently operating on broadband networks.

2.6.1 Evolution of Broadband Embedded Devices

Embedded device technologies continue to advance in performance and density as evident in the examples discussed in Sections 2.5 and 2.6. Over the past decade a number of trends have emerged

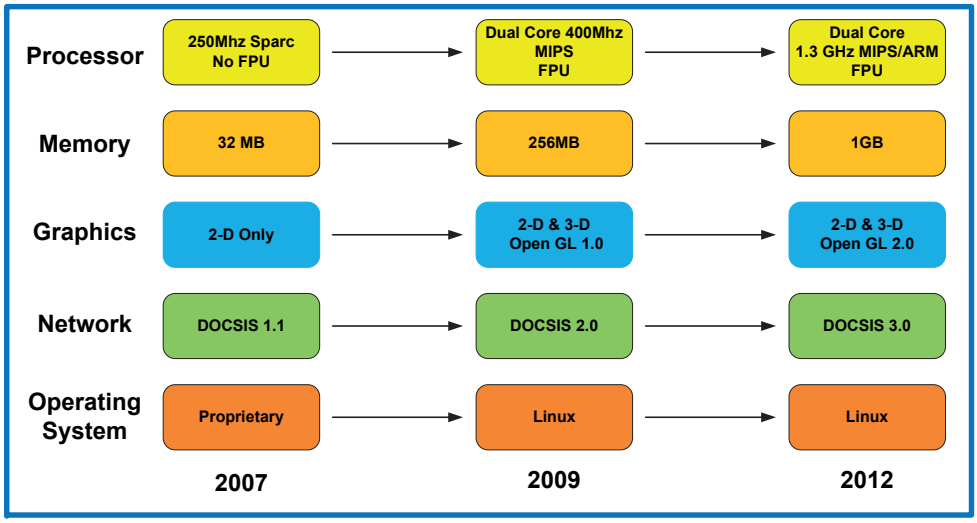


Figure 2.32: Evolution of STB devices between 2007 and 2012.

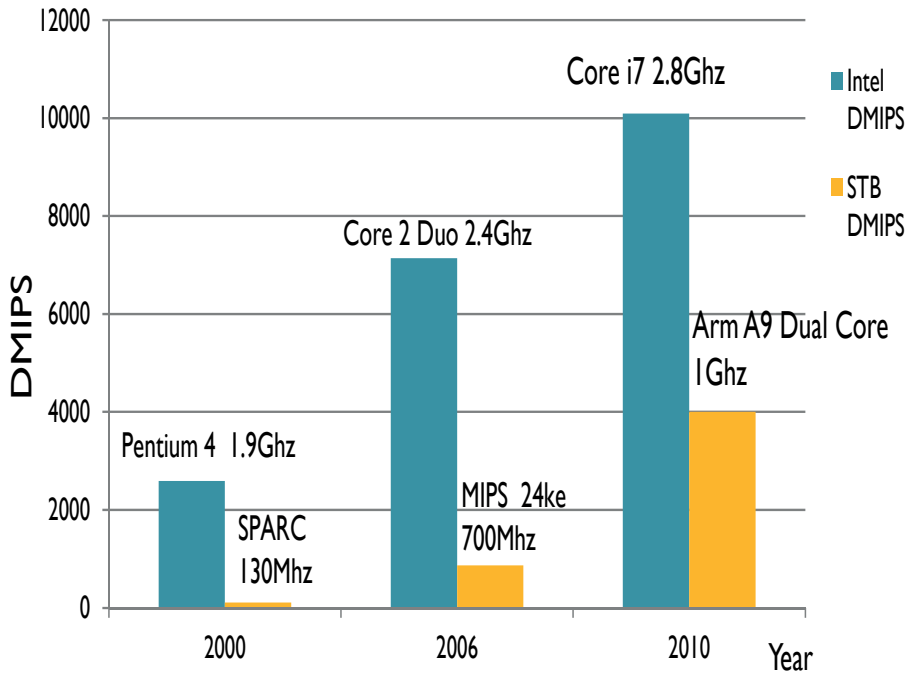


Figure 2.33: Performance gap decrease between desktop PC and embedded processors.

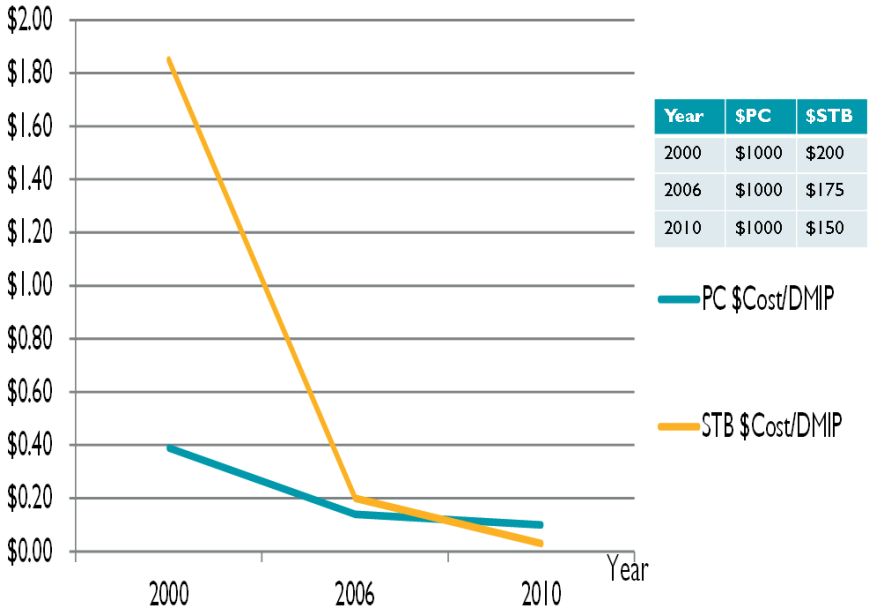


Figure 2.34: Desktop PC versus STB cost/DMIP.

that are influencing the evolution in functionality, performance and cost of consumer-electronic devices relative to larger classes of computing systems such as high-performance computing systems (HPC). The impact to broadband STBs devices and embedded systems in general has been favorable and is illustrated in Fig. 2.32, Fig. 2.33 and Fig. 2.34. In terms of the evolution of STB functionality, Fig. 2.32 illustrates the five year trend in STB capabilities between 2007 and 2012. In the span of less than five years, a STB has evolved from hosting a single 300Mhz processor without floating-point unit, one 2-D graphics unit and just 32MB of memory to a modern multi-core system-on-chip (SoC) with two 1.3-Ghz processing cores, multiple video and 3D graphics accelerators, and 1GB of memory. Fig. 2.32 illustrates the STB historic trend towards increased performance, across all functional subsystems including CPU, memory, graphics, and networking. During this same time period the STB operating system has also transitioned from proprietary to an open Linux based software system. Fig. 2.33 and Fig. 2.34 illustrate a relative decrease in computational performance gap and cost/performance between PC/blade servers and STB devices. This trend is primarily driven by the convergence to multi-core processors and an upper bound on the maximum processor frequencies achievable. The convergence is further driven by the fact that similar processor cores are now utilized in the design of both embedded devices and high-performance computational systems

based on commodity blade based servers. Due to limits in the continued growth in processor clock frequency, the relative gap between high-performance systems and embedded device processing capabilities continues to narrow. In fact, the shrinking performancer gap is currently having a relatively different impact on server systems versus STB electronics because the computational power of STBs, as are other embedded devices such as cell-phones, tablets, and Smart-TVs, is increasing at a faster rate than the computational power of server class systems. For example Fig. 2.34 illustrates typical non-DVR STB costs dropping from $175 to between $150 and $100 dollars, with performance increasing 4 times over a 2 year period. Note that the clock rate of the STB processor was 15 times slower in the year 2000; however, in the year 2010 the difference has dropped to within 3 times in the last 2 years.

2.7 Summary

This chapter provided a high-level overview of multiple technologies including distributed and Cloud computing frameworks as well as broadband technologies that comprise typical broadband cable service provider systems. Managed Broadband DOCSIS networks provide necessary physical layer and protocol infrastructure to enable communication services for QoS supported, data delivery over broadcast and unicast methods using TCP/IP protocols. Section 2.4.2 presented a cable service provider system, illustrating a system architecture model that scales to support the operation of millions of embedded devices, which attach to the broadband network over DOCSIS protocols presented in Section 2.4.1.1. The embedded devices themselves fall into a number of categories depending on their functionality. As illustrated in Section 2.5 and Section 2.6, however, the embedded processor and device technologies are closing the performance gap with high-performance computing clusters and data-center class computing systems. By integrating managed broadband service provider systems, networks and embedded devices, distributed computation across broadband embedded devices enables a new class of heterogeneous computing infrastructure as well as new opportunities for new Cloud services.

Chapter 3

Broadband Embedded Computing Evaluation And Feasibility

3.1 Introduction

In this chapter an empirical study is conducted to validate and demonstrate the feasibility of broadband embedded computing. As introduced in Chapter 2, a model of distributed parallel computation utilizing broadband embedded devices requires a distributed broadband embedded device infrastructure that can provide both computational resources and network bandwidth sufficient for distributed computation in addition to delivery of traditional video, data and other interactive services. I evaluate the feasibility for utilization of STB devices for broadband embedded computing by measuring their operating characteristics during typical usage under real world conditions. All experimental testing results are obtained through statistical data gathered from a realistic broadband embedded system environment over a one month period. The results show that devices such as set-top boxes are good candidates for emerging broadband embedded computation due to their operating characteristics and usage patterns.

3.2 Experimental Methodology

An experimental framework for collecting and analyzing empirical data from Cisco 4200 STB devices and DOCSIS 1.1 network traffic was designed and implemented. The Cisco set-top boxes are

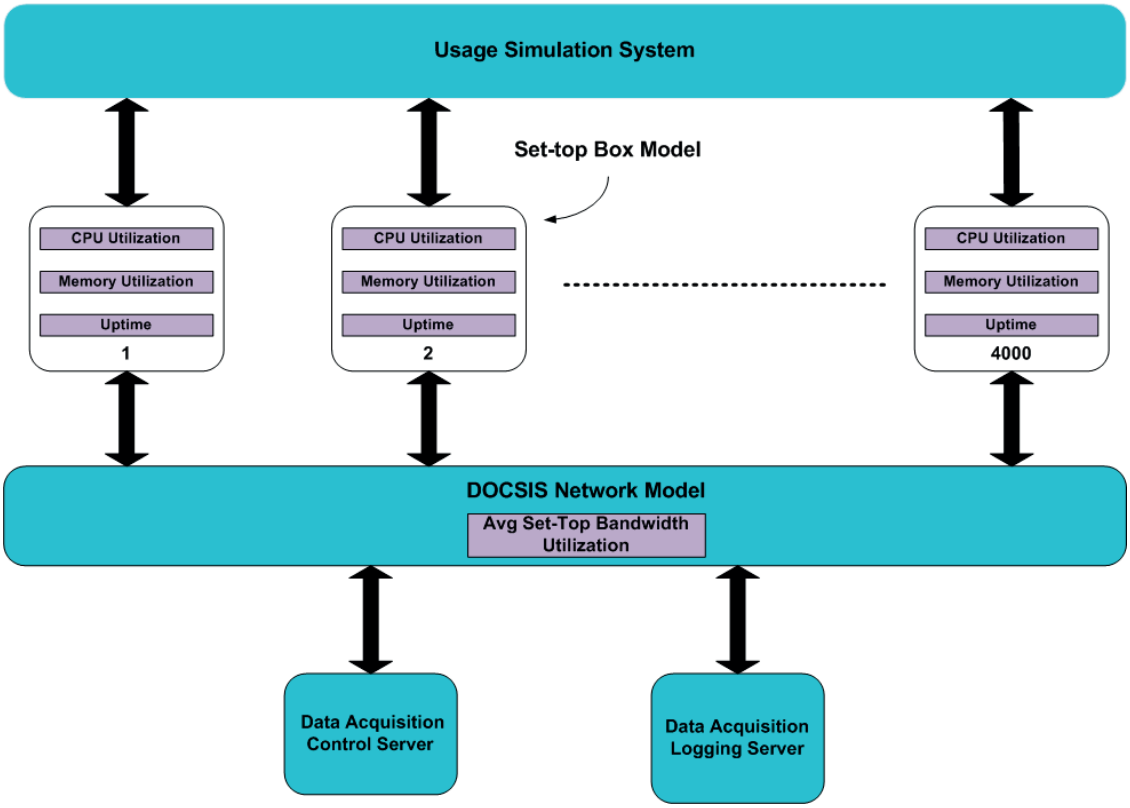


Figure 3.1: Experimental system model.

those available in 2004 and contain 32MB of memory and a 170MHz CPU. Fig. 3.1 illustrates a high-level representation of the measurement system and the data gathered from the STB devices and DOCSIS network. The STB device behavior is modeled in terms of CPU utilization, free memory, and uptime using available STB operating system parameters. Device model data is collected from approximately 4000 devices, selected randomly, for a period of one month over varying load conditions such as video streaming, interactive usage, or idle time, typical of usage patterns. Concurrently, measurements of the network utilization as a function of different services over time is used to evaluate the STB bandwidth requirements in the presence of other data services. Network utilization is measured by collecting transmit and receive byte counters at the broadband router for each DOCSIS 1.1 device grouped by service. It should be noted that DOCSIS 1.1 network bandwidth is effectively scheduled on a best-effort basis and can be further managed utilizing QoS profiles as described in Section 2.4.1.3. As a managed service, network performance requirements would typically be defined as part of system design requirements.

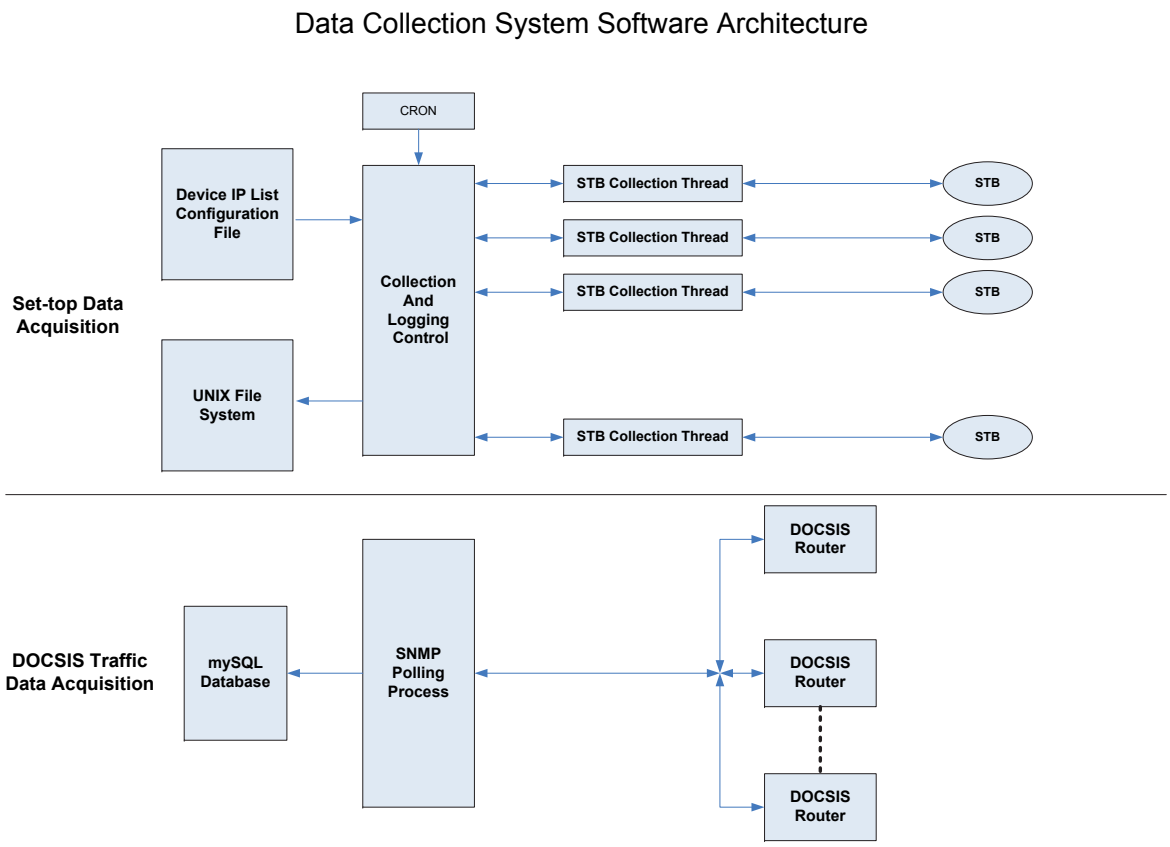


Figure 3.2: Block diagram illustrating experimental system architecture.

3.3 Experimental System Design and Implementation

Referring to Fig. 3.2 the test system is comprised of 4000 STB devices and a centralized STB statistic collection and logging system developed for the data-acquisition experiments. The collection and logging system is made up of two Sun 480 UNIX servers that hosts the data acquisition software written in Java. The collection software is launched from a CRON job which runs every 15 minutes and polls the complete population of STB devices, retrieving the STB device model parameters shown in Fig. 3.1. The collection software is multithreaded with a dispatch control module launching a new thread for each set-top polling process. This is done to ensure that the collection system would be non-blocking and poll all STB devices, as there is no assurance that any single device will respond because it could be turned off or disconnected. Network statistics collection is implemented

using standard SNMP polling software adjusted to retrieve byte counter set-top box Management information base (MIB) variables for each device. All transmit and receive byte counters are broken down by service and QoS profile. Data is stored using a combination of mySQL and the UNIX file system. Since a large amount of data is generated, a set of tools for processing and visualizing the data is necessary. A Java post processor program was used to normalize and summarize the data for this purpose. The program reads in record data produced by the collection system and produces output files that can be visualized using a graphics or plotting package. Origin lab graphing software was used to generate all the final surface plots described in the next section.

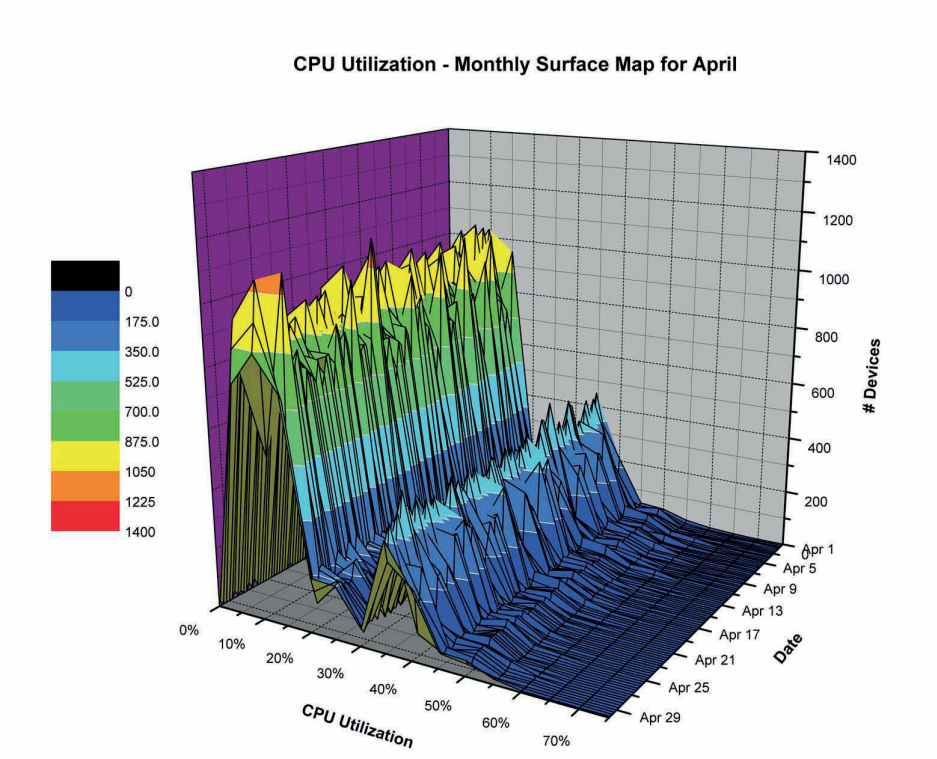


Figure 3.3: CPU utilization.

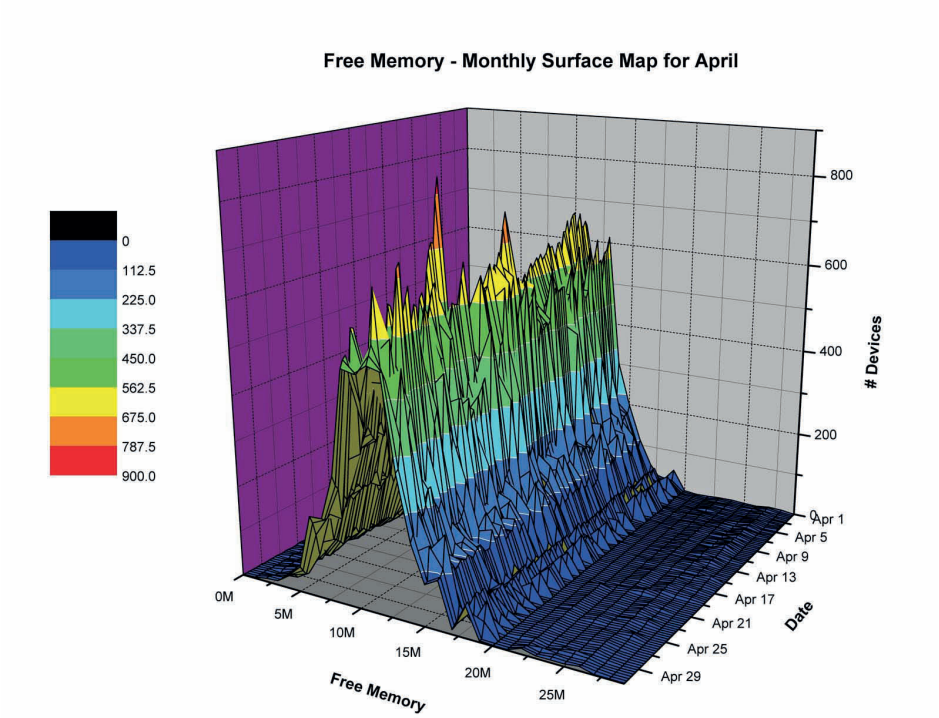


Figure 3.4: Memory utilization.

3.4 Experimental Results

3.4.1 Set-top Device Characteristics

Experimental data was collected for STB CPU utilization, free memory, and device uptime, over a one month acquisition period. Data was processed to generate a series of plots. Figures 3.3, 3.4 and 3.5 shows the resulting surface plots for CPU, free memory, and uptime respectively. Each plot represents frequency distributions of device count on the Y axis versus the X axis model attribute, over time in the Z axis.

Fig. 3.3 CPU utilization is below 50% with the majority of devices below 20%. Such low CPU utilization suggests that the application processor is underutilized with the MPEG processor performing most of the system work. The calculated mean is 19% with a standard deviation of 12%, most likely a result of the second peak that occurs between the 30% and 45% CPU utilization levels. Fig. 3.4 also shows that the amount of available free memory is typically between 5MB and 15MB

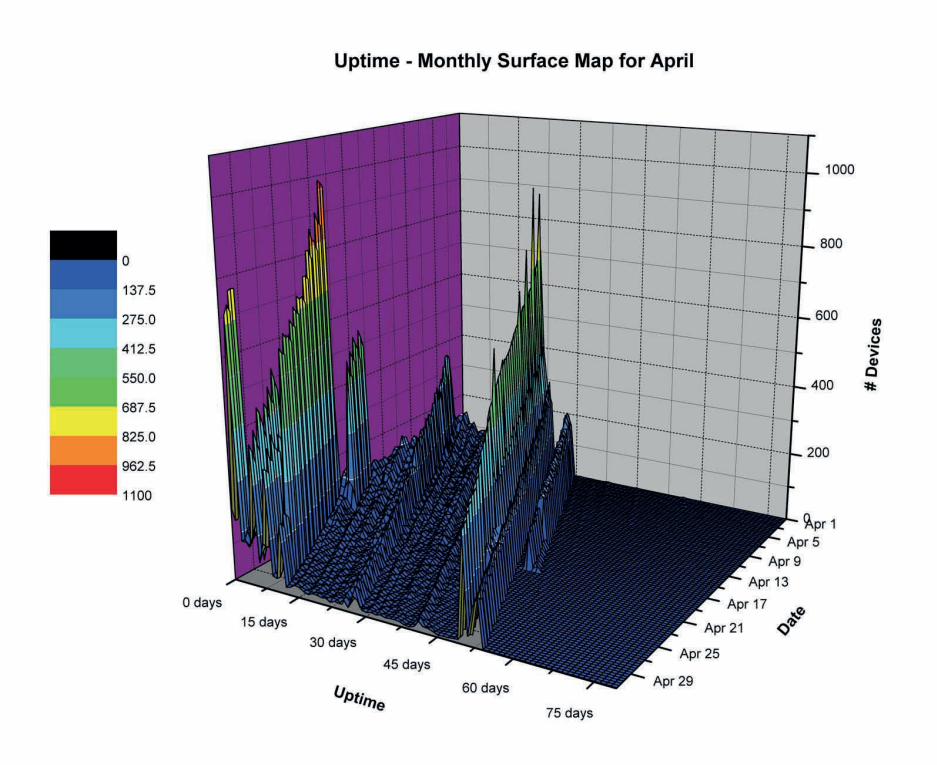
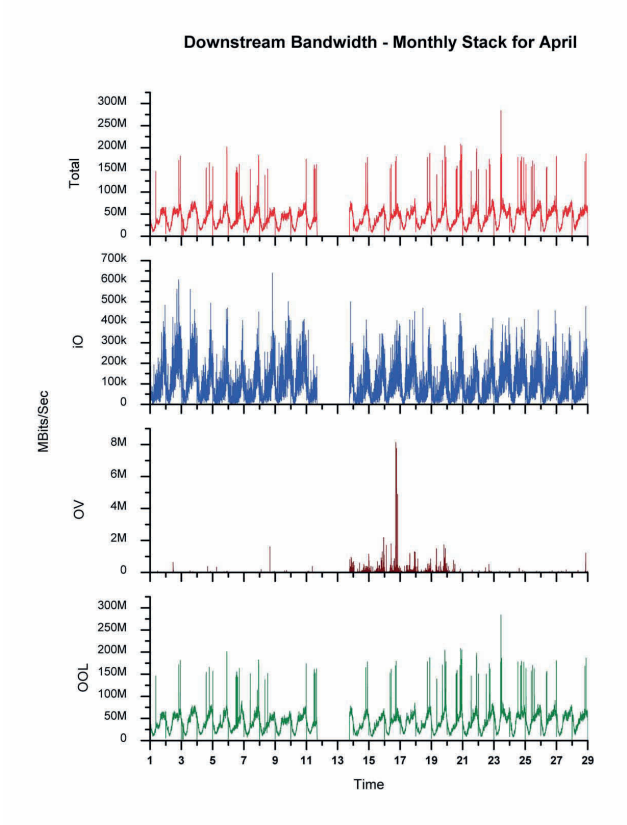


Figure 3.5: Device uptime.

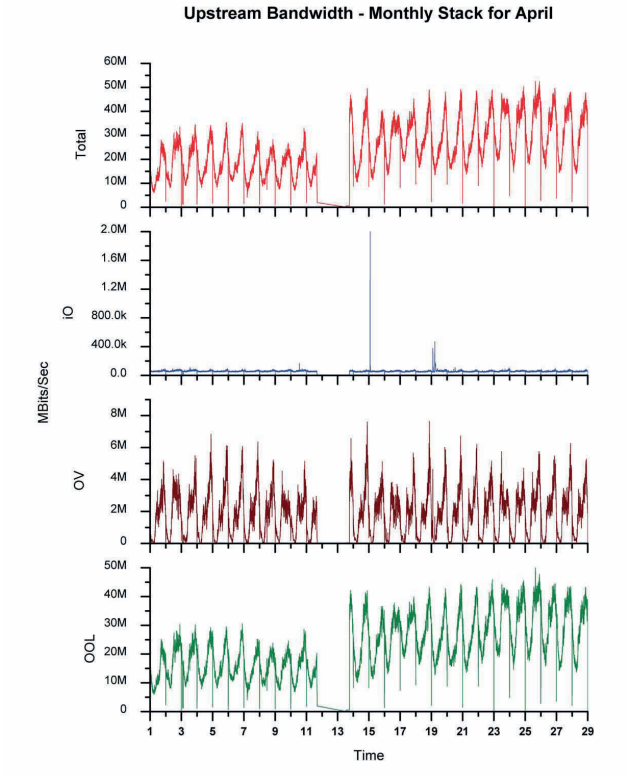
with a calculated mean of 12MB and standard deviation of 3MB. STB devices available in 2004 were very memory constrained but consistent with their initial target functionality. It is interesting to note that CPU and free memory exhibit minimal variations over time. This behavior supports the argument that the application processor is utilized only during short interactive sessions, or other command and signaling operations while the MPEG processor handles most of the system overhead. The STB uptime behavior shown in Fig. 3.5 indicates that most devices are available typically around 15 days but as high as 60 days. The calculated mean is 16 days with a standard deviation of 18 days. The variation in uptime statistics is due to different STB devices having been rebooted at different times during the acquisition window.

3.4.2 Broadband Network Characteristics

All set-top devices share a common TCP/IP DOCSIS network with cable modems and VoIP services. In this study measured router statistics for each of the DOCSIS traffic flows is obtained to



(a) Downstream utilization



(b) Upstream utilization

gain insight into current STB network utilization in the presence of other services. The STB devices in this study are connected to a router that can support a total of 304MBits/sec of downstream traffic and 200MBits/sec of upstream traffic. Note that router capacity is a function of the number of line cards and how the RF network is connected to the router. Generally, as more capacity is required, additional router cards and/or partitioning is also required. Hence, specific device or service bandwidth is usually a defined parameter during the design process.

Fig. 3.6 illustrates the downstream and upstream network utilization grouped by service. Calculated statistics are as follows: The monthly set-top network bandwidth has a downstream mean of 101Kbits/sec, standard deviation of 82Kbits/sec, and peak utilization of 638Kbits/sec. The upstream mean is lower at 56Kbits/sec, standard deviation of 28Kbits/sec, and typical peak of 85Kbits/sec. There was however a onetime peak of 2Mbits/sec during a master reboot that forced all devices to generate numerous concurrent requests. This can be seen as a spike in Fig. 3.6(b). One other anomaly worth noting is the loss of activity shown on Fig. 3.6(a). This is simply a two-day loss of data collection during a system outage. In the presence of all services, STB network utilization remains low at less than 1% of total router bandwidth. In contrast, the cable modem service utilizes over 94% of router downstream capacity and 91% upstream. This is consistent with the dominate usage of cable modem services. Utilizing a STB device in the context of a distributed concurrent system would require additional capacity planning. This could potentially result in additional router scaling or network partitioning as a function of overall system performance requirements.

3.5 Related Works

Related research in evaluating service provider managed, embedded devices such as set-top boxes and broadband networks for their feasibility for non-traditional utilization has focused on overall architectures and their energy efficiencies for large-scale computation. Recent work by Batista, *et al.* [14] describes an architecture for harnessing under-utilized STB processor cycles for Internet scale high-performance computation and volunteer computing initiatives such as SETI@Home [53]. However in this work, limited testing in terms of actual embedded device suitability is described, because there is an underlying assumption that STB devices may voluntarily participate in the

proposed architecture. From the perspective of total power consumption and energy efficiency, Furlinger, *et al.* [38] analyze the current state of distributed computing on mobile and consumer electronic devices reporting performance results against a 4-node Apple TV cluster. In contrast, the contributions presented in this chapter emphasize the acquisition and measurement of device and system parameters data that can directly support the propositions (that under-utilized processor cycles are available for computation). The study in this chapter is similar to the works of Dischinger, et al. [30] and Maier, *et al.* [88] in broadband network characterization, with the major difference in measurement techniques. Whereas in this study the point of network measurement is through statistics taken from broadband routing infrastructure, the other research works performed their measurements from the perspective of the edge client devices.

3.6 Summary

The most common cable service provider managed embedded device, the set-top box, offers good potential for computation beyond its primary functions. Results from Section 3.4 indicate that STB processor and memory are largely under-utilized, and there is adequate bandwidth available on typical broadband networks for heterogeneous computing. However, the devices utilized for these experiments are based on 10 year old processor technology, and are, therefore, low in performance and capacity compared to typical blade servers found in clusters, or even mobile devices in-use today. Trends in next generation STB device SoC technology shall continue to narrow the gap in raw performance, as embedded devices including set-top boxes, connected-TV's, home-gateways, and tablets make use of similar high-performance multi-core processors found in blade and cluster server systems. Given these trends, and the experimental data illustrating the usage profiles of managed broadband devices and systems, a high-performance heterogeneous computational system based on the concept of broadband embedded computing can be realized by aggregating a large network of embedded devices, designing the network for QoS correctness, and leveraging programming models taken from distributed and Cloud computing environments. Chapter 4 describes the implementation of a first-generation system to support this thesis: it demonstrates a heterogeneous broadband embedded computing system utilizing a mixture of Sun Solaris blade servers, embedded set-top devices and a subset of the MPI library for message passing software to solve a parallel

Bioinformatic application problem.

Chapter 4

First Generation System For Broadband Embedded Computing Utilizing Open MPI

4.1 Introduction

In Chapter 3 the set-top box a managed broadband embedded device, was shown to exhibit functional charecteristics and utility in enabling its use within heterogeneous distributed computing architectures. There exists then the opportunity to leverage the multitude of large scale Multiple Service Operator (MSO) infrastructures to build a variety of distributed broadband embedded computing platforms based on the distributed computing technologies described in Chapter 2 such as MPI and MapReduce (See Sections 2.2.1 and 2.2.2). The composition of traditional cluster computing with broadband embedded computing supports a mix of application workloads spanning from bioinformatics, high-performance computing (HPC), social networking and even large-scale distributed processing required for Big-Data problems in data mining and analytics. In this chapter, I present a contribution that encompass heterogeneous computing, in the form of composing traditional data-center computing clusters, with distributed embedded systems, in the form of a first-generation system for broadband embedded computing and in conjunction with the Open MPI message passing environment for distributed computing. An important aspect of this work

is the implementation of a resource-constrained MPI library featuring a subset of point-to-point communications primitives for embedded devices and a runtime environment that enables basic interoperability between heterogeneous systems in the proposed system architecture.

As the performance of MSO-managed networks continues to grow and STBs start featuring sophisticated multicore processors [34], I propose a heterogeneous system architecture that combines a traditional Unix-based computer cluster with a broadband network of STB devices. To further experimentally validate the potential of this idea, a complete prototype system is implemented which represents a scaled-down version of the proposed architecture but can fully support a representative MSO streaming-video service (Section 4.2).

To validate the feasibility of implementing a broadband embedded computing system, the distributed message passing computing programming model based on the MPI software is implemented. The MPI implementation developed is an inter-operable subset of Open MPI which can run on the STB real-time operating system and, therefore, can act as the middleware for the heterogeneous system. The implementation and system integration is described in Sections 4.3 and 4.4. The first-generation broadband embedded computational system is tested by porting an MPI-implementation of ClustalW, a computationally-intensive bioinformatics application that is executed on both the computer cluster and the network of STBs (Section 4.5). ClustalW solves the *Multiple Sequence Alignment* problem, which is one of the most important problems in Bioinformatics.

Experimental results show that it is possible to execute ClustalW efficiently on our system while the STBs continue simultaneously to operate their primary functions, i.e. decode MPEG streams for monitor display and run an interactive user interface, without any perceived degradation in interactive performance comparing user-interface loading time or evidence of visual artifacts such as macro-blocking (Section 4.6). Indeed, I observe that content services on the STBs are unaffected by the presence of parallel computation due to the separation of content and application processing functions within their micro-architectures. Major challenges, however, need to be addressed in order to provide a highly-scalable heterogeneous runtime system and an efficient messaging-passing environment for distributed computation with millions of embedded devices.

In the remaining sections, I discuss: feasibility, implementation challenges, and opportunities in utilizing embedded STB devices for broadband embedded computing, heterogeneous parallel

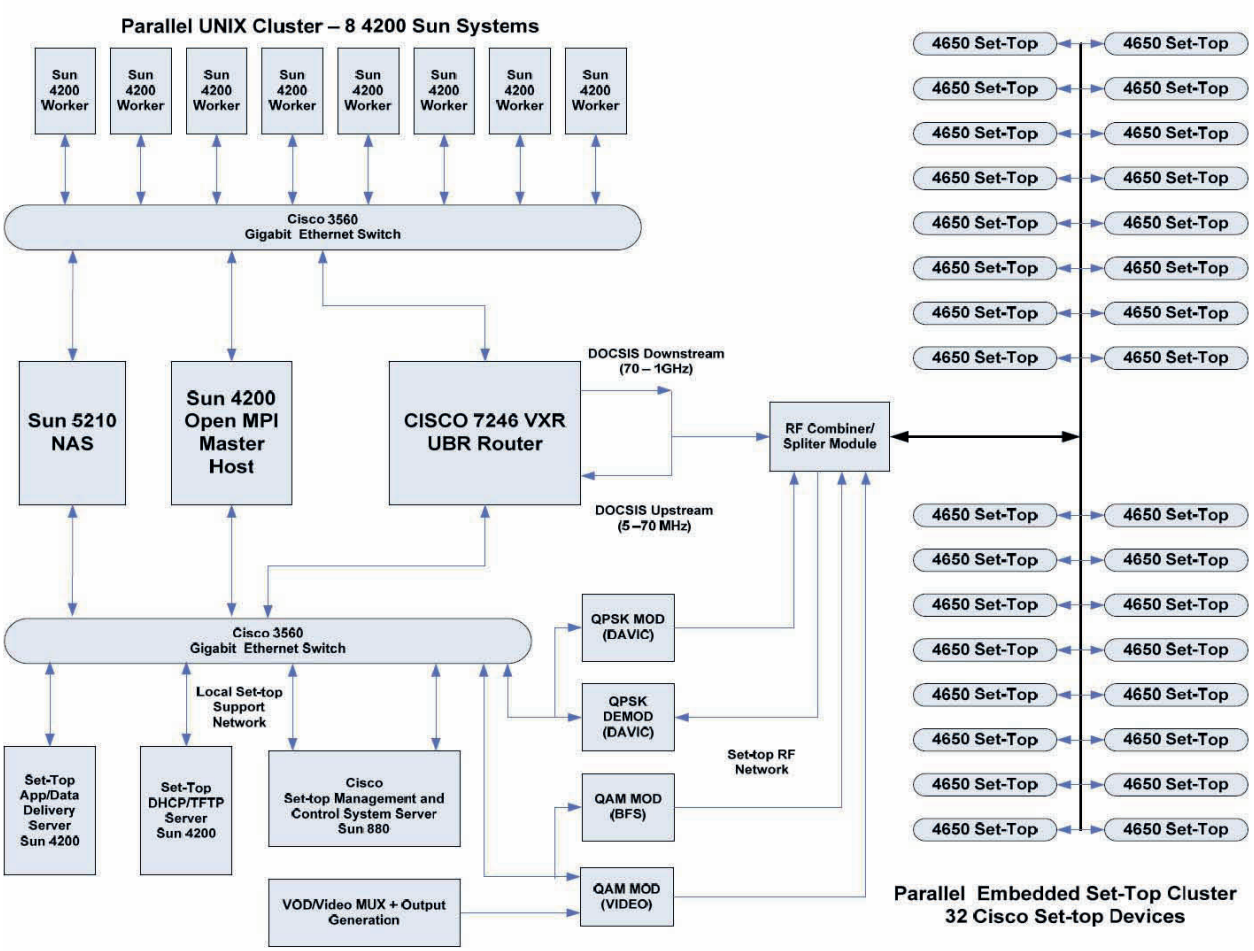


Figure 4.1: Block diagram of the complete prototype of the proposed heterogeneous system for broadband embedded computing and parallel application execution.

computing, and distributed processing. The development and experimental results from testing this system further supports the thesis claim that broadband embedded computing is a viable computational system architecture for heterogeneous Cloud based computing systems.

4.2 System Architecture

Fig. 4.1 gives a detailed view of the prototype system designed and implemented as a scaled-down, but complete and representative, version of the broadband embedded computational system for execution of heterogeneous parallel applications that I envisioned. This is a distributed-memory,

message-passing parallel computer that combines two functionally-independent clusters: a traditional Unix-based computing cluster with eight 4200 Sun machines (the *Linux Cluster*) and an embedded set-top cluster with 32 Cisco 4650 devices (the *Set-Top Cluster*). The two clusters share a common control server: the *MPI Master Host*. This server consists of a Sun 4200 processor that is connected to the Linux Cluster and Set-Top Cluster through a pair of Cisco Gigabit Ethernet network switches. The MPI Master Host initiates and typically coordinates all MPI processes across the clusters. Parallel applications based on Open MPI can execute on either cluster independently or on the whole system as a single large heterogeneous cluster.

The backbone network of the Linux Cluster is built using Gigabit Ethernet. Instead, the Set-Top Cluster requires a broadband router for converting between the DOCSIS network [57] and the Gigabit Ethernet backbone. The DOCSIS standard broadband-network technology for TCP/IP over Radio Frequency (RF) cable is described in Section 2.4.1.1. Notice that each broadband router can support over 10,000 STBs, thus providing large scale fan-out from the Linux Cluster to the Set-Top Cluster.

All 4200 Sun machines of the Linux Cluster are configured with two 2.8 GHz AMD Opteron dual-core processors, 16GB of system memory and run the Solaris 10 operating system. A Sun 5210 network attached storage array (NAS) provides 750GB of disk space using Sun NFS. The NAS system is also dual-connected using Gigabit Ethernet in the same manner as the MPI Master Host. This allows for a common directory structure and file access model for all processing nodes including the STB devices through a data access proxy system. This is particularly important for the execution of parallel MPI applications because each Open MPI host requires access to a common file system repository.

The Set-Top Cluster consists of 32 Cisco 4650 STBs that are connected using a RF network for data delivery using MPEG and DOCSIS transport mechanisms. The Cisco 4650 is a modern STB that contains a 700MHz MIPS processor, a dedicated video and graphics processor, 128MB of expandable system memory and many network transport interfaces including DOCSIS 2.0, MPEG-2, and DAVIC [62]. Indeed an important architectural feature of modern STBs is the multiple-processor design which allows the MIPS processor, graphics and video processors, as well as network processors to operate in parallel over independent buses. For instance, this makes it possible for user-interface applications to execute efficiently in parallel with any real-time video processing.

The DOCSIS 2.0 TCP/IP and MPEG-2 transport stream interfaces are based on quadrature amplitude modulation (QAM) protocols for transmitting and receiving signals on North American digital cable systems. DAVIC is a legacy 1.54Mbps interface predating DOCSIS and is used only for start-up signaling during STB initialization. The DOCSIS 2.0 standard provides for an inter-operable RF modem, based on TDMA protocols organized in a star topology connecting the central router and the STB DOCSIS interface. Devices on DOCSIS share access to the network, as arbitrated by the central router, and operate effectively at up to 38Mbps in the downstream direction (towards the STB) and 30Mbps in the upstream direction (towards the cluster). The MPEG-2 interface is primarily used for decoding video programs, but can also receive data delivered via the Broadcast File System (BFS) service on a dedicated QAM frequency. In this prototype system the BFS data delivery mechanism is used to deliver the embedded runtime environment to the STBs. A number of additional devices are required for STB management, application and data delivery, as well as video-content distribution. Indeed, the Set-Top Cluster is part of a scaled-down version of a complete cable system that consists of two additional subsystems: (1) a subsystem that provides STB management and control and (2) a collection of specialized devices for routing, transport, and distribution that support MPEG video and data delivery and transfer of TCP/IP data between the Gigabit Ethernet network and the RF set-top DOCSIS network.

As shown in Fig. 4.1, the management and control subsystem consists of a Sun 880 server, which is responsible for the initialization, configuration, and delivery of broadcast applications and data using its BFS carousel. Broadcast applications are STB executables that are simultaneously available to all STB devices connected to the dedicated broadband network. A STB device tunes to a specific channel frequency and receives the broadcast application or data of interest using a proprietary Cisco communications protocol. The BFS data is sent from the central server, or head-end, at regular cyclical intervals—hence the term carousel—over MPEG-2 directly into a QAM device where it is modulated onto the RF cable plant at a specified frequency for STB reception. For non-broadcast applications, a Sun 4200 is used as an Apache HTTP server that delivers application executables and data in parallel to all requesting STBs through the DOCSIS TCP/IP broadband network. Basic TCP/IP network services are provided by a Sun 4200 running DHCP and TFTP. DHCP is used to assign an Internet address to each STB. TFTP is the primary method for distributing configuration information. A video content channel using a single video source and

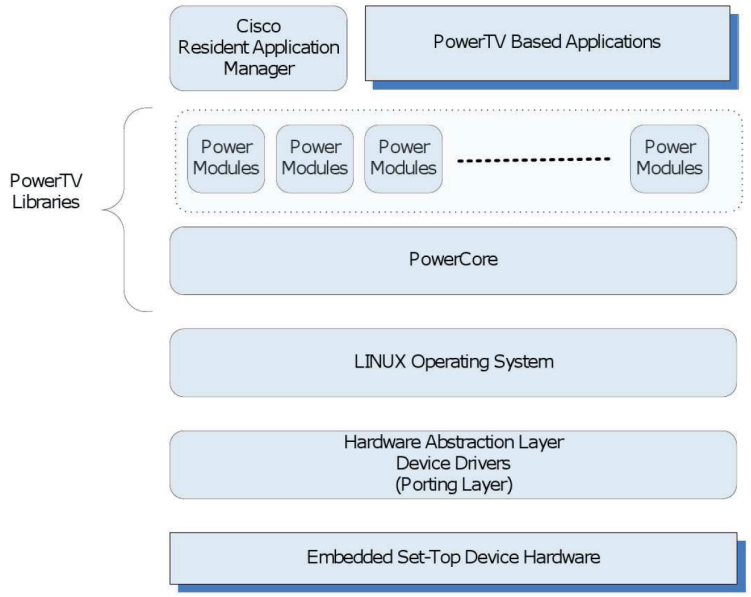


Figure 4.2: STB embedded software stack.

a MPEG video-output generator is used for video display on all STBs. Interconnect between the Set-Top Cluster and the Linux Cluster as well as RF transport for video and data is supported by a number of specialized devices, including the Cisco 7246 UBR Router. For transmitting and receiving DAVIC, DOCSIS and MPEG-2 data over the RF cable network, a set of QAM/QPSK modulators and demodulators is shown along with a RF combiner/splitter module that connects all devices together.

In summary, the prototype system described is representative of a real cable system that allows the execution of high-performance applications on the embedded processors of the set-top boxes under realistic operations scenarios. For instance, target applications such as the MSA program described in Section 4.5 are executed on the embedded processor, while the rest of the components in the STB, and particularly the MPEG video processing chain, are busy providing streaming-video content. In fact, the experimental results for the MSA application described in Section 4.6 were obtained while the STBs were simultaneously decoding a test set of MPEG videos for display on a collection of monitors and running an MSO interactive user interface with no perceived degradation of content display.

4.3 Software and Middleware

Set-Top Box Embedded Software. Each Cisco 4650 set-top device runs a hybrid software environment that consists of Cisco's PowerTV middleware layered over a embedded real-time operating system (RTOS) which is based on a Linux kernel (Fig. 4.2). All applications are written to run on top of the PowerTV application program interface, a proprietary Cisco API, and not on top of the Linux kernel. PowerTV itself consists of PowerCore, and PowerModules, a set of device-independent libraries. The PowerModules libraries provide functionality for communicating on the network, accessing applications or data from the BFS, tuning control such as changing channels, managing MPEG transport streams, encryption services, a widget library called PowerDraw for writing graphically rich applications, and a complete ANSI C library. In the prototype system, the TCP/IP PowerModule, which is compliant with BSD sockets, is used to implement the basic MPI send and receive routines as described in the next section.

While the Linux operating system and PowerCore resides in FLASH memory, all PowerModules are designed to be independent and downloadable over the network at boot time, or on-demand, through dynamic instantiation much like a dynamic shared library available in Unix. Additionally, there are special user-level processes such as the resident application manager that control all application life-cycles associated to user-interface applications, communications with the central services, as described in the previous section, and the overall STB user environment.

Open MPI System Software. The first prototype of the proposed Broadband Embedded Computing system is based on the use of the Open MPI programming model (version 1.2). Open MPI is an open-source implementation of the Message Passing Interface (MPI) standard for developing parallel applications that execute on distributed memory parallel cluster systems [100]. Open MPI is modular and configurable. It provides an extensible runtime environment called ORTE and MPI middleware APIs to support robust parallel computation on systems ranging from small mission-critical and embedded systems to future peta-scale supercomputers [39]. Open MPI is based on the *modular component architecture (MCA)*, a lightweight component architecture that allows for on-the-fly loading of frameworks, component modules, and runtime selection of features (including network device, OS, and resource management support), thus enabling the middleware to be highly configurable at runtime. Fig. 4.3 illustrates the MCA layered component approach [39]. Among all the Open MPI frameworks, the OMPI and ORTE frameworks are of primary interest

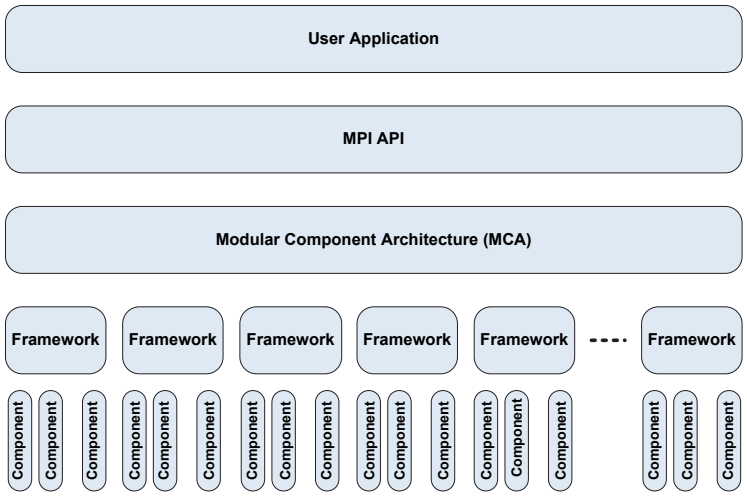


Figure 4.3: Open MPI component architecture.

because they support the point-to-point MPI operations and runtime process execution between the Linux Cluster and the Set-Top Cluster.

The Open MPI framework (OMPI) includes the actual MPI API layer as well as the underlying components necessary to implement the APIs. Consistent with the MCA, the MPI layer is actually built on top of other management and messaging layers. These, for instance, handle point-to-point messaging over TCP/IP, which supports the primary MPI APIs used in this work. In Open MPI this is implemented using the point-to-point management layer (PML), the byte transfer layer (BTL) and the BTL management layer (BML) [39]. PML handles the upper-level interface to the MPI API layer as well as message fragmentation, assembly, and re-assembly. BML abstracts the network transport layer by managing one or more BTL modules that support actual movement of data over various network interfaces such as TCP/IP over Gigabit Ethernet, high-performance Infiniband, or even a shared-memory multiprocessor systems executing MPI applications.

The Open Run Time Environment (ORTE) framework is primarily responsible for the MPI environment resource discovery and initialization, process execution, I/O, and runtime control. The execution of a parallel application is started by running the `mpirun` command which activates an ORTE daemon process. This process contacts each MPI host in the cluster to initiate a local peer ORTE process. During startup all ORTE processes participate in a resource discovery and

eventually begin execution of the parallel application on all hosts that make up the cluster. ORTE continues to coordinate the MPI application processes until the completion of its execution.

4.4 Porting Open MPI to STB Devices

In order to execute the target parallel application on our prototype system we developed an interoperable subset of the Open MPI software infrastructure to run on the Set-Top Cluster and interact with the full Open MPI 1.2 implementation running on the Linux Cluster. While trends in embedded computing lead to continued improvements in computational capabilities with each generation of STBs, currently these devices are still limited in terms of memory and processor resources. Hence, an early design consideration was to determine the *minimum subset* of the Open MPI API needed for developing a workload that would enable meaningful experimentation and performance evaluation with some real parallel applications. After analyzing the complete MPI specification and selecting the target application we determined that only the nine API functions reported in Table 4.1 were necessary for our current purposes. While the first six API functions are often sufficient to support many applications, the `MPI_Pack()` and `MPI_Unpack()` functions were added because they are required for the target application, i.e. the MSA program discussed in Section 4.5.

Leveraging the modularity and configurability of Open MPI we developed a library and runtime layer that is an inter-operable, compatible implementation of a subset of the OMPI and ORTE frameworks to run on the Cisco 4650 STB embedded software stack illustrated in Fig. 4.2. We included the following inter-operable components of the OMPI point-to-point framework: the MPI layer supporting the nine APIs listed above, a PML component implementation, and a BTL component for TCP/IP messaging over either Gigabit Ethernet or the DOCSIS broadband network. None of the other OMPI component frameworks were supported in the first prototype implementation. We combined the implementation of the API of Table 4.1 with the OMPI and the Open MPI ORTE software frameworks into a single PowerTV application library `cv_mpi` that is loaded on the STB during boot time. Parallel applications written for the PowerTV API running on the STB access this library at runtime in a way similar as MPI developers would load the dynamic shared Open MPI library `libmpi.so` on a Unix system.

API Function	Description
`MPI_Init()`	Initialize the MPI execution environment
`MPI_Finalize()`	Terminate MPI execution environment
`MPI_Send()`	Basic blocking point to point send operation
`MPI_Recv()`	Basic blocking point to point receive operation
`MPI_Wtime()`	Return elapsed time on the calling processor
`MPI_Comm_rank()`	Return the rank of the calling process in the communicator
`MPI_Comm_size()`	Return the size of the group associated with a communicator
`MPI_Pack()`	Pack a datatype into contiguous memory
`MPI_Unpack()`	Unpack a datatype from contiguous memory

Table 4.1: The set of supported MPI API functions.

The set-top ORTE module supports the minimum ORTE protocol transactions to launch, pass command arguments to, and terminate the given MPI parallel application process. For instance, when a parallel MPI application built using the API functions of Table 4.1 is started on the master Unix Sun 4200 host of our prototype system by running the command line "`mpirun -np 33 <args> <MPI program>`", the ORTE process running on the MPI Master Host contacts the 32 ORTE-compliant processes running on the corresponding 32 STBs. Each of these processes utilizes the DOCSIS TCP/IP network to download the parallel application on demand. Once the initialization of the runtime environment is completed, the MPI application starts its execution in parallel on the Linux Cluster and the Set-Top Cluster.

Limitations of the First-Generation Implementation. In order to execute the MPI applications launched from the Linux Cluster, the STB devices require an inter-operable implementation of the ORTE framework. By analyzing the standard ORTE framework we identified a number of challenges in terms of protocol overhead and fault tolerance. ORTE is a complex framework: a large portion of the implementation is designed for device discovery, exchange of process environment, and network information, occurring between all host devices. This creates scaling issues as the Set-Top Cluster size increases from thousands to millions of computational devices. As an illustration of the ORTE runtime overhead, testing of our experimental implementation revealed that ORTE requires a minimum of 165 bytes as measured by `tcpdump` per ORTE node-map data

structure. This list is sent to all hosts in the system during the ORTE initialization phase as part of a group "allgather" operation. This is acceptable for a typical cluster network of a few hundred or thousand nodes but it would not scale to a network of five million or more STBs because over 1GB of information would be sent to each STB, a quantity that exceeds the memory resources available in today's STBs. Hence, for our prototype system, we implemented a light-weight ORTE framework that relies on a statically-configured environment and consists of the minimal protocol that is sufficient to inter-operate with the full Open MPI environment of the Linux Cluster.

A second challenge to system scalability is the reliable execution of MPI applications. In many cases, the possible failure of a single MPI application would result in the termination of all processes within the process group. A large-scale system with millions of devices will likely have frequent failures. Hence, for future development of large-scale systems based on our architecture we plan to incorporate solutions such as those proposed as part of Open MPI-FT [78].

4.5 Multiple Sequence Alignment

To evaluate the feasibility of our heterogeneous parallel system as well as its performance in comparison to a more traditional computer cluster we chose *Multiple Sequence Alignment (MSA)*, one of the most important problems in bioinformatics. Since MSA is an NP-hard problem, in practice most of its instances require the use of approximation algorithms such as the popular ClustalW program [114]. There are various parallel versions of ClustalW, including ClustalW-MPI [85] and Mason (multiple alignment of sequences over a network) [26; 32]. We started from the source code of Mason, which is based on the Mpich implementation of MPI, and we ported it on our system, which uses Open MPI, so that we could run it on both the Linux Cluster and the Set-Top Cluster. This allows us to run multiple experiments to compare the parallel execution of different instances of the MSA problem on different system configurations of each cluster as discussed in Section 4.6. In all cases, the output of the Mason program produces a final multiple sequence alignment along with detailed timing measurements for each stage of the computation. Next, we describe the approximation MSA algorithm used by ClustalW and its Mason parallel implementation.

Solving MSA with ClustalW. MSA is the problem of aligning biological sequences, typically DNA sequences or protein sequences, in an optimal way such that the highest possible number of sequence elements is matched. Given a scoring scheme to evaluate the matching of sequence elements and to penalize the presence of sequence gaps, solving the MSA problem consists in placing gaps in each sequence such that the alignment score is maximized [26]. The ClustalW approximation algorithm consists of three phases. Given an input of N sequences, Phase 1 computes the *distance matrix* by aligning and scoring all possible pairs of sequences. For N input sequences, there are $\frac{N \cdot (N-1)}{2}$ possible optimal pair-wise alignments that can be derived with the dynamic programming algorithm of Needleman and Wunsch [96] as modified by Gotoh [41] to achieve a $O(n^2)$ performance, where n is the length of the longest sequence. The resulting distance matrix is simply a tabulation of score metrics between every pair of optimally-aligned sequences. Phase 2 of the algorithm processes the distance matrix to build a *guide tree*, which expresses the evolutionary relationship between all sequences. The guide tree can be derived with the *neighbor-joining clustering algorithm* by Saitou and Nei [106]. Phase 3 produces the final multiple sequence alignment of all N original sequences by incrementally performing $(N-1)$ pair-

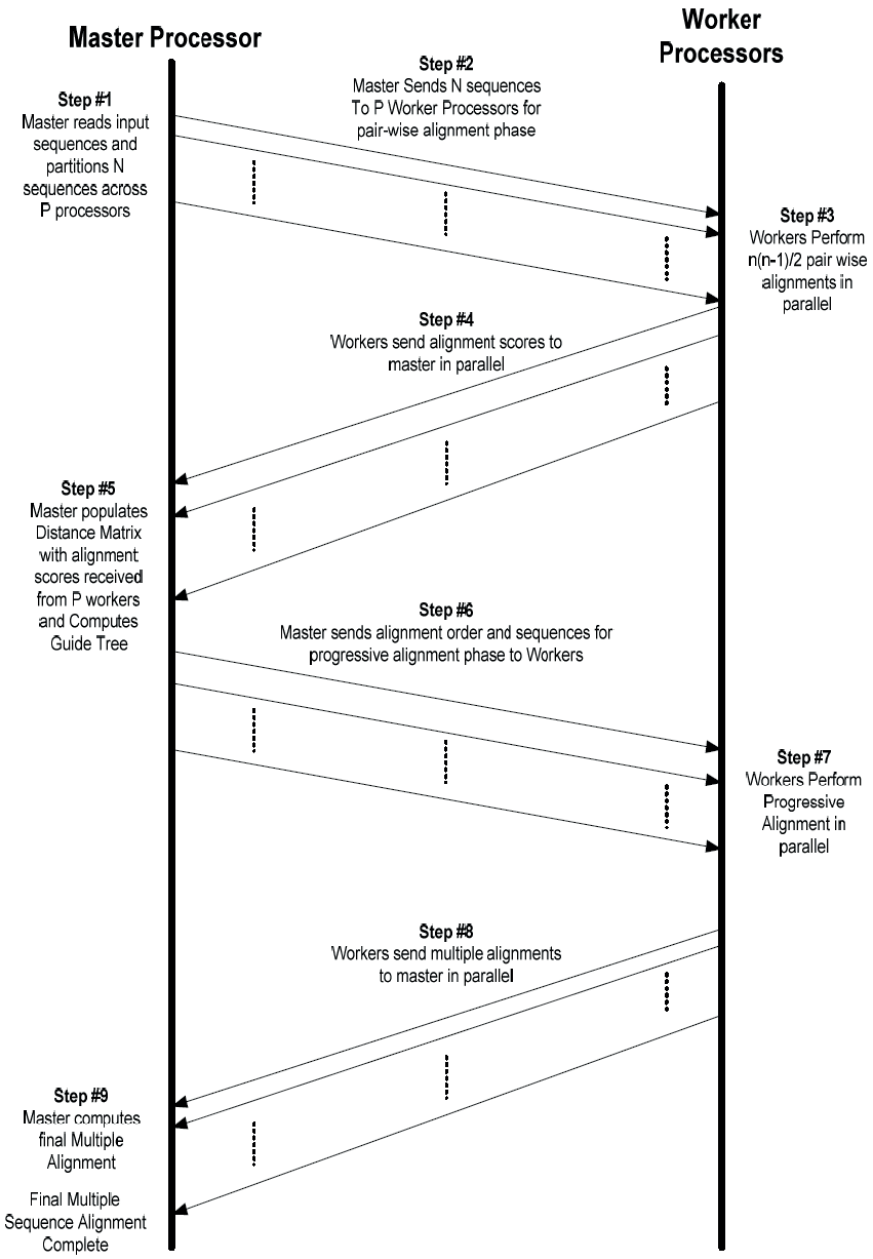


Figure 4.4: Parallel execution of ClustalW algorithm.

wise alignments in the order specified by the guide tree (*progressive alignment algorithm*) [37; 114].

Parallel MPI Implementation of MSA. Mason is a parallel implementation of ClustalW for execution on distributed-memory parallel systems using MPI [26; 32]. It proceeds through nine steps (Fig. 4.4):

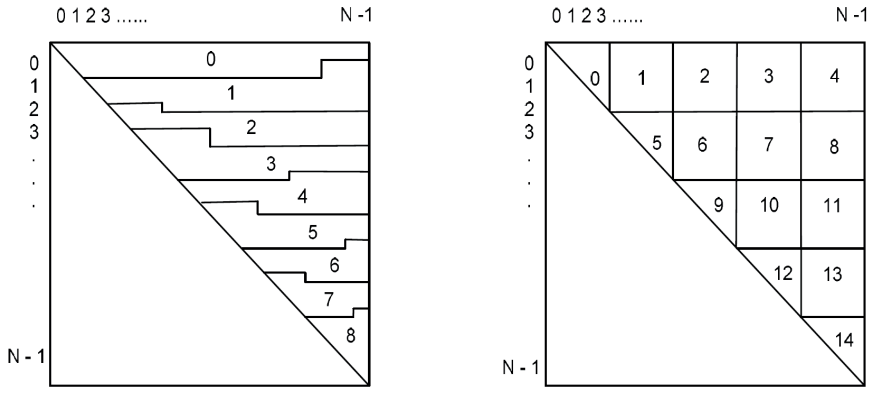


Figure 4.5: Alternative partitions of distance matrix.

1. The master host processor reads the input file containing N sequences and allocates an $N \times N$ distance matrix structure that is used to hold all optimal alignment scores between any two sequences. The master also partitions the distance matrix by dividing up the sequences among the P worker processors to distribute the workload, network, and resource requirements during the alignment step.

2. The master sends all required sequences to the P worker processors based on the distance matrix partitioning scheme computed in Step 1. Each processor has a fraction N_f of the total number of N sequences to align.

3. The P worker processors compute their $\frac{N_f \cdot (N_f - 1)}{2}$ alignments in parallel using the pairwise alignment dynamic programming algorithm.

4. All worker processors send their resulting alignment scores back to the master in parallel.

5. After receiving all scores and completing the distance matrix the master builds the guide tree.

6. The master computes and sends an alignment order along with the respective sequences required for progressive alignment to all worker processors.

7. All worker processors perform progressive alignment in parallel.

8. All worker processors send their partial multiple alignments back to the master in parallel.

9. The master progressively aligns the remaining multiple alignments to produce the final result.

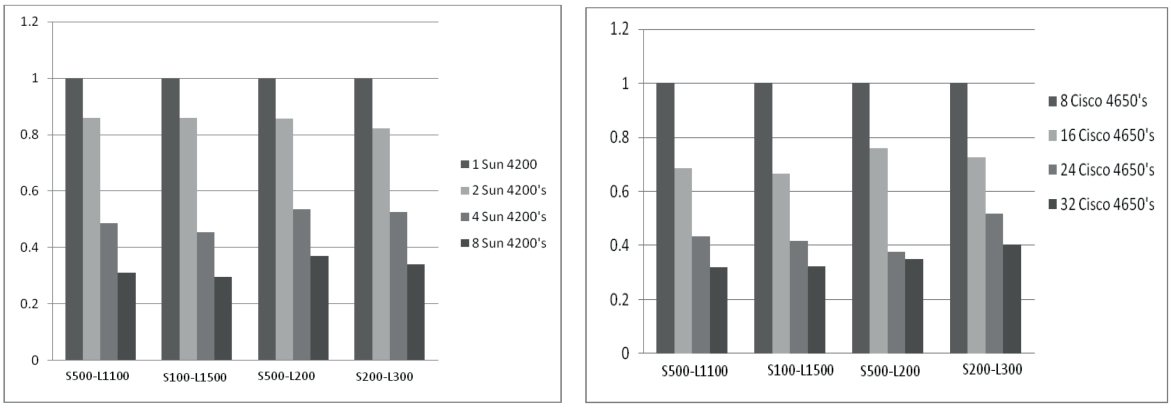
Processor		Overall Execution Time (Sec) - Floating Point				
Type	#	S500-L1100	S100-L1500	S500-L200	S200-L300	Avg.
	1	5129	830	426	162	
Sun 4200	8	1580	245	157	55	
	Speedup	**3.2**	**3.4**	**2.7**	**2.9**	**3.1**
	8	51989	8185	4256	1769	
Cisco 4650	32	16617	2651	1485	714	
	Speedup	**3.1**	**3.1**	**2.9**	**2.5**	**2.9**

Table 4.2: Overall execution times and speedups for two different system configurations of each cluster (Floating Point.)

Processor		Overall Execution Time (Sec) - Fixed Point				
Type	#	S500-L1100	S100-L1500	S500-L200	S200-L300	Avg.
	1	4666	757	383	146	
Sun 4200	8	1427	228	146	51	
	Speedup	**3.3**	**3.3**	**2.6**	**2.9**	**3.0**
	8	29732	4924	2575	1090	
Cisco 4650	32	9507	1629	908	441	
	Speedup	**3.1**	**3.0**	**2.8**	**2.5**	**2.9**

Table 4.3: Overall execution times and speedups for two different system configurations of each cluster (Fixed Point).

Experimental results by Datta and Ebedes show that 96% of computational time is spent in the derivation of the distance matrix, i.e. in the first three steps of Fig. 4.4, while the remaining time is split between the other two phases [26]. While the distance-matrix computation is the main target for parallelization, maximizing the achievable speedup depends on the strategy that is used to partition the matrix among the P worker processors. A possible approach consists in sending all N sequences to all processors so that each processor computes exactly $\frac{N \cdot (N-1)}{2P}$ pairwise alignments. This approach, which is illustrated in Fig. 4.5(left), distributes evenly the workload among the processors, but has the highest message-passing cost since all processors receive all N sequences, whether they are used or not. Alternative partitioning strategies have been proposed to reduce the communication cost [26]. The strategy shown in Fig. 4.5(right) assigns a square section of the distance matrix to most of processors, while some processors receive one of the smaller sections along the diagonal. This method reduces the amount of message passing in exchange for an unequal workload distribution. For small input sizes the first approach outperforms the second because the communication costs are still relatively low and all processors are fully utilized [26]. However, for



(a) Linux Cluster (float) (b) Set-Top Cluster (float)

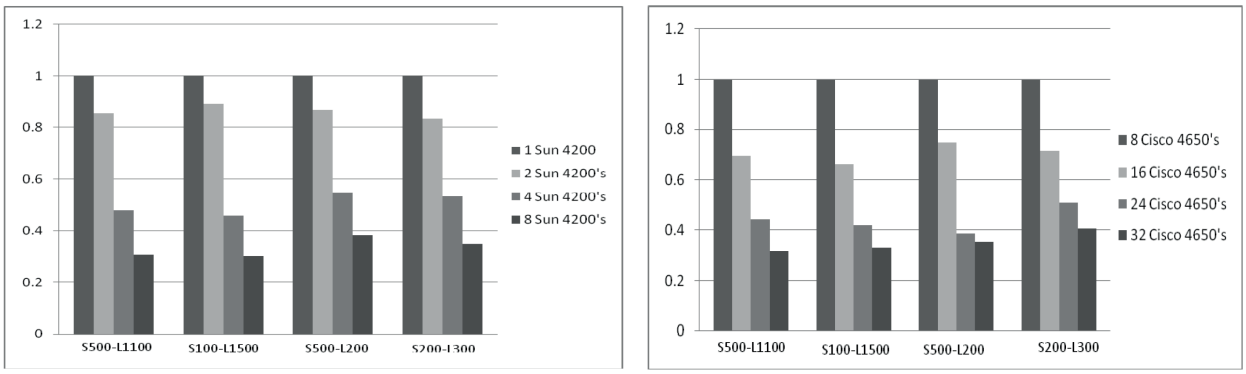
Figure 4.6: Relative parallelization speedup: Linux Cluster vs. Set-Top Cluster (floating-point computation).

aligning large sets of long DNA or protein sequences, which consist of perhaps thousands or even tens of thousands of sequences, the second approach may be more convenient because the resulting distance matrix can be partitioned such that communication costs and processor memory resources are minimized.

4.6 Experimental Results

Using the MSA problem as the target application, we completed a set of experiments on the prototype system of Fig. 4.1. While our system allows us to analyze various combinations of devices, in these experiments we focused on comparing different configurations of the Linux Cluster, which consists only of Sun 4200 nodes, with different configurations of the Set-Top Cluster, which consists only of Cisco 4650 STBs. Specifically, we considered four Linux Cluster configurations with $1, 2, 4$ and 8 Sun 4200 acting as worker processors and four Set-Top Cluster configurations with $8, 16, 24$ and 32 STBs acting as worker processors. Every configuration uses the same Sun 4200 processor as MPI master host. In each experiment we run MASON on a particular input data set that consists of a given number of DNA sequences.

Generation of Data Sets. Given an input file that specifies a set of transition probabilities, the ROSE sequence generator tool returns sets of either protein or DNA sequences that follow an evolutionary model and, therefore, are more realistic than purely random-generated sequences [112].



(a) Linux Cluster (fixed)

(b) Set-Top Cluster (fixed)

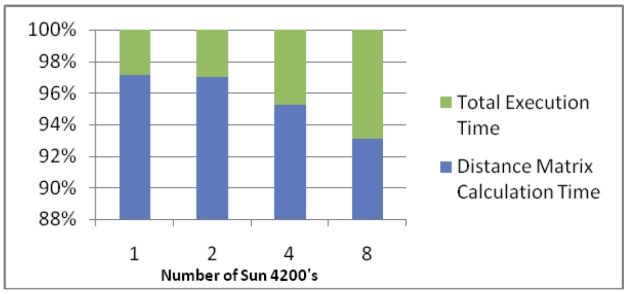
Figure 4.7: Relative parallelization speedup: Linux Cluster vs. Set-Top Cluster (fixed-point computation).

Using ROSE's standard input settings we generated four sets of DNA sequences, which present different size and complexity:

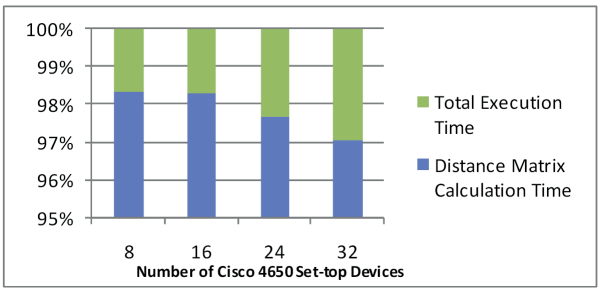
- *S500-L1100*: 500 DNA Sequences with 1100 base pairs.
- *S100-L1500*: 100 DNA Sequences with 1500 base pairs.
- *S500-L200*: 500 DNA Sequences with 200 base pairs.
- *S200-L300*: 200 DNA Sequences with 300 base pairs.

In general, larger sets of longer sequences are computationally more complex than smaller sets of shorter sequences. The algorithm partitions N sequences into $\frac{N_{part} \cdot (N_{part}-1)}{2}$ tasks, each requiring the pairwise alignment of M_{part} sequences. Hence, an upper bound on the computational complexity of MSA is given by $O((N_{part})^2 \cdot (M_{part\_max})^2)$, where M_{part_max} denotes the longest sequence in the set of M_{part} sequences making up any given sequence partition N_{part}. In our collection the *S500-L1100* set is the most computational complex while the *S200-L300* set is the least complex. The other cases lie somewhere in between.

Floating-Point vs. Fixed-Point Operations. As part of our experiments, we analyzed also the performance impact of converting floating-point operations in MASON to fixed-point operations for each possible configuration of both clusters. This analysis is important because current STBs do not feature a floating-point hardware unit and only support floating point operations through emulation in software. Hence, estimating how much of the current performance gap is due to the lack of this unit allows us to better extrapolate the performance that could be obtained when running



(a) Linux Cluster configurations



(b) Set-Top Cluster configurations

Figure 4.8: Execution time breakdown for each of the four configurations of the two clusters.

other algorithms, which necessarily require floating-point precision, on future clusters of next-generation of embedded devices, which are expected to contain floating-point units. We focused our effort on converting to fixed-point operations only the code in the first phase of the algorithm (distance-matrix computation) because it accounts for over 90% of the overall computation time. The conversion was achieved by replacing float variables with long integers and multiplying by a constant factor sufficient to maintain five digits of precision in all scoring routines comprising the pairwise-alignment algorithm. Results were converted back to floating point from fixed-point values by dividing all fixed-point score values by a constant factor prior to populating the distance matrix. In terms of accuracy, we found less than 1% difference in results beyond five digits of precision.

Execution Time Breakdown. Figures 4.8(a) and 4.8(b) report the execution time breakdown of the floating-point version of MASON with the S500-L1100 input data set for each of the four configurations of the Linux Cluster and Set-Top Cluster, respectively. The results for the fixed-point version as well as for the other data sets are similar. In both cases, as expected, the distance-matrix computation accounts for well over 90% of the overall execution time regardless of the number of processors. This number varies from 1 to 8 for the Linux Cluster and from 8 to 32 for the Set-Top Cluster. These results are in line with similar results presented in the literature for other parallel implementations of CLUSTALW [32; 26; 85] and confirm the validity of the parallelization efforts discussed in Section 4.5.

Parallelization Speedup. Table 4.2 and Table 4.3 reports the overall execution times in seconds obtained running both the floating-point version and the fixed-point version of MASON with each input data set on two configurations of the Linux Cluster (consisting of 1 and 8 Sun 4200

Data Set	32 STB vs. 1 Sun 4200		
	(float)	(fixed)	Gain (%)
S500-L1100	3.24	2.04	37
S100-L1500	3.19	2.15	33
S500-L200	3.49	2.37	32
S200-L300	4.41	3.02	31

Table 4.4: Comparing a Set-Top Cluster with 32 boxes to 1 Sun 4200: gain due to fixed-point computation.

processors respectively) and two configurations of the Set-Top Cluster (consisting of 8 and 32 Cisco 4650 STBs respectively). For instance, for the case of the floating-point version of MASON, the data set for the hardest problem (S500-L1100) is processed in $5129s$ by a single Sun 4200 while a cluster with eight Sun 4200 processors takes only $1580s$ (a speedup of 3.2). Meanwhile, for this problem, a cluster of eight Cisco 4650 devices need $51989s$ but this time is reduced to $16617s$ for a cluster of 32 devices (a speedup of 3.1). The speedup values are similar across the four data sets and regardless of the program version (floating-point or fixed-point) with an approximate average value of $3X$ for the Linux Cluster and $2.9X$ for the Set-Top Cluster. Figures 4.6 and 4.7 illustrate the relative speedups due to parallelizations normalized to the slowest computation time for each data set as we increase the number of nodes for each of the four configurations of the Linux Cluster and Set-Top Cluster for both types of computation, respectively. Notice that while the execution time does not decrease linearly as we increase the number of Sun processors (or Cisco devices), an important result is that *across all the various data sets both the Linux Cluster and Set-Top Cluster exhibit similar scaling and performance improvements, with comparable speedup due to parallelization in both platforms.*

How Many Set-Top Boxes to Make a High-Performance Processor? The first row of Table 4.4 reports the ratio of the execution time of the fastest Set-Top Cluster configuration (32 Cisco 4650) over the execution time of the slowest Linux Cluster configuration (1 Sun 4200) for the various data sets. For instance, this ratio is $16617/5129 = 3.24$ when the two clusters run the floating-point version of MASON with the S500-L1100 data set. In other words, 32 STB devices take 3.24 as much time as a single Sun processor to perform the same task. Or, again, we could say that one high-performance processor is equivalent to 104 STB devices. On the other hand, this ratio

drops to 2.04 when the fixed-point version of MASON is used *for both clusters* [1], an improvement of 37% as shown in the remaining rows of Table 4.4. This translates in a new *equivalence gap factor* of 66 STBs per high-performance processor. Finally, notice that the frequency of the Sun 4200 processor is four times as fast as the frequency of the embedded processor in the Cisco 4650 STB. Hence, accounting for this frequency ratio, the equivalence gap factor would become 16.5.

4.7 Lessons Learnt

Impact of Communication. In comparing the relative platform speedups we observe that the communication cost or overhead due to MPI is not a factor on either the Linux Cluster and Set-Top Cluster for our experimental system. This is due to the small size of the experimental broadband network and limited number of embedded devices. In a larger broadband network with many STBs, a key consideration will be assuring ample network bandwidth and minimal latency to each embedded device. MSOs are currently supporting millions of embedded devices over DOCSIS networks and plan to continue improving overall network performance through ongoing protocol enhancements that will include the following:

1. Quality-of-service (QoS) enforcement on a per-application basis assuring minimum bandwidth allocations.

2. Increased network performance in both directions to achieve over 300Mb/s downstream and 120Mb/s upstream through channel bonding. Channel bonding is part of the DOCSIS 3.0 standard which increases network performance by concatenating multiple channels into one larger virtual channel [48].

3. Making best use of the DOCSIS multicast and broadcast capabilities wherever possible.

Additionally, MSOs are improving the DOCSIS broadband network by continuing to reduce the number of embedded devices per DOCSIS channel. With fewer devices per channel, additional

[1] Notice that using the fixed-point computation for the MSA problem is faster for both clusters, but is relatively better for the Set-Top Cluster because the STBs do not have a floating-point hardware unit. Naturally, there are other important scientific problems that necessarily require floating-point operations. Our approach to parallel computing will be applicable to these problems only when STBs will feature a floating-point unit, which is expected to happen soon [34].

access slots within the TDMA scheme are available, thus increasing the throughput for all devices on that channel. There are, however, challenges wherever collective communications are part of the underlying computation because DOCSIS communications among set-top, as all DOCSIS devices, occur through centralized broadband routers. This introduces additional latencies for peer-to-peer communications and, therefore, must be accounted for within the application design of collective communications.

4.8 Related Works

While the utilization of large scale networks of embedded devices for heterogeneous computing within a managed, dedicated system cloud raises new challenges and opportunities in system scalability and performance, the idea of harnessing distributed embedded systems, particularly over the Internet, is not new. A number of initiatives have focused on utilizing volunteer PC hosts or game consoles for solving difficult problems such as those found in Computational Biology. For instance, Folding@Home [50] and GridRepublic [63] were formed to leverage the enormous number of idle processor cycles available on the Internet. In these projects, specialized software is downloaded to participating PCs or game consoles, such as the PlayStation 3 featuring the IBM Cell multi-core processor, with units of computation dynamically offered by subscribers typically through a screensaver application or background agent running on the host device. In this model, when the agent is available it receives tasks assigned by a master server to be processed on a best-effort basis. Results are sent back to the master server when the task is completed. This system offers scalability on the order of the number of PCs or game consoles active on the Internet at any give time. Its success shows the potential of harnessing distributed embedded devices that are widely deployed by the tens of millions units today. Still the fact that the "device participation" is not predictable ultimately limits throughput guarantees, maximum task concurrency, and service-level agreement between actors.

Most related works in the area of integrating message-passing middleware into embedded devices have focused on reducing the size of the MPI stack. Lightweight MPI (LMPI) is based on a thin-client model where the MPI API layer is implemented on top of a thin-client-message protocol which communicates with a proxy server that supports a full MPI stack [7]. LMPI client requests

are routed through the proxy server, which acts on behalf of one or more MPI thin-client user processes. A key benefit of the LMPI approach is the elimination of the need for an operating system on the embedded device. Other approaches attempt to reduce the size of the MPI stack through refactoring or are based on a bottom-up implementation of a minimal MPI stack, as in the case of Embedded MPI (eMPI) [91]. Similarly to eMPI we use a bottom-up approach to implement the minimal MPI API set. But in our proposed system each Cisco STB contains a modern real-time operating system that can support a native MPI implementation. Also, while previous work in executing MPI on embedded devices has focused on small test kernels, from an application-viewpoint our work is closer to the work of Datta [32; 26] and Li [85] because we evaluate our parallel system with a real workload.

4.9 Summary

A proposed heterogeneous platform architecture for distributed computing, in particular broadband embedded computing, that leverages traditional Unix cluster technologies in combination with a broadband network of embedded set-top boxes (STB) is presented and contributed as unique work in the area of heterogeneous computing and embedded systems. An implemention of a complete prototype system that fully represents a scaled-down version of the proposed architecture including an inter-operable subset of Open MPI to integrate the systems. A parallel version of the ClustalW bioinformatics application is ported on the system by completing the necessary optimizations to reduce the memory requirements for execution on the STBs and improve parallel workload data distribution. Experimental testing established that it is possible to execute ClustalW on the prototype system while the STBs continue simultaneously to operate their primary functions, i.e. decoding MPEG streams for monitor display and running an interactive user interface, without any perceived degradation. Further, experimental results show that scaling up the system by adding more STBs to the embedded cluster gives equivalent performance gains as scaling up the number of processors in a traditional Unix cluster. While the proposed platform architecture has the potential of scaling to millions of units, a number of critical challenges in the area of protocol overhead and lack of complete interoperability when implementing the MPI ORTE runtime environment and MPI library for embedded devices is identified. Addressing these challenges is my next goal and

will be presented in Chapter 5.

The presented work however validates the feasibility and motivation of developing a more complex system infrastructure for Broadband Embedded Computing. Given the technology trends in MSO broadband networks and in hardware/software solutions for STBs, as well as other embedded devices operating on service provider networks, this study demonstrates that to leverage a broadband network of embedded devices for Broadband Embedded Computing is not only an interesting proposition for supplying a low-cost and energy-efficient computing platform, but can support both computationally-intensive service-provider workloads, heterogeneous Cloud services, and emerging consumer-driven applications that require a computational workload distributed accross data-center clusters and a network of embedded devices.

Chapter 5

Second Generation System For Broadband Embedded Computing Utilizing Open MPI

5.1 Introduction

In this chapter a second-generation heterogeneous system for broadband embedded computing based on distributed message passing and Open MPI is implemented. The implementation improves upon the system described in Chapter 4 in a number of key areas including:

1. A new contribution in architecture and implementation whereby runtime environment scalability is made possible through a generalized virtualization framework model that maps and abstracts embedded devices within the processor space of data-center class server hosts; The new framework also includes a contributed caching technique for minimizing embedded device memory requirements and reducing network latency during Open MPI runtime environment and system-wide application process initialization.

2. A full implementation of the MPI library primitives, including collective operations.

3. A larger experimental system implementation that now includes 128 set-top devices with enhanced memory and processor performance compared to those devices utilized in the system

of Chapter 4.

Additionally, new experiments to fully characterize the second-generation system and results are presented.

The second-generation system prototype implements a small-scale version of the proposed larger scale MSO service-provider system where a Linux Cluster features nine high-end blade servers and the Embedded Cluster now consists of 128 STBs. The two clusters are interconnected through the broadband network of a complete head-end cable system (as described in Section 5.2). While the cable system remains fully operational in terms of its original function (e.g. by distributing streaming-video content to the STBs which render it to their displays), it is possible to simultaneously and effectively execute other parallel applications demonstrating the STB as a broadband computing resource by leveraging the additional computation resources that are available in the STB multi-core processors. Specifically, in the second-generation system, a complete port of the Open MPI software library to the set-top box optimized for embedded devices is implemented. As discussed in Section 5.3, this porting posed important challenges in terms of resource management and scalability. These challenges are addressed by performing a virtualization of the embedded processors that allows them to transparently inter-operate with the computer cluster using the message-passing model (Section 5.4).

To evaluate the resulting second-generation system and its utility for broadband embedded computation using Open MPI, various experimental results were carried out as described in (Section 5.5). First, it is demonstrated that the system can execute the complete set of benchmarks for point-to-point and collective operations that are part of Intel MPI IMB benchmarks. Then, in order to gain further insight into the relative performance scaling of the Embedded Cluster versus the Linux Cluster, we run two important parallel applications: ClustalW-MPI (The bioinformatic parallel sequence alignment application discussed in Section 4.5) and Tachyon (a parallel ray-tracing application). The experimental results confirm the important convergence trend between traditional computing and embedded computing using Open MPI and support the case for broadband embedded computing.

The chapter is organized as follows: First, I describe the heterogeneous system architecture that includes a Linux Cluster and complete next-generation broadband cable system with contemporary embedded set-top devices. Next, I describe an important contribution of this dissertation: a model

and implementation of embedded processor virtualization for integrating a distributed embedded system into a Linux computational cluster. These contributions consist of server and embedded client software implementations which I refer to as the Open Embedded Runtime Environment, or OERTE fully described in Section 5.4. Section 5.4 concludes with a description of the second-generation Open MPI Embedded libraries significantly expanded to include complete support of MPI version 2.0 primitives for full inter-operability with MPI standard version 2.0 compliant hosts. The remaining sections discuss the experimental workloads: IMB, ClustalW-MPI, and Tachyon, used for evaluating the system, followed by an analysis of test results. Finally, I conclude with related work and a summary for this chapter.

5.2 The System Architecture

Fig. 5.1 provides a complete view of the second-generation heterogeneous system architecture for broadband embedded computing system that we designed and implemented. It is composed of four main subsystems.

Computer Cluster. The Linux Cluster consists of a traditional network of nine blade servers and Network Attached Storage (NAS). Each blade has two quad-core 2.0GHz Xeon processors with 32GB of memory and 1Gb/s Ethernet interface. Each processor runs Debian Linux. One of the nine blades acts as Master Host, i.e. is dedicated to the Open MPI runtime management and is the master server for the Linux Cluster and the Embedded Cluster host nodes. These use NFS to mount the 2TB Sun storage array which provides a remote common file-system partition to store both applications and data for each of the executing MPI processes across both clusters. The master system also hosts the virtualization software to map the embedded processors into the runtime environment of the Linux Cluster.

Embedded STB Cluster. The Embedded Cluster consists of 128 Samsung SMT-C5320 set-top boxes (STB) that are connected with a radio-frequency (RF) network for data delivery using MPEG and DOCSIS transport mechanisms. The Samsung SMT-C5320 is an advanced (2010-generation) STB featuring a dual-core SoC with a Broadcom MIPS 4000 class processor, a floating-point unit, dedicated video and 2-D/3-D-graphics processors with OpenGL support, 256MB of expandable system memory, 64MB Flash memory, and many network transport interfaces (DOC-

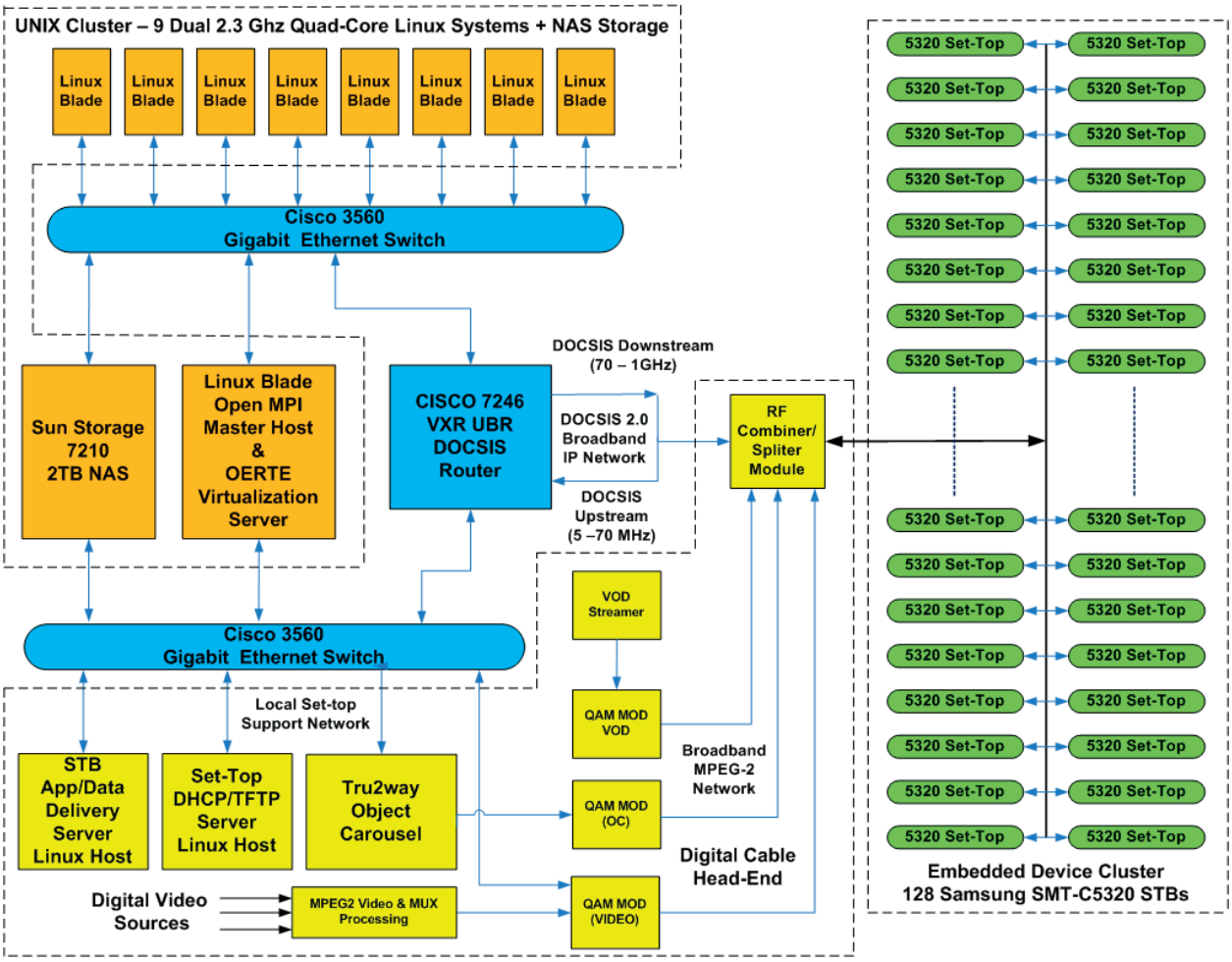


Figure 5.1: The proposed heterogeneous system architecture for broadband embedded computing.

SIS 2.0, MPEG-2/4 and Ethernet). Indeed an important architectural feature of modern STBs is the multi-core architecture design which allows the MIPS processor, graphics/video processors, and network processors to operate in parallel over independent buses. Hence, user-interface applications (such as the electronic programming guides) can execute in parallel with any real-time video processing. *Indeed, it is the growing parallel-computing capability of the emerging SoC architectures for STBs that enables the execution of applications outside the realm of interactive-TV, thus opening the opportunity for large-scale broadband embedded computing that we are pursuing with our work.*

Digital Cable Head-End. This is responsible for controlling the Embedded Cluster devices and providing all interactive television services including: electronic program guide, user-interface,

video-on-demand (VOD), and the delivery of MPEG-2 videos. Our digital head-end supports the current generation of STBs based on the Cablelabs Tru2way standard [6] and is a *scaled-down but complete implementation* of a modern digital DOCSIS-based broadband cable system in-use at today's largest MSOs. As shown in Fig. 5.1, its core components include:

1. The Tru2way Object Carousel for MPEG-2 delivery of Embedded Cluster applications and Tru2way-standard STB signaling.

2. Two Linux hosts for TCP/IP DHCP and TFTP network services, which are required for assigning system-wide IP addresses and DOCSIS cable-modem configuration data to all Embedded Cluster devices.

3. A HTTP application/data server that supports interactive television services via TCP/IP over the DOCSIS network.

4. Support for MPEG-2 video sources that are multiplexed and grouped into digital channels, including a single channel for VOD streams.

5. A RF distribution and combining network that utilizes a Cisco QAM modulator device to translate digital input signals from the carousel, multiplexed MPEG sources, and VOD server, into modulated QAM256 RF frequencies, which can be combined with the DOCSIS router RF output to feed the broadband network of STBs.

DOCSIS is a standard broadband-network technology for TCP/IP over RF cable that is described in Section 2.4.1.1 [57]. It provides for an inter-operable RF modem, based on TDMA protocols organized in a star topology connecting the central router and the STBs. The SMT-C5320 DOCSIS 2.0 TCP/IP and MPEG-2 transport stream interfaces use quadrature amplitude modulation (QAM) protocols for transmitting and receiving signals on North American digital cable systems. Devices on DOCSIS share access to the network, as arbitrated by the central router, and operate effectively at up to 27Mbps in the downstream direction (towards the STB) and 27Mbps in the upstream direction (towards the cluster). The MPEG-2 interface is primarily used for decoding video programs, but can also receive applications or data delivered via the Tru2way Object carousel (OC), a *broadcast file system* service on a dedicated QAM frequency. This data is sent from the head-end at regular cyclical intervals—hence the term carousel—over MPEG-2 directly

into a QAM device where it is modulated onto the RF cable plant at a specified frequency for STB reception. Broadcast applications are STB executables or data that are simultaneously available to all STBs connected to the broadband network. A STB device tunes to a specific channel frequency and receives the application/data of interest according to the Tru2way protocol. The carousel may also deliver Tru2way signaling and other forms of data over DOCSIS as multicast group messages following the DOCSIS Set-top Gateway, or DSG protocol [4]. *In the prototype system this data-delivery mechanism is used to control the STB boot-up and user-interface applications.*

For non-broadcast applications, a Linux application/data server is used as an Apache HTTP server that delivers application executables and data in parallel to all requesting STBs through the DOCSIS TCP/IP broadband network. As mentioned previously, basic TCP/IP network services are provided by a Linux host running DHCP and TFTP. DHCP is used to assign an Internet address to each STB. TFTP is the primary method for distributing DOCSIS configuration information. A video-content channel using a single video source and a MPEG video-output generator is used for video display on all STBs.

Network. The system network is a managed dedicated broadband network which is divided into three IP subnets to isolate the traffic between the DOCSIS-based broadband Embedded Cluster network, the Linux Cluster network, and the digital cable head-end. Its implementation is based on two Cisco 3560 1Gb/s Ethernet switches and one Cisco 7246 DOCSIS broadband router. The upper switch in Fig. 5.1 interconnects the 8 blades along with the NAS and master host. The lower switch aggregates all the components on the head-end subnetwork. The DOCSIS subnetwork is utilized by the Embedded Cluster whose traffic exists on both the Linux Cluster and the digital head-end network. The broadband router has 1Gb/s interfaces for interconnection to the Linux Cluster and head-end networks and a broadband interface for converting between the DOCSIS network and the Ethernet backbone. Each broadband router can support over 16,000 STBs, thus providing large scale fan-out from the Linux Cluster to the Embedded Cluster. While in a normal cable system the Linux Cluster and the digital cable head-end do not necessarily need to share traffic, we connected them over Gigabit Ethernet because this enables, for instance, the execution of MPI collective operations among the Linux Cluster and Embedded Cluster nodes in a seamless way.

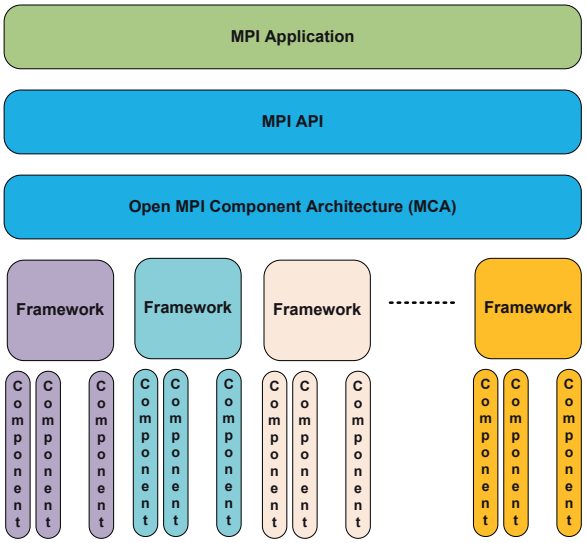


Figure 5.2: The Open MPI modular component architecture (MCA).

In summary, while being representative of a real cable system, our prototype system allows us to execute MPI application processes simultaneously on both the Linux Cluster blades and the Embedded Cluster processors under realistic operations scenarios. For instance, we can execute multiple workloads such as the IMB benchmarks described in Section 5.5.2, the MSA application described in Section 4.5, and the Ray Tracing application described in Section 5.5.3 on the embedded processor, while the rest of the components in the STBs, and particularly the MPEG video processing chain, are busy providing streaming-video content. *A key element of our system is the managed broadband network, which not only enables the heterogeneous system implementation, but it offers also a dedicated and massively-scalable infrastructure that can be leveraged for broadband embedded computing.* In fact, the experimental results for the MSA application described in Section 4.6 were obtained while the STBs were simultaneously decoding a test set of MPEG videos for display on a collection of monitors and running an MSO interactive user interface with no perceived degradation of content display.

5.3 Open MPI: Basics and Challenges

Open MPI is an open-source implementation of the Message Passing Interface (MPI) library for development of parallel applications on distributed memory computer architectures [100]. Open MPI is the result of merging and combining three main previous MPI implementations and is currently among the most popular libraries for high-performance computing applications. As illustrated in Fig. 5.2, the Open MPI design is centered on the Modular Component Architecture (MCA), which provides a flexible and configurable environment for design-time development and run-time installation of various software frameworks [39; 100]. An MCA framework is a construct that is created for a single, specific tasks and provides a public interface. Examples of tasks are: the launch of processes on the local host, the execution of collective operations, and forwarding input-output from MPI application processes. A framework uses the MCA services to find and load components at run-time. An MCA component is a self-contained implementation of a framework's interface, which can be inserted into the Open MPI code base at run-time and/or compile-time. An MCA module is an instance of a component. The component modules are self-contained software units that export well defined interfaces and can be dynamically selected and composed with other modules at run-time.

Open MPI is a large project with many different sub-systems. Fig. 5.3 shows the three major ones, which build on each other according to a layered and structured model. OMPI is the top layer and contains the actual implementation of the MPI application program interface. The *Open Runtime Environment (ORTE)* is responsible for managing the launch and runtime lifecycle of the parallel processes of a given MPI application. Both OMPI and ORTE rely on the underlying Open Portability Layer (OPAL), which contains the utility and "glue" code needed to integrate the higher-layer component modules with the native (host) operating system.

The Open Runtime Environment (ORTE). In the Open MPI 1.4.2 release, the ORTE subsystem follows the Open MPI MCA architecture and has 14 distinct frameworks, which offer a flexible and highly-configurable runtime environment by supporting various tasks including: managing process mapping or affinity, launching of MPI processes onto physical processing cores, managing of MPI cluster-wide process lifecycle during execution, error messaging, redirection of process I/O, and process-wide group communications facilities. Thanks to the MCA, ORTE framework components may be replaced with different implementations, all dynamically configurable at

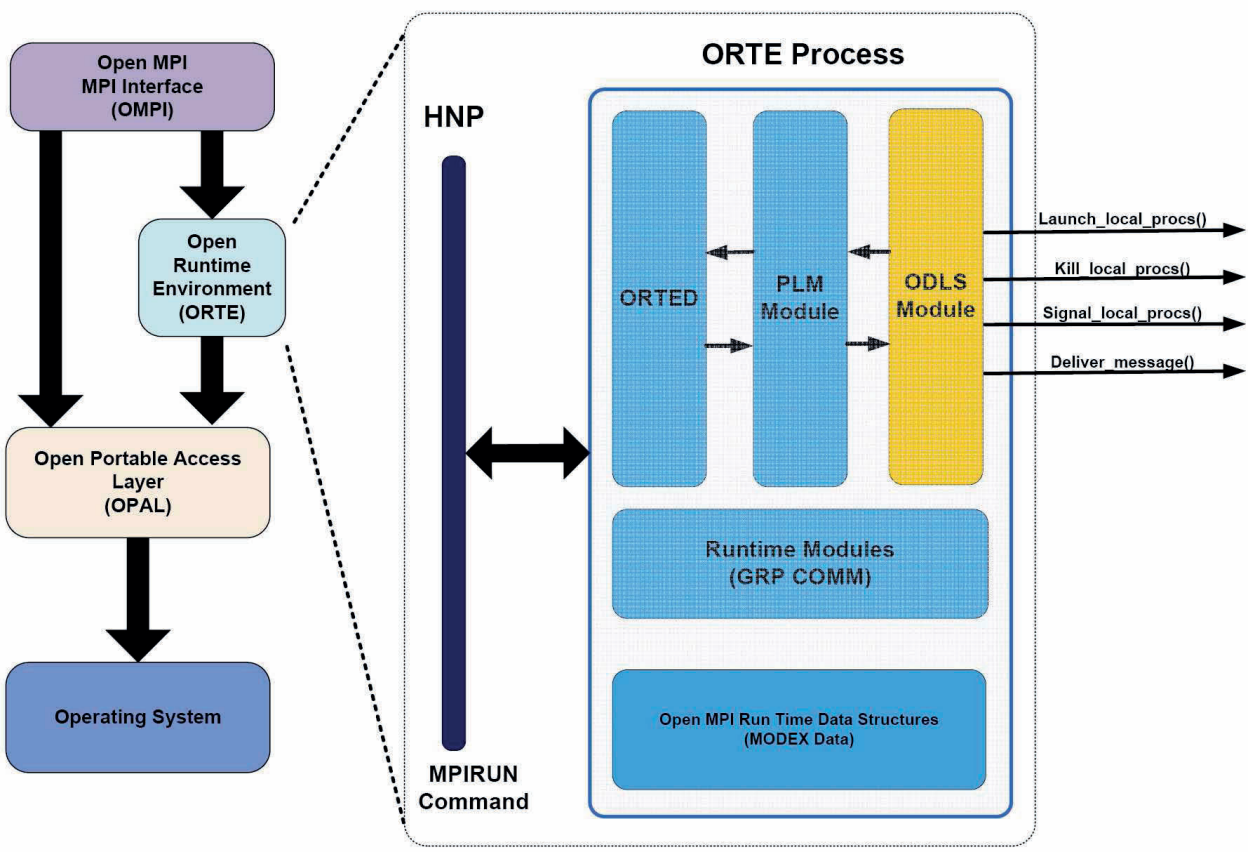


Figure 5.3: The three Open MPI subsystems and the primary ORTE modules.

runtime. Fig. 5.3 shows the primary ORTE modules required in our virtualization implementation described in Section 5.4.2. The execution of a MPI job is initiated by running the `mpirun` command on a computer in the cluster, which therefore becomes the *host node process (HNP)*. As the newly-designated master node, the HNP initiates one or more ORTE Daemon (ORTED) processes on each client host node supporting ORTE, through a remote execution protocol (e.g. RSH or SSH), or a specialized process-launcher communication protocol. Each ORTED process communicates with the Process Lifecycle Management (PLM) module whose functions include controlling the actual mapping of MPI processes to the processing cores and managing their complete execution. For each process, this includes runtime initialization, application launch, signaling, message delivery, and termination. The PLM performs these operations in conjunction with the *ORTE Daemon Local Launch System (ODLS)*, a module which defines an interface contract for each of them and

launches local processes on the MPI host node. Finally, the ORTE framework uses `grpcomm`, a group communication module, to distribute message information among the peer ORTE client hosts and the HNP.

Challenges in Porting Open MPI to Embedded Devices. One important message operation required to launch any Open MPI application is the sharing of per-process module information, or *module exchanges (modex)*. This is performed by `grpcomm` which executes an `allgather()` operation as follows: each ORTE client gathers local process information and sends it to the HNP, which assembles the information for all processes running on every client and then re-distributes it to all clients. The operation requires fairly high network bandwidth and fairly large memory on each host. Further, these requirements scale up with the number of hosts in the cluster. This represents a significant *scaling challenge* for the effective utilization of Open MPI in a heterogeneous computing system that aims at leveraging millions of embedded devices as the one that we envision. For instance, if we assume a typical modex data-structure of 500 bytes, and a cluster with P hosts and N MPI processes per host, then each host must store $P * N * 500$ bytes of data. In a large system where P can be of the order of millions, even if we have a small number N of processes, the memory requirements to store modex data could easily exceed 1GB per embedded device. This is an unrealistic requirement for today's embedded SoCs. But even if future SoC architectures were able to accommodate it, it would require to consume a significant data-transfer time simply to copy into the host memories, thus undermining the performance gains from parallelizing the computation. In the next section we discuss how we addressed these challenges. In particular, we leveraged the Open MPI MCA architecture to modify the functionality of the ORTE subsystem by replacing the ORTE ODLS component module with a new ODLS module. This new software framework is called *OERTE*, standing for *Open Embedded Runtime Environment*. The new module is interfaced with a newly-developed embedded version of the ORTE framework to support the virtualization of the embedded processors. This allows us to decouple the embedded-process modex management so that all ORTE modex operations are performed at the server side (as described in Section 5.4.2).

5.4 Embedded Processor Virtualization and Embedded Software Optimization

This section describes the characteristics of the embedded STB software environment and contributed model implementation towards solving a fundamental problem in launching a large number of MPI processes across a broadband network of distributed embedded devices such as STBs for heterogeneous computing. In particular, two new implementations are described; 1) the new OERTE server and client software implementation of ORTE for MPI application execution across embedded devices that supports the virtualization of embedded processors into the Linux Cluster and, 2) a complete implementation of Open MPI software libraries, optimized for resource constrained embedded devices.

5.4.1 Embedded STB Software Environment

The STB software environment is based on an embedded version of Linux with a reduced footprint of only 16MB. This was obtained by minimizing the size of the required kernel, utilities, and associated libraries, which are resident in Flash memory. For example, the embedded Linux operating system does not include facilities for desktop window systems, development tools, multiple shell environments, or utility packages typical of a full-package Linux distribution. The shell, provided by BusyBox [56], consists of over 100 common Linux and GNU utilities, whose implementation is highly optimized for embedded devices, requiring only 400KB. The kernel, based on the Linux v2.6 distribution [66], includes support for threads, BSD socket interfaces, network services (NFS, DHCP, Telnet, etc.), and standard GCC libraries. This is sufficient for developing and executing sophisticated multi-threaded Linux applications and MPI applications.

The STB initialization process occurs as follows: during power-on or a reboot, the STB Flash-based boot-loader starts the Linux kernel. This executes all initialization scripts found in the `/etc/init.d` and `/etc/rc.d` directories. All network interfaces are configured using DHCP and the remote file-systems are NFS-mounted from the NAS Storage. To support the development of interactive television applications, the STB also initializes a Java Virtual Machine and a set of Java class instances. After the execution of all STB initialization scripts, a start-up script in `/etc/rc.local` runs a special Linux application which provides a runtime-environment manager

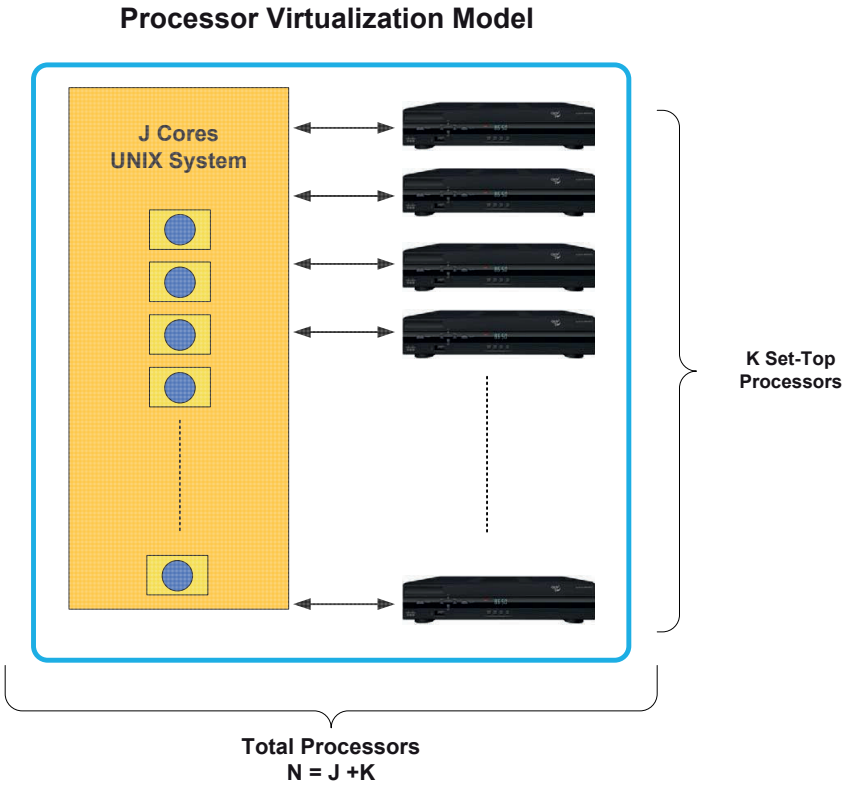


Figure 5.4: Embedded device virtualization.

required for the STB interoperation with the Linux Cluster Open MPI environment. The runtime environment application OERTE is described further in Section 5.4.2. The OERTE client executes as a background daemon process listening for OERTE server side command and control signalling to manage the launch and lifecycle management of all STB executing MPI applications.

5.4.2 Embedded Processor Virtualization

A critical step and key contribution in the design of a heterogeneous system for broadband embedded computing is the virtualization of the STB embedded processors in the context of the Open MPI ORTE and the TCP/IP networking environment. First, embedded processors in the Embedded Cluster network are mapped into the processor domain of the Linux Cluster system.

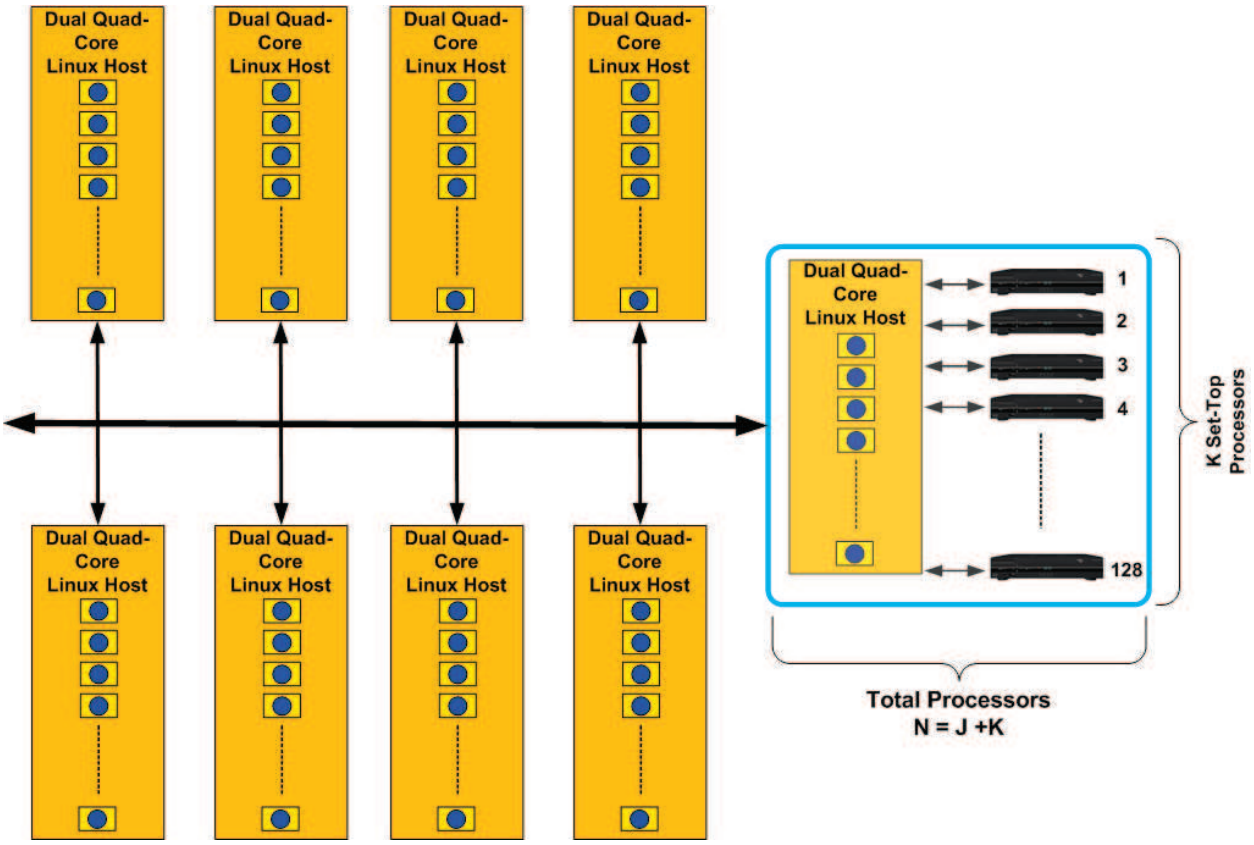


Figure 5.5: Virtualization of the STB embedded processors.

Second, the executions of the Open MPI processes running on the Embedded Cluster are mapped into the Linux Cluster. This was accomplished by implementing those software components which are necessary to support the mapping of the runtime and lifecycle management for these Open MPI processes into the standard Open MPI runtime software environment running on the Linux Cluster. Embedded Processor virtualization is illustrated in Fig. 5.4. Fig. 5.5 illustrates the high-level architecture of the resulting implementation in the context of a heterogenous cluster. On the right-end side, K embedded processors are mapped into a Linux host, which contains J processors. From the external viewpoint of other Linux nodes in the Linux Cluster, the host system becomes a heterogeneous multi-processor system with a total of $N = J + K$ processing cores. The host system is virtualized in the sense that the Linux Cluster nodes are unaware of the Embedded Cluster and simply view the virtualized host as a single N-processor Open MPI compute node. As a result, the

overall heterogeneous system equipped with K additional virtualized embedded processors may be utilized in any of three possible configurations in the experimental system:

1. A 129 node heterogeneous cluster consisting of a single Linux master node plus 128 Embedded Cluster processors.

2. A Linux Cluster consisting of the single Linux master node and eight Linux compute nodes.

3. A heterogeneous cluster consisting of the Linux master node, the eight Linux Cluster nodes, and the 128 Embedded Cluster nodes.

In order to complete the embedded-processor virtualization, the STB process runtime management environment is integrated into the OPEN MPI process runtime environment by implementing the OERTE framework described next. The OERTE framework provides protocol transformation and adaptation between the two heterogeneous runtime environments. Specifically, a new version of the OPEN MPI ORTE called *Open Embedded Runtime Environment (OERTE)* was developed. It consists of four major components: 1) a new ODLS module, 2) an OERTE server, 3) an OERTE embedded client that runs on embedded STB devices, and 4) Open MPI Embedded, an optimized Open MPI library for the resource-constrained embedded STB devices.

The New ODLS Module. As shown in Fig. 5.3, the original ORTE contains a ODLS module, which is responsible for: the launch/termination of an OPEN MPI process on the local host, various signaling, and the managing of modex-entry communications between the launched process and the ORTED during the initial phases of its execution. In particular, to manage a local process the ODLS external interface contract defines four *primary functions*: the process launch is initiated with `Launch_local_procs`, which specifies how many processes (along with their input arguments) must start on the computer host; abnormal termination is obtained by sending `Kill_local_procs` to all executing processes; Linux process signals, such as `SIGSTOP` are passed to all executing MPI processes with `Signal_local_procs`; and, finally, the modex data-exchange operations are performed with `Deliver_message`.

In this implementation we replace the standard ODLS module with a *New ODLS Module* at runtime, as shown in Fig. 5.6. The new module implements the primary functions by forwarding all requests over a TCP/IP socket interface to a new external *OERTE server*. This is a component

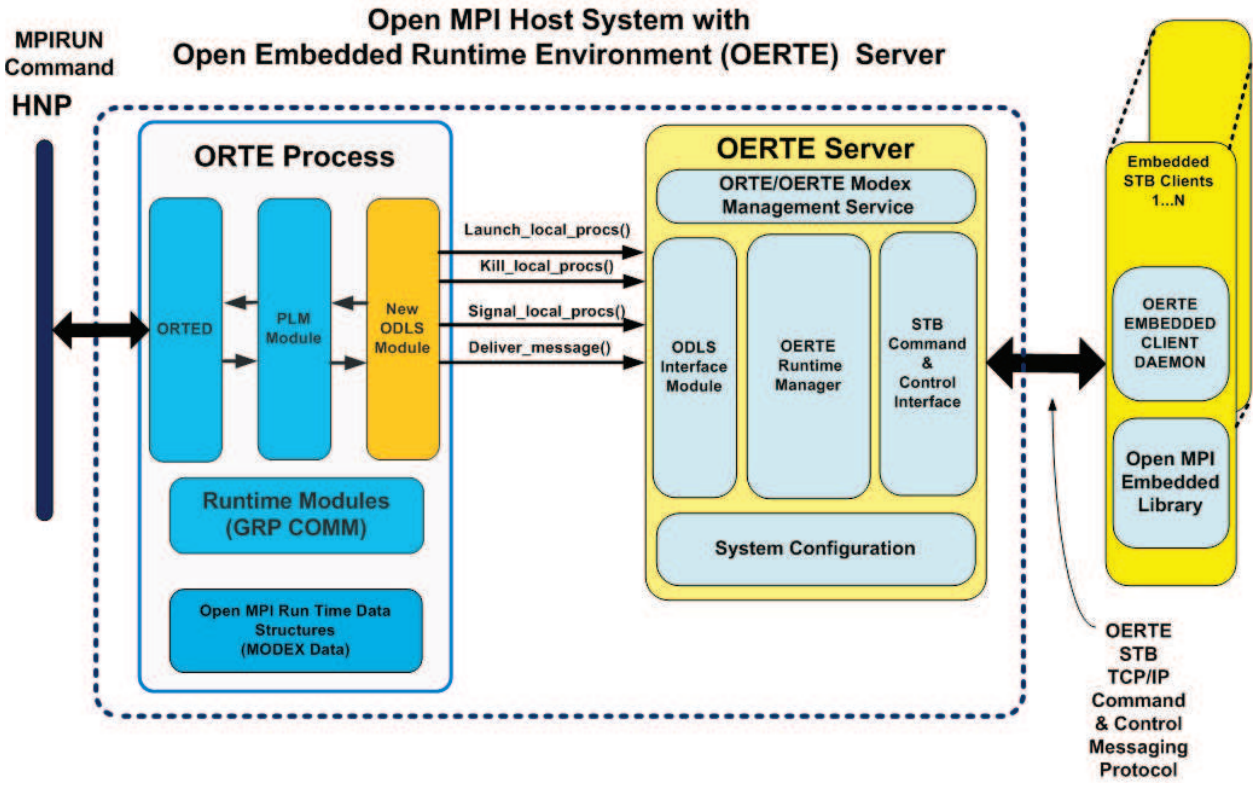


Figure 5.6: The OERTE server architecture.

which performs the ODLS functions in the context of the distributed embedded system environment and returns all responses in a manner that is equivalent to the original ODLS module function implementation as if these functions were executed locally. Response messages are sent from the OERTE server to the New ODLS Module over a second TCP/IP socket. In this manner the New ODLS Module provides a bi-directional bridge between the standard ORTE environment operating on the Linux Cluster and the external OERTE server, which is optimized to manage the Embedded Cluster runtime environment.

Open Embedded Runtime Environment (OERTE) Server. The OERTE server is a stand-alone multi-threaded Java server that acts as a runtime management server for the Embedded Cluster. It executes alongside the ORTE process and transforms ODLS function calls into operations that can be executed on the Embedded Cluster. As shown in Fig. 5.6, the OERTE server architecture includes an ODLS Interface module, which listens on the sender socket for the command functions from the New ODLS Module. As these are received, they are converted into

an internal Java object representation that is passed to the Runtime Manager. This works in conjunction with the Command & Control Interface module to coordinate the sequencing of message deliveries over TCP/IP to the client daemons running on the Embedded Cluster nodes. Responses from these processes are handled in a similar way: a client initiates a TCP/IP connection to the server and delivers response messages to the Command & Control Interface; these are converted back by the Runtime Manager to the appropriate ODLS-response format for transmission to the New ODLS Module for further processing before returning to the PLM module and finally the ORTED process.

The OERTE server has also a subsystem for generation and processing of modex data, which contains all the Internet address and port number information associated with the Linux Cluster and Embedded Cluster devices. This information is required by MPI processes to communicate with one another during execution of point-to-point or collective communication operations. The modex data is shared with all MPI processes through a modex-exchange operation that is coordinated among all cluster ORTED processes and the Host Node Process (HNP). Recall the HNP is the host system that initiates all MPI processes across a computational cluster when the mpirun command is executed.

In order for the Embedded Cluster to inter-operate with the Linux Cluster, the OERTE server must provide Embedded Cluster modex data to the ORTED process running on the same host in a manner that is transparent to all other Linux Cluster hosts as well as the HNP. This is achieved as follows. First, during OERTE server initialization, a configuration file that defines the Embedded Cluster network parameters is loaded into an in-memory data-structure buffer which is modeled after the original modex data structure. When the modex data associated to all the virtual processes (i.e. the processes running on the embedded devices) is requested by the ORTED process, the New ODLS Module makes a request to the OERTE server, which simply returns the information loaded in the data-structure buffer. This is then forwarded to the ORTED and delivered to the HNP, which aggregates all modex data from all hosts running ORTED in the heterogeneous cluster. The modex exchange process is completed when the HNP sends all the data from all nodes in both the Linux Cluster and the Embedded Cluster to each ORTED process. In the case of the Embedded Cluster, however, this information is delivered to the OERTE server, which stores it in a cache memory where it can be accessed by the processes running on the STBs through an on-demand caching

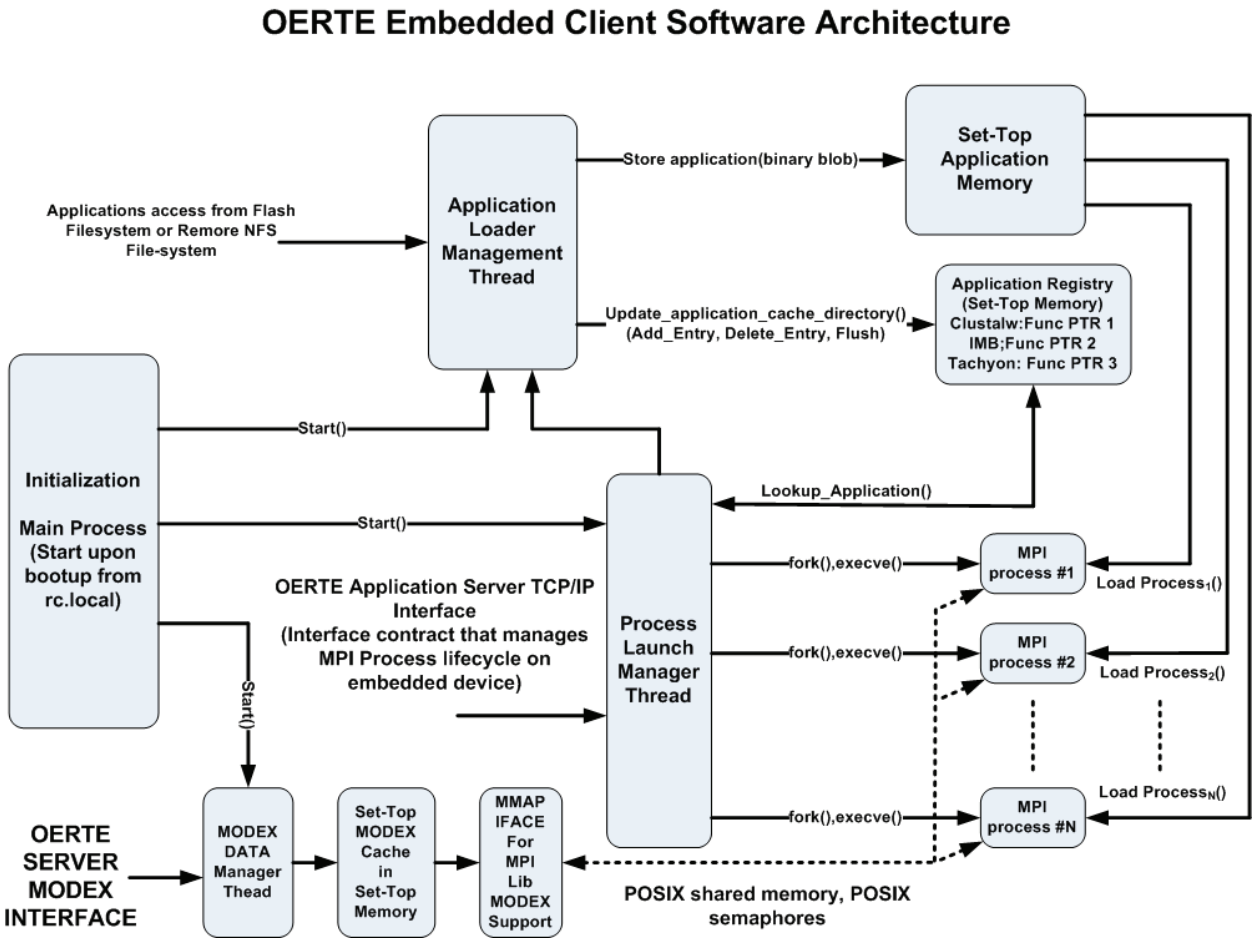


Figure 5.7: The software architecture of the OERTE embedded client.

mechanism. This mechanism is implemented in the OERTE embedded client process running on the STBs.

OERTE Embedded Client. To complete the OERTE virtualization framework, the OERTE embedded client was developed as a complete new replacement of the Open MPI ORTE module optimized for embedded devices. It consists of an application-client daemon that runs as a background process on the embedded Linux operating system. It is responsible for accessing MPI applications locally or from remote services, such as NFS-mounted file systems, and provides the runtime execution environment on the STB. This includes a number of functions: process launch, delivery of modex data and process signals, re-direction of I/O from the executing MPI application

to the OERTE server, and process termination. With an approximate size of 32KB, it utilizes a relatively small amount of the STB memory resources.

Fig. 5.7 shows the software architecture of the OERTE embedded client. The main process is started at STB boot-time from a standard Linux `rc.local` boot script and executes in the background continuously waiting for command and control signalling messages from the OERTE server. This main process forks a sequence of three threads which are executed asynchronously by all sub-systems: 1) a Process Launch Manager; 2) an Application Loader; and 3) a Modex Data Manager.

The Process Launch Manager is responsible for handling command and control protocol communications with the OERTE server. It manages the MPI application lifecycle by processing commands from the OERTE server to launch, deliver signals to, and terminate applications. It also accesses the Open MPI application from the STB-resident memory (as stored by the Application Loader) and starts each MPI application by executing a fork and a Linux system call. All applications are child processes of the Process Launch Manager, which may deliver Linux signals and can support redirection of application I/O as required. The Application Loader thread is responsible for accessing and bringing into the STB memory the intended MPI application as determined by the Process Launch Manager thread. The specific method for loading an application is abstracted within the Application Loader, e.g. applications can be accessed independently and concurrently from the STB local Flash memory or retrieved from a remote file system. As future delivery methods become available the Application Loader can be extended. Finally, a separate Modex Data Manager thread manages a small in-memory cache to support the Open MPI application modex-data look-up in coordination with the OERTE as shown in Fig. 5.6.

Open MPI Embedded Library. An important contribution of this work is the development of an optimized, reduced-footprint version of the Open MPI library to minimize the use of STB memory resources following an approach similar to the one used by McMahon *et al.* for the mpich mpi software distribution [91]. Specifically, we removed those frameworks and modules which are not applicable in the STB environment by modifying the Linux build tools appropriately. For example, we removed the Byte-Transfer-Layer (BTL) modules which support delivery of messages over one or more network interfaces other than TCP, such as Infiniband or shared memory. Similarly, we removed modules for parallel I/O and vendor-specific debugging modules. Finally, thanks to

the embedded processor virtualization discussed above, we could remove also most of the ORTE subsystem. In terms of size reduction, our optimized embedded version is about 32% smaller than the full Open MPI library, which is about 3.1MB. While this represent a significant memory saving for the current STB generation, as the memory capacity of embedded systems continues to grow we expect that this optimization will become less necessary.

In this implementation the ORTE framework was significantly reduced since the embedded processor runtime environment is no longer based on it. Indeed, we replaced ORTE with two new modules. First, the OERTE embedded client replaced the original ORTED. Second, the ORTE module for group communications (`grpcomm`) was refactored to support cache-based modex-lookup operations of the OERTE client instead of the ORTE collective operation method. This is a key optimization to overcome the scaling challenge (discussed in Section 5.3) of deploying Open MPI applications on a large-scale distributed embedded system. Whereas the standard ORTE implementation relies on a group collective `allgather` operation to exchange modex data among all processes resulting in a memory storage requirements of the order of $O(N * P)$, in our implementation the embedded client requires $O(1)$ memory thanks to the fixed size modex-caching subsystem. In this subsystem, all modex look-up operations are made to a local cache whose entries are co-managed by the OERTE server and embedded client processes.

5.5 Experimental Results

To validate the prototype system as well as the new Embedded Open MPI implementation OERTE server and client system, three sets of experiments with different workloads taken from MPI benchmarks, bioinformatics and Ray Tracing were completed. The goal of the experiments with the IMB benchmark suite is to demonstrate that the virtualization approach enables the interoperability between a Linux Cluster and Embedded Cluster to run any Open MPI application. The goal of the experiments with Ray Tracing and MSA is to evaluate the scaling potential of the system as a parallel execution platform.

5.5.1 Experimental Setup

Recall from Fig. 5.1 that the Linux Cluster consists of 9 Linux blades with dual 2Ghz quad-core Xeon processors (one blade acting as master host) while the Embedded Cluster consists of 128 Samsung SMT-C5320 STBs each with a single dual-core 400MHz Broadcom processor. In our experiments, we executed each workload test using all 8 blades plus the master on the Linux Cluster and all 128 embedded STBs on the Embedded Cluster. For each experiment on the Linux Cluster each workload is repeatedly executed by scaling the number of MPI processes (from 8, through 16 and 32, to 64) while evenly distributing them across all Xeon cores. On the Embedded Cluster, each workload is repeatedly executed by scaling the number of MPI processes (from 8 to 128) and distributing them with a one-to-one mapping on the 128 STBs (i.e. each STB runs at most one MPI process).

5.5.2 IMB MPI Benchmarks

The IMB benchmark suite developed by Intel consists of three parts (IMB-MPI1, IMB-MPI2, and IMB-IO) and provides an efficient way to measure the performance of the main MPI functions [19]. IMB enables the measurement of collective communications performance of a distributed computing cluster based on MPI. The following IMB-MPI1 benchmarks were executed which allows testing the important single-transfer, parallel-transfer, and collective communication operations: `ping-pong`, `send-recv`, `exchange`, `allreduce`, `reduce`, `reduce-scatter`, `allgather`, `gather`, `scatter`, `bcast`, `alltoall`, `barrier`. In particular, `ping-pong`, measures startup latency and throughput for a single-transfer message exchange between two processes. Parallel-transfer benchmarks, such as `send-recv` and `exchange`, measure the throughput of concurrent messages sent or received by a particular process in a periodic chain. The collective benchmarks measure the time needed to communicate among a group of processes in different patterns. We run each benchmark with various message sizes (in bytes): 64, 1K, 8K, 32K, 128K, and 256K. The tests were executed on both the Linux Cluster and the Embedded Cluster varying the number of MPI processes from 8 to 64 across all 8 nodes in the case of the Linux Cluster and 8 to 128 MPI processes across all 128 STB devices in the case of the Linux Cluster. The following experiments were executed from the IMB benchmarks listed in Table 5.1.

Each bar diagram in Fig. 5.8 shows the execution time as function of the number of processors

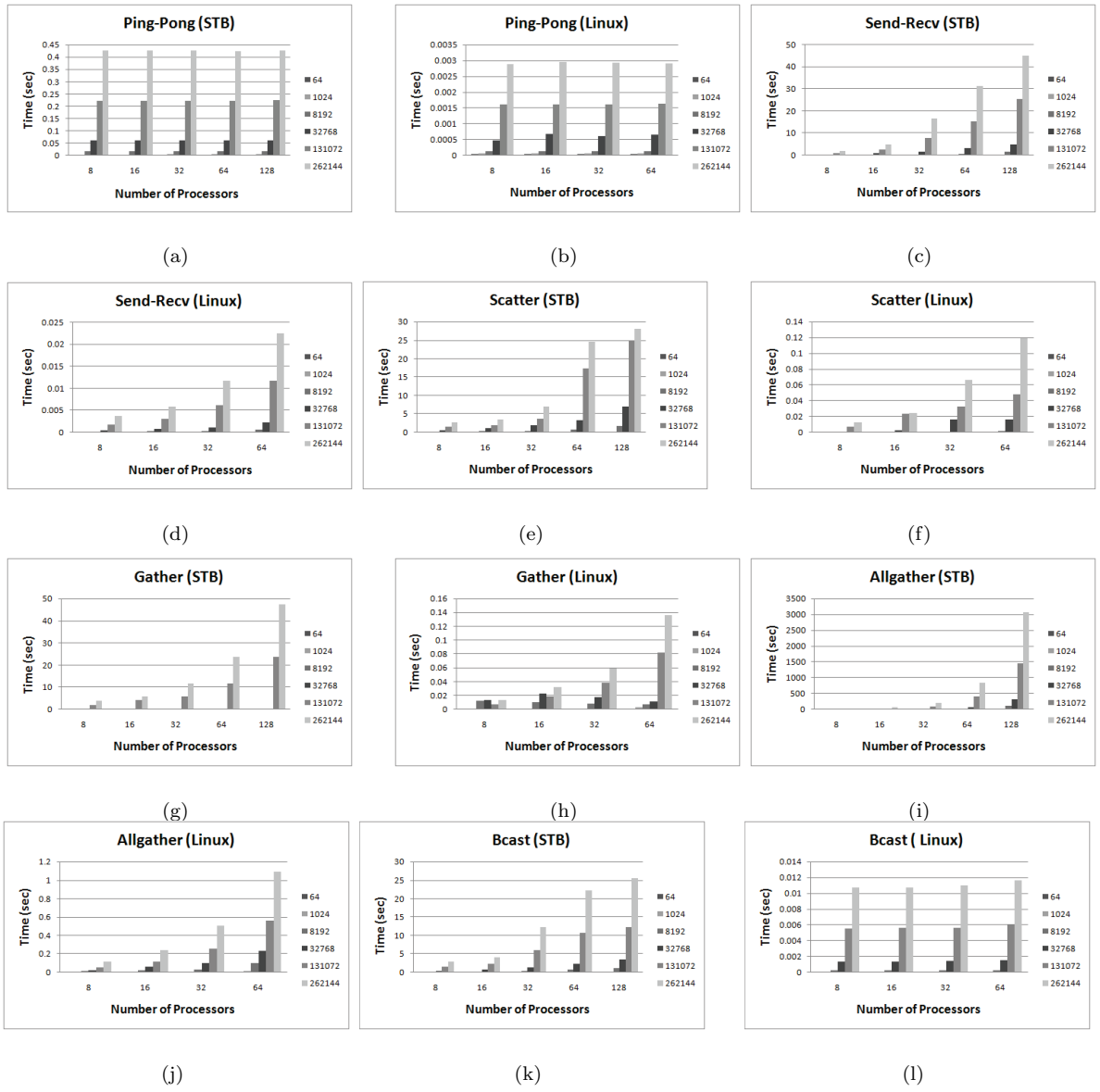


Figure 5.8: IMB test results.

and message size. For each test on both clusters, as we increase the number of processors the execution time increases, except for `ping-pong` and `bcast` on the Linux Cluster where it depends only on the message size. In `ping-pong`, the communication involves only two computer nodes and the average execution times over all message sizes are 122ms and 0.9ms for the Embedded Cluster and the Linux Cluster, respectively. This large difference is due to the performance gap between the Linux Cluster Gigabit Ethernet network and Embedded Cluster DOCSIS network. In fact, the

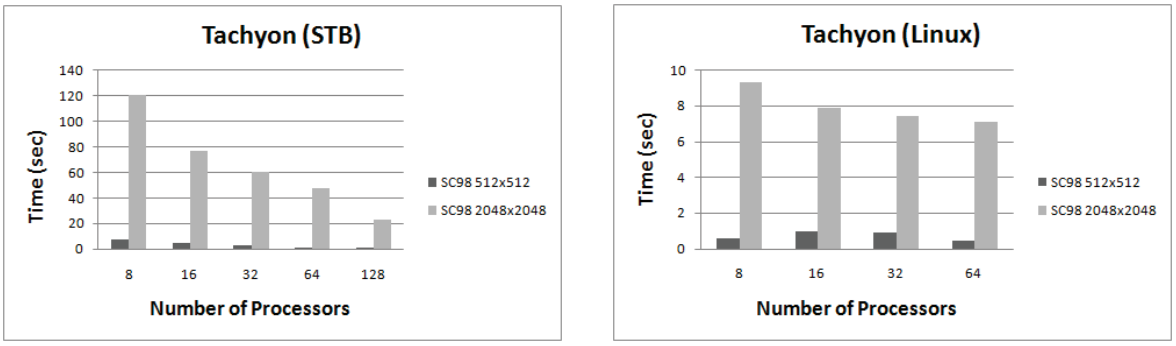
ping-pong	send-recv	exchange
allreduce	reduce	reduce-scatter
allgather	gather	scatter
alltoall	bcast	barrier

Table 5.1: IMB Experiments.

reported bandwidth for this benchmark averages between 45MB/s and 50MB/s for Ethernet but only 0.39MB/s for DOCSIS. The reported execution time of `bcast` (a broadcast communication test) on the Linux Cluster remains approximately constant as we vary the number of computer nodes (averaging 3.7s across all message sizes) and it increases only as we increase the message size: for 64B messages the average execution time is 0.07ms while for larger 256KB message becomes 10ms. In contrast, the execution times of `bcast` for the Embedded Cluster increase as we increase both the number of nodes (from 840ms for 8 STBs to 7.1s for 128 STBs) and the message size (from 200ms for 64B messages to 13.4s for 256KB message, averaged across all node counts). The difference in performance and the sensitivity to the node number between the Linux Cluster and Embedded Cluster are due not only to the lower bandwidth of DOCSIS but also to the highly-optimized implementations of this Open MPI collective operations, which the Linux Cluster nodes can access.

For the rest of the results of Fig. 5.8, the execution time increases directly proportional to the number of nodes and the message sizes. In all cases, it is far less on the Linux Cluster than on the Embedded Cluster (approximately by a factor of 100) and the reasons are similar as above: first, Gigabit Ethernet offers 100 times the bandwidth in comparison to DOCSIS (and significant higher for communications between the Xeon cores that are on the same chip); second, the Linux Cluster can run implementations of such collective Open MPI operations as `gather`, `scatter`, and `allgather` that are highly optimized.

Besides these facts, however, the important conclusions of these experiments are: (1) the validation that Embedded Open MPI Implementation running on the Embedded Cluster can execute correctly all IMB MPI benchmarks and (2) the demonstration that the performance of the collective MPI operations scales consistently across both the Linux Cluster and the Embedded Cluster environments. The next sets of results show how for those Open MPI applications that do not benefit for high-performance implementations of collective operations the performance gap between the



(a) (b)

Figure 5.9: Ray-Tracing results: (a) Embedded Cluster; (b) Linux Cluster.

two clusters is much smaller and decreases with the scaling of the parallel applications and cluster size.

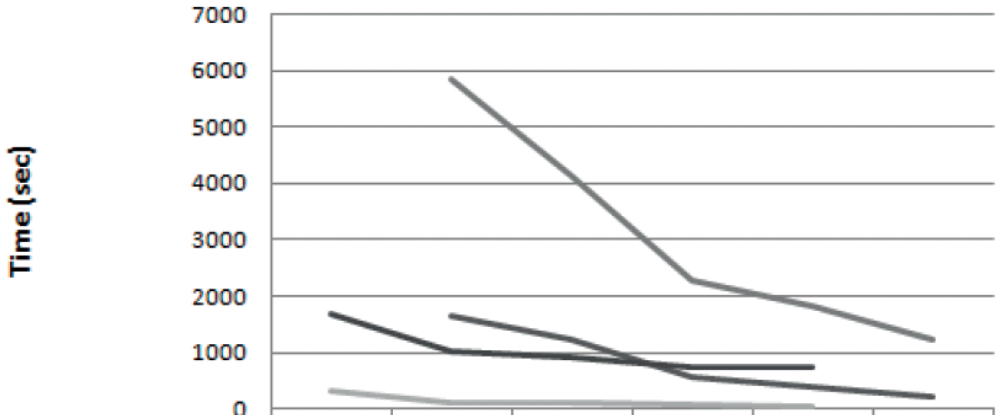
5.5.3 Parallel Ray Tracing

The TACHYON Ray-Tracer is a parallel application workload to evaluate the scaling performance of our system on a parallel image-processing application. Ray tracers are used to render scene images in games, 3-D modeling/visualization, and virtual reality applications [86]. They are well suited for parallelization thanks to their high data parallelism; each pixel in the rendered image can be processed independently and, therefore, different pixels can be assigned by the master host to different computer nodes [111]. The workload is embarrassingly parallel, where a single master host partitions the work associated to rendering an image into pixel blocks that are then assigned to multiple worker processing nodes that do not require any communications with one another, other than the master host. The TACHYON Ray-Tracer renders an image by using a scene description library of primitives as input data, which it parses into an internal scene object database. The scene description input-data contains constructs that define the location and viewing direction of camera and light sources; the locations, shapes, and various types of different objects such as polygons, spheres, cylinders, boxes, and triangles making up a scene [86]. Other parameters such as image resolution can also be specified.

The Tachyon rendering process constructs an output image by decomposing the full image viewing plane, or grid of pixels. into blocks. They are then assigned to computer nodes for ren-

dering, along with the scene object database which is fully replicated on each processor node. The rendering computation occurs in parallel as each processing node operates on a different assigned block of pixels. For each processing node, its assigned block of pixels are colored based on the light sources and objects defined in its local copy of the scene data-base. Each computer node has a local copy of the scene database. The ray-tracing process on each node proceeds as follows. To compute the correct color for each pixel, a ray is shot from the camera through the viewing plan into the scene [86]. The ray is then checked against the scene data-base list of objects to determine the first that it intersects. Next, the light sources are checked to see if any of the light rays reaches that intersection. If so, the color to be reflected is calculated based on the color of the object and the color of the light source. The resulting color is assigned to the pixel where the camera ray and the viewing plane intersect. This process is iterated until all pixels in the assigned image block are processed. Each processor node sends its assigned block of ray-traced image to the master host where the final scene output file is generated. In our experimental setup, we used a scene input file (SC98) from the TACHYON distribution and rendered it in two resolutions (512x512 and 2048x2048) to account for two different computational complexities. Similar to our other experiments, we executed the Tachyon application on the Linux Cluster and Embedded Cluster increasing the number of MPI processes across available computer or embedded device nodes. On the Linux Cluster we execute across all 8 Linux hosts, plus the master host, increasing the number of MPI processes from 8, 16, 32, and 64 in each experimental iteration. We execute the experiments in a similar manner on the Embedded Cluster, however we increase the number of MPI processes to 128 utilizing all 128 embedded STB devices.

As shown in Fig. 5.9, both clusters exhibit improved performance as the number of MPI processes grows. For the high-resolution case, as we increase the number of nodes from 8 to 64 the execution time improves from 9.3s to 7.07s (a speedup of 1.3) on the Linux Cluster and from 120.7s to 47.9s on the Embedded Cluster (thus resulting in a higher speedup of 2.5, which becomes 5.3 with 128 STBs). For the 512x512 resolution, as we go from 8 to 64 nodes, the execution time goes from 0.58s down to 0.44s on the Linux Cluster and from 6.9s to 1.27s on the Embedded Cluster (and down to 1.01s with 128 STBs). For the Linux Cluster the performance gain is flat and the execution time is I/O bounded, thus resulting in overall speedup of just 1.3, with the parallel portion of the application having a speedup of 4.3. Instead, the Embedded Cluster overall speedup is



	2	8	16	32	64	128
S500-L200-STB		1652.449	1235.773	585.511	380.853	215.933
S500-L200-Linux	341.72	129.708	102.365	64.862	50.856	
S1500-L100-STB		5834.074	4128.24	2283.493	1841.269	1246.293
S1500-L100-Linux	1687.018	1020.148	935.435	755.346	729.744	

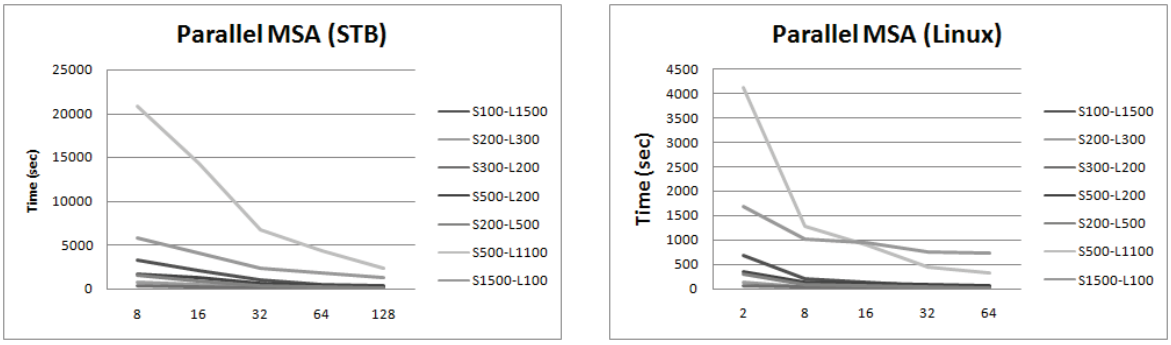
Figure 5.10: Impact of sequence size/length on performance scaling.

Processor		Overall Execution Time (Sec)							
Type	#	S100-L1500	S200-L300	S300-L200	S500-L200	S200-L500	S500-L1100	S1500-L100	Avg.
Linux	1	692	129	62	341	289	4130	1687	
	64	43	15	8	51	21	326	730	
	Speedup	**16.1**	**8.6**	**8.1**	**6.7**	**13.7**	**12.7**	**2.3**	**9.7**
C5320 STB	8	3257	693	319	1652	1522	20854	5834	
	128	391	113	44	216	154	2340	1246	
	Speedup	**8.3**	**6.1**	**7.2**	**7.7**	**9.8**	**8.9**	**4.7**	**7.3**

Table 5.2: Experiments with Multiple Sequence Alignment: Overall execution times and speedups of each cluster.

5.5 (6.9 with 128 STBs), with its parallel portion having a speedup of 7.7 (14.9 with 128 STBs).

In summary, the Embedded Cluster benefits more than the Linux Cluster when the ray-tracer image computation is large, requiring more pixel calculations computed in a data-parallel model. Instead, the Linux Cluster is penalized when the ray-tracer image computation is small, as more time is spent performing I/O relative to computation. This effect is illustrated in Fig. 5.9(b) where the Linux Cluster execution time for the 512x512 case remains flat as we increase the number of processing nodes. In contrast, Fig. 5.9(a) shows that for the 2048x2048 case, the Embedded Cluster exhibits linear scaling as we increase the number of STBs.



(a) (b)

Figure 5.11: MSA execution time across all benchmarks: (a) Embedded Cluster; (b) Linux Cluster.

5.5.4 Multiple Sequence Alignment

This is the fundamental problem in bioinformatics that was presented in detail in Section 4.5: instances of DNA or protein sequences must be optimally aligned so that the highest possible number of their elements match. Given a scoring scheme to evaluate this matching and penalize the presence of sequence gaps, to solve MSA consists in placing gaps in each sequence to maximize the alignment score [26]. Since MSA is an NP-hard problem, an approximation algorithms such as ClustalW is typically used [114]. We run the MASON parallel ClustalW implementation using MPI [26; 32], which proceeds as follows: the master host partitions the N input sequences among P worker processors; these perform pair-wise alignment on their set of sequences in parallel; alignment scores are sent back to the master which constructs the guide tree and distributes the computed guide order along with associated sequences where the workers then compute a partial MSA; finally, the workers send their partial multiple alignments back to the master, which performs the final stage of progressive alignment. We used ROSE, a tool that produces synthetic sets of DNA sequences which follow an evolutionary model [112], to generate 7 sequences of various length and base pair. In the sequel, the encoding *Sx-Ly* denotes a data set of x DNA sequences with y base pairs (e.g. *S100-L1500* means "100 DNA Sequences with 1500 base pairs").

Multiple Sequence Alignment Parallelization Speedup. Fig. 5.11 shows the results of executing the parallel MSA application on the Embedded Cluster and Linux Cluster. Again, as the number of MPI processes increase from 2 to 64 across the Linux Cluster nodes, and 8 to 128 across the Embedded Cluster nodes, the execution time decreases consistently for all the seven synthetic

test sequences.

Table 5.2 reports the overall execution times for all sequences for two particular configurations of each cluster (1 and 64 processors for the Linux Cluster, and 8 and 64 STBs for the Embedded Cluster) together with the relative speedups. In most cases the Linux Cluster has higher speedup. This is expected given the benefits of high-performance Xeon processors and Gigabit Ethernet (as verified with the IMB benchmarks). In two cases (sequences S500-L200, S1500-L100), however, the Embedded Cluster outperforms the Linux Cluster in terms of speedup: as shown in Fig. 5.10, it exhibits higher relative performance gains as we increase the number of sequences, when the average sequence length is relatively short. This is due to the data-parallelism portion (aligning partitioned $\frac{N \cdot (N-1)}{2}$ sequence permutations independently) of the MSA algorithm, which benefits a cluster with a large number of nodes. In contrast, the Linux Cluster gives higher performance gains for longer sequences as their processing requirements has complexity bounded by $O(n^2)$, thus favoring the higher performance blade servers when the number of sequences is small compared to their length. However, Fig. 5.10 shows that in moving from 64 to 128 STBs the Embedded Cluster actually manages to complete the application execution in a time that is shorter that the time taken by the Linux Cluster with only two blades.[1] This suggests that an Embedded Cluster with sufficient processing nodes is suited for a wider range of data-intensive, parallel applications where very large data-sets must be processed.

Discussion. As we evaluate these experimental results (and particularly the IMB ones), if we factor out any performance gain advantage of the Embedded Cluster due to the data-parallelism of the workloads, it is clear that differences in network performance have a significant impact on the overall execution time. As we look to further improve our system, a number of factors including physical network and software architecture must be considered to reduce these differences. In terms of physical network, the Embedded Cluster system was tested on a lab environment (where contention exists) that did not include the use of QoS parameters to manage bandwidth allocations. The DOCSIS standard has facilities for managing and prioritizing bandwidth on a per device or

[1] As one considers the significance of obtaining the same performance of two blade servers with a cluster of over 64 STBs, it should be kept in mind that each blade features 64-bit processors running at a clock frequency which is five times higher than the clock frequency of the 32-bit processor of the STB! In other words, for certain classes of Open MPI applications the gap between the two cluster is being reduced.

application basis. While this may reduce contention, the physical network is still bounded by a maximum upstream and downstream bandwidth of 27Mb/s. To overcome this limitation, future DOCSIS 3.0 networks will use channel bonding to obtain higher bandwidth up to 340Mb/s. Further performance improvements are possible for certain communication operations that make natural use of broadcast and multicasting techniques.

Trends in STB Hardware and Software. As discussed in Chapter 2, the performance of embedded devices continues to improve with each new generation. We expect this trend to continue as consumer-driven applications and services are becoming ubiquitous across multiple screens, both mobile and at home. For instance, the Apple iPad contains more computing power than most workstations of just a few years ago. The large commoditization and shrinking performance gap between the most powerful processors and the embedded consumer-electronic devices is a powerful force behind the concept of broadband embedded computing that is envisioned in this dissertation. Future generations of our system will utilize these new emerging devices while continuing to improve the communication architecture and targeting new applications and workloads.

STB cost considerations. STB costs continue to drop, however, at $100 to $500 per-device, represent a significant expense to cable system operators. Approximately 10% to 15% of STB devices are replaced each year with new devices, due to failures or damage. This ongoing replacement results in a continuous progression of STB device capabilties and performance. Additionally, cable operators recoup STB capital outlays through regulated leasing of equipment to subscribers. In this manner, broadband operators continue to operate and manage both STB devices and their broadband network while economically distributing large numbers of STBs to subsribers.

5.6 Related Works

There are a number of recent efforts on the virtualization of mobile devices for grid computing [15; 76]. This work focuses on the utilization of virtualization to enable the transparent integration of embedded computing and *managed broadband networks* with data-center class server computing systems for distributed computation based on the message-passing model.

Related works such as LMPI and eMPI described methods for integrating message-passing middleware into embedded devices by reducing the size of the MPI stack [7; 91]. Similarly to LMPI

and eMPI, we used a top-down approach to minimize the MPI framework and eliminate unnecessary software modules, resulting in a reduced memory library foot-print. In this work, however, each STB contains a modern real-time Linux operating system that can support a complete native MPI implementation enabling the elimination of unnecessary software modules while still maximizing functionality. Also, while previous work in executing MPI on embedded devices has focused on small test kernels, we can run complete application that use of a rich set of MPI operations, including collectives.

Google designed and deployed a massively-parallel system comprised of commodity dual-core PCs running Linux combined with its custom Map-Reduce framework for parallel computing [28]. This platform is distributed across many data-centers and was estimated in size at over 450,000 systems [36]. A possible future large-scale version of our proposed architecture would have important differences with the Google platform, including the use of a broadband network of embedded devices instead of a network of clusters of PCs and the use of a hybrid MPI and Map-Reduce application model which is today an active area of research.

5.7 Summary

A second-generation heterogeneous distributed system architecture for broadband embedded computing using Open MPI that is fully interoperable and provides scalable process launch within the Open MPI ORTE runtime environment is presented. Key contributions in this work include a new method to integrate networks of embedded processors with computer clusters through a software virtualization framework for large-scale application process launching called Open Embedded Runtime Environment or OERTE. This enables embedded processors to transparently inter-operate with computer clusters using the message-passing model. The second-generation prototype is implemented and evaluated with three sets of experiments to validate its operations and scaling potential. The experimental results indicate that the performance of the system is impacted by the existing broadband DOCSIS network, which is not optimized for Open MPI collective operations. Future work is required in this area. Chapter 7 presents a solution to improving the communication performance of broadband embedded computing systems for certain broadcast classes of collective operations across broadband networks. The experimental work presented, however, does demon-

strate that the second-generation system is already capable of delivering significant performance gains for some classes of Open MPI applications. This suggests a wealth of opportunity in leveraging broadband embedded computing for heterogeneous Cloud system applications, especially those that are data-intensive or data parallel in nature.

Chapter 6

Broadband Embedded Computing System For MapReduce Utilizing Hadoop

6.1 Introduction

Chapters 4 and 5 described two broadband embedded computing framework implementations based around the Open MPI message passing model for distributed parallel computing, which is predominant within scientific and parallel computing application domains. However, as the growth in the amount of data created, distributed and consumed continues to expand at exponential rates, systems built to support the MapReduce programming model are seeing a surge in interest to address computational requirements surrounding the Big Data phenomenon.

According to a recent research report from the International Data Corporation, the amount of digital information created and replicated has exceeded the zettabyte barrier in 2010 and this trend is expected to continue to grow "as more and more embedded systems pump their bits into the digital cosmos" [22]. In recent years the MapReduce framework has emerged as one of the most widely used parallel computing platforms for processing data on very large scales [82]. While MapReduce was originally developed at Google [28], open-source implementations such as Hadoop [51] are now gaining widespread acceptance. The ability to manage and process data-

intensive applications using MapReduce systems such as Hadoop has spurred research in server technologies and new forms of Cloud services such as those available from Yahoo, Google, and Amazon.

Meanwhile, the Information Technology industry is experiencing two major trends. On one hand, computation is moving away from traditional desktop and department-level computer centers towards an infrastructural core that consists of many large and distributed data centers with high-performance computer servers and data storage devices, virtualized and available as Cloud services. These large-scale centers provide all sorts of computational services to a multiplicity of peripheral clients, through various interconnection networks. On the other hand, the increasing majority of these clients consist of a growing variety of embedded devices, such as smart phones, tablet computers, and television set-top boxes (STB), whose capabilities continue to improve while also providing data locality associated to data-intensive application processing of interest. Indeed, the massive scale of today's data creation explosion is closely aligned to distributed computational resources of the expanding universe of distributed embedded systems and devices. Multiple Service Operators (MSOs), such as cable providers, are an example of companies that drive both the rapid growth and evolution of large-scale computational systems, consumer and business data, as well as the deployment of an increasing number of increasingly-powerful embedded processors.

This ongoing work in developing platforms for broadband embedded computation is motivated precisely by the idea that ubiquitous adoption by consumers of embedded devices and the combination of the technology trends in embedded systems, data centers, and broadband networks opens the way to a new class of heterogeneous Cloud computing for processing data-intensive applications. In particular, in this chapter I present an implementation of *broadband embedded computing system for MapReduce utilizing Hadoop* as an example of one such systems. Its potential application domains include: ubiquitous social networking computing, large-scale data mining and analytics, and even some types of high-performance computing for scientific data analysis. In particular, the chapter presents a heterogeneous distributed system architecture which combines a traditional cluster of Linux blade servers with a cluster of embedded processors interconnected through a broadband network to offer massive MapReduce data-intensive processing potential (and, potentially, energy and cost efficiency).

The implementation of the Hadoop MapReduce framework is presented as extensions to the

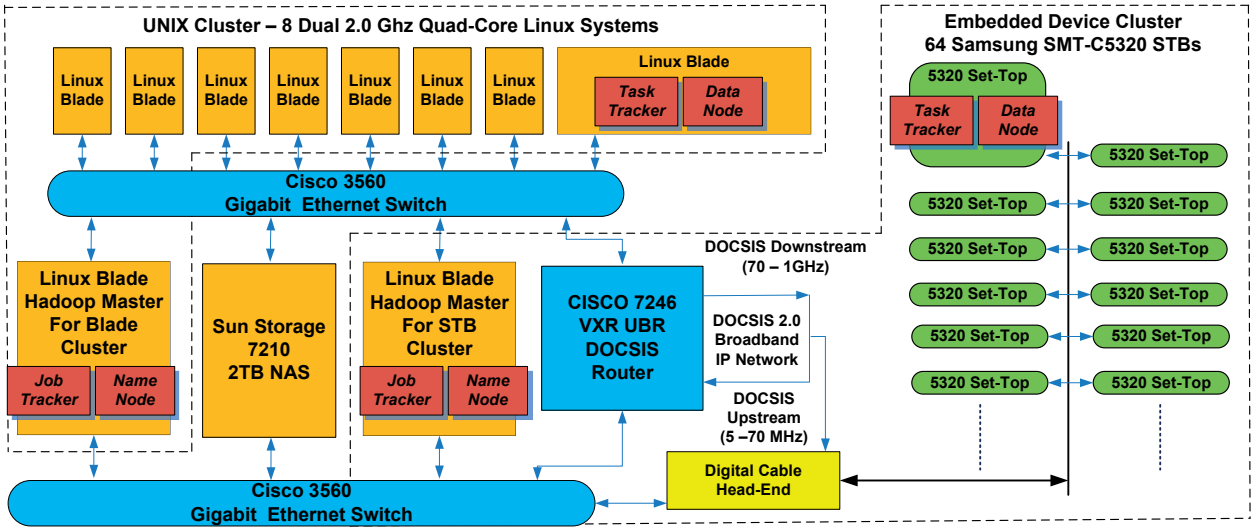


Figure 6.1: Architecture of the broadband embedded computing system for MapReduce utilizing Hadoop.

broadband embedded computing system architectures introduced in Chapters 4 and 5. As discussed in Section 6.3, this porting posed important challenges in terms of software portability and resource management. These challenges are addressed in two ways. First, porting techniques are developed for embedded devices that leverages back-porting of enterprise software in order to implement the Hadoop system for embedded environments. Second, to execute MapReduce applications on such resource-constrained embedded devices as STBs, both memory and storage requirements are optimized by eliminating unnecessary software components of the Hadoop platform. The result is an embedded version of the Hadoop framework.

Section 6.4 presents a set of experiments which confirm that the embedded system implementation of the Hadoop runtime environment and related software libraries runs successfully a variety of MapReduce benchmark applications. Also, in order to gain further insight into the relative performance scaling of the Set-Top Cluster versus the Linux Cluster while running MapReduce applications, the number of processing elements (which correspond to the number of Hadoop nodes) and the size of the input data are varied. Overall, the experimental results expose the Set-Top Cluster performance sensitivity to certain classes of MapReduce applications and indicate avenues of future research to improve our system.

6.2 The System Architecture

Fig. 6.1 provides an overview of the architecture of the system developed and built: this is a heterogeneous system that leverages a broadband network of embedded devices to execute MapReduce applications by utilizing Hadoop. It is composed of four main subsystems.

Linux Blade Cluster. The Linux Cluster consists of a traditional network of nine blade servers and a Network Attached Storage (NAS). Each blade has two quad-core 2GHz Xeon processors running Debian Linux with 32GB of memory and a 1Gb/s Ethernet interface. One of the nine blades is the Hadoop master host acting both as NameNode and JobTracker for the MapReduce runtime management [51]. Each of the other eight blades is a Hadoop slave node, acting both as DataNode and TaskTracker [51] while leveraging the combined computational power of the 8 processing cores integrated on the blade. The blades use the Network File System (NFS) to mount the 2TB Sun storage array, which provides a remote common file-system partition to store applications for each of the executing Hadoop MapReduce applications. For storing the Hadoop Distributed File System (HDFS) data, the blades use their own local hard-disk drive (HDD).

Embedded STB Cluster. The Set-Top Cluster consists of 64 Samsung SMT-C5320 set-top boxes (STB) that are connected with a radiofrequency (RF) network for data delivery using MPEG and DOCSIS transport mechanisms. The Samsung SMT-C5320 is an advanced (2010-generation) STB featuring an SoC with a Broadcom MIPS 4000 class processor, a floating-point unit, dedicated video and 2D/3D-graphics processors with OpenGL support, 256MB of system memory, 64MB internal Flash memory, 32GB of external Flash memory accessible through USB, and many network transport interfaces (DOCSIS 2.0, MPEG-2/4 and Ethernet). Indeed, an important architectural feature of modern STBs is the heterogeneous multi-core architecture design which allows the 400MHz MIPS processor, graphics/video processors, and network processors to operate in parallel over independent buses. Hence, user-interface applications (such as the electronic programming guides) can execute in parallel with any real-time video processing. From the viewpoint of running Hadoop applications as a slave node, however, each STB can leverage only the MIPS processor while acting both as DataNode and TaskTracker. [1] This is an important difference between the

[1] In the Set-Top Cluster, there is also a Linux blade which is the Hadoop master node, acting both as NameNode and JobTracker.

Set-Top Cluster and the Linux Cluster. Finally, in each STB, a 32GB USB memory stick is used for HDFS data storage, while NFS is used for Java class storage.

Network. The system network is a managed dedicated broadband network which is divided into three IP subnets to isolate the traffic between the DOCSIS-based broadband Set-Top Cluster network, the Linux Cluster network, and the digital cable head-end. Its implementation is based on two Cisco 3560 1Gb/s Ethernet switches and one Cisco 7246 DOCSIS broadband router. The upper switch in Fig. 5.1 interconnects the 8 blades along with the NAS and master host. The lower switch aggregates all the components on the head-end subnetwork. The DOCSIS subnetwork is utilized by the Set-Top Cluster whose traffic exists on both the Linux Cluster and the digital head-end network. The broadband router has 1Gb/s interfaces for interconnection to the Linux Cluster and head-end networks as well as a broadband interface for converting between the DOCSIS network and the Ethernet backbone. Each broadband router can support over 16,000 STBs, thus providing large-scale fan-out from the Linux Cluster to the Set-Top Cluster.

Embedded Middleware Stack. The embedded middleware stack is based on Tru2way, a standard platform deployed by major cable operators in U.S. as part of the Open Cable Application Platform (OCAP) developed in conjunction with Cablelabs [4]. Various services are delivered through the Tru2way platform including: chat, e-mails, electronic games, video on-demand (VOD), home shopping, interactive program guides, stock tickers, and, most importantly, web browsing [6]. To enable cable operators and other third-party developers to provide portable services, Tru2way includes middleware based on Java technology that is integrated into digital video recorders, STBs, TVs, and other media-related devices.

Tru2way is based on Java ME (Java Micro Edition) with CDC (Connected Device Configuration) designed for mobile and other embedded devices. The Tru2way standard follows FP (Foundation Profile) and PBP (Personal Basis Profile) including: io, lang, net, security, text, and util packages as well as awt, beans, and rmi packages, respectively. Additional packages include JavaTV for Xlet applications, JMF (Java Media Framework), which adds audio, video, and other time-based media functionalities, and MHP (Multimedia Home Platform), which comprises classes for interactive digital television applications. On top of these profiles, the OCAP API provides applications with Tru2way-specific classes related to hardware, media, and user-interface packages unique to cable-based broadband content-delivery systems.

Remark. While this rich set of Java profiles offer additional features to the embedded Java applications, there exists a significant gap between the Java stack provided by Tru2way and the Java Platform Standard Edition (Java SE), which is common to enterprise-class application development. Hence, since the standard Hadoop execution depends on the Java SE environment, we had to develop a new implementation of Hadoop specialized for the embedded software environment that characterizes devices such as STBs. We describe our effort in the next section.

6.3 Porting Hadoop to the Broadband Embedded System

There are several issues that need to be addressed in order to successfully run Hadoop on a distributed embedded systems like our broadband network of STB devices.

First, Hadoop and Hadoop third-party libraries require many bootstrap classes not supported by the Tru2way JVM. Also, for many classes the Tru2way JVM supports only a subset of methods: e.g., both Tru2way and Java SE have the `java.lang.System` class, but the `java.lang.System.getenv()` method exists only in Java SE.

Second, the Tru2way JVM only supports older versions of Java class file formats while Hadoop is developed using many Java 1.6 language features including: generics, enums, for-each loops, annotations, and variable arguments.

Third, the task of porting Java applications to another JVM with different profiles is quite challenging and, differently from porting native codes to JVM [13; 80], it has not been actively studied in the literature. If not an impossible task, to modify Hadoop and the Hadoop third-party libraries at the source code level is not really practical because there are more than fifty of such libraries and, in some cases, their source code is not available.

Finally, despite all the efforts to improve the JVM portability [102; 108], to port the Java SE JVM to the STB environment is very hard because these embedded devices do not support key features such as frame buffer or native implementations.

To address these challenges, we have developed a binary level porting method for embedded devices that imports missing class files and retro-translates all the class files so that the embedded Tru2way JVM can execute them. Our method leverages the Java Backport package, which is the implementation of JSR 166 (`java.util.concurrent` APIs), introduced in Java SE 5.0 and further

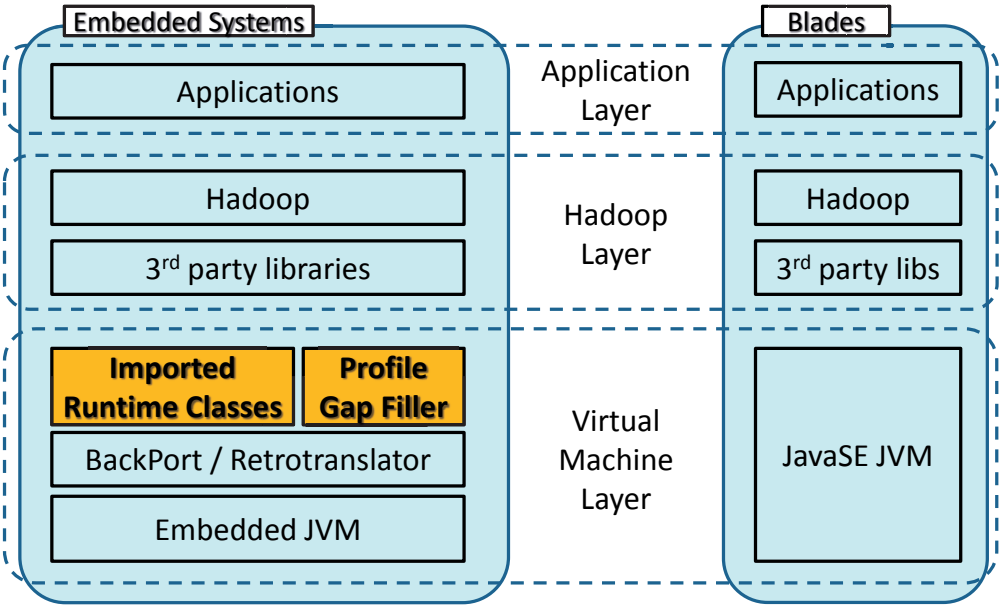


Figure 6.2: Two software stacks to support Hadoop: STB vs. Linux Blade.

refined in Java SE 6.0, for older versions of Java platforms [2]. The Retrotranslator has two main functionalities: 1) it translates newer class files into an older format for an older JVM; and, 2) it extends the Backport package so that most Java SE 5.0 features are available for an application that runs on the Java SE 1.4 and Java SE 1.3 JVMs [3]. The runtime classes from those two packages can be added to the Tru2way JVM.

Fig. 6.2 shows the resulting software stack to support the execution of Hadoop in the embedded environment of an STB running the Tru2way JVM and contrasts it with the traditional software stack based on the Java SE JVM running on a common Linux blade. In particular, the embedded software stack includes the *Imported Runtime Classes*, which are the results of the backporting technique, and the *Profile Gap Filler*, which collects all additional components that were developed specifically for the embedded STB devices.

Fig. 6.3 illustrates the procedure developed to port Hadoop and all the Java packages necessary for running Hadoop to the STB devices. *While it was developed and tested for our broadband embedded system, for the most part this procedure is a contribution of general applicability to port Java applications originally developed for the Java SE JVM to other embedded systems which have different and more limited JVMs:* e.g., this procedure can be followed also for porting any Java

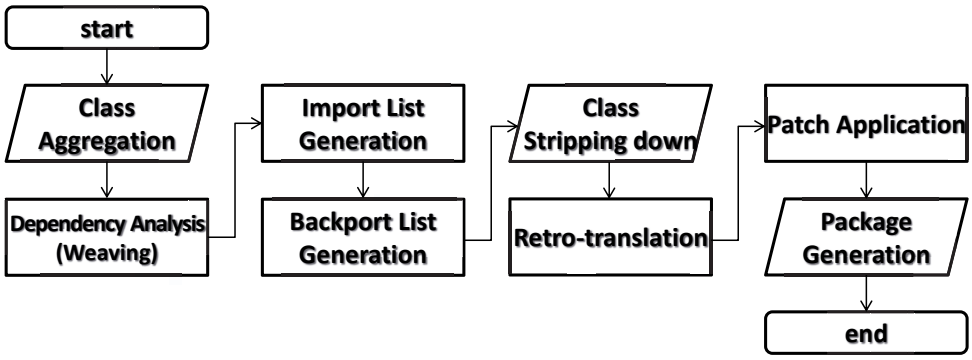


Figure 6.3: Porting the Hadoop-supporting Java classes to the STB devices.

applications to other JVM such as BD-J or Android's Dalvik [43; 93]. The procedure consists of a sequence of eight main steps:

1) Class Aggregation. Here all the input classes are simply copied into a single directory and the priorities among the duplicated or collided classes are determined.

2) Dependency Analysis. For this step, which is key to implementing efficiently a large Java application like Hadoop on resource-constrained embedded devices, we developed a novel dependency analysis technique called *Class Weaving*. This starts by analyzing the class dependencies within a Java package as well as across the packages and then changes the dependency to reuse as much as possible those classes which are available in the embedded Java ME environment. The goal is to generate all the information on class dependencies that is necessary at later steps to minimize the number of classes which will be imported from the various open-source Java SE runtime libraries (and to strip out all unnecessary classes from the original packages.) Fig. 6.4 illustrates how Class Weaving works: a class dependency tree is generated by analyzing each class while minimizing the number of classes to be imported. For example, Hadoop's `TaskTracker` class uses the `Pattern` class, which in turn uses the `Matcher` class: both these classes exist in Java SE but not in the STB Java ME environment and, therefore, need to be imported. On the other hand, the `Pattern` class uses the `Character` class, which exists also in Java ME and, therefore, it will not be imported from Java SE: instead, the `Pattern` class will be woven to use Java ME's `Character` class.

3) Import List Generation. Based on the information collected at the previous step, the list of classes to be imported is generated. At this step, the list can be refined through additional customizations. Unlike most JVMs, some embedded JVMs have their bootstrap classes embedded in a way that are not accessible to the application developers and provide only stub classes to

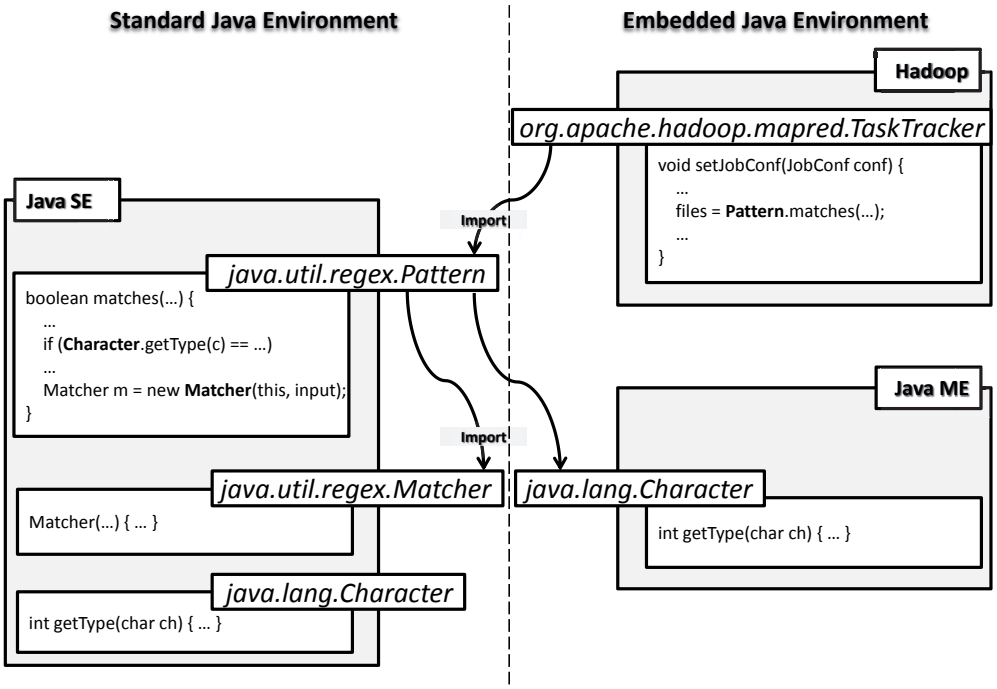


Figure 6.4: Example of applying the proposed Class Weaving method.

	Hadoop & 3rd-party libs	JavaSE Bootstrap
Before	14490	10110
After	4141	5978

Figure 6.5: Class count before & after class stripping optimization.

them. For instance, packages like `xerces` or `log4j` do exist in the actual bootstrap classes for internal purposes but are not included in the stub classes.

4) Backport List Generation. The Java class loaders check if the package name of the target class begins with the `'java.'` prefix when the class file location is not in the bootstrap classpaths and, if so, returns an error. To avoid this, the prefix needs to be changed: e.g., in the case of our system with the `'edu.columbia.cs.sld.backport.ocap.java.'` prefix. A list of the mappings between the original and the new prefix is generated for all the imported classes with package names that begin with `'java.'` to be used later in the retro-translation step.

5) Class Stripping Optimization. Since many embedded systems have limited memory and storage resources, only the necessary Java classes should be stored in the embedded device. This is achieved by collecting dependency trees that begin with the *seed classes*, which include the

entry point `Xlet` class that launches Hadoop DataNode and TaskTracker, various classes that are dynamically loaded from configuration files or from the source code, and the patched classes. In our case, this step results in a 60% reduction of the number of classes that must be deployed in the STBs, as shown in Fig. 6.5.

6) Retro-translation. Since the Tru2way JVM recognizes classes up to Major Version Number 48, all the class files with Major Version Number 49 or higher need to be retro-translated. Most packages, including Hadoop, provide classes with major version number 50, which corresponds to Java 1.6. At the binary level, the class file formats and package names of Hadoop, Java SE, and the application libraries need to be properly modified.

7) Patch Application. While a number of classes were imported from open-source Java SE runtime libraries through the Class Weaving technique described above, we had to newly develop a number of missing classes and methods which needed to be optimized before being added to the Java stack of the STBs. The same was necessary for classes that could not be imported from the open-source Java SE runtime library due to the native implementations. Also, patches were necessary to fix some defects found in the Tru2way implementations.

8) Package Generation. This final step generates the packages that will be launched on the Tru2way JVM from the stripped classes, links a custom class loader that will load user-defined `Mapper` and `Reducer` classes, and binds an entry point `Xlet` that will execute Hadoop DataNode and TaskTracker.

6.3.1 Challenges in Porting Hadoop to STB Devices

The number of JVM processes supported in the system is one of the biggest differences between the STB Java environment and a Linux blade server utilizing Java SE. While the users of the latter can launch multiple instances of JVM, only one JVM instance can be launched during boot time within an STB. On the other hand, there are two important behaviors in Hadoop that rely on the capability of multiple JVM executions: first, TaskTracker and DataNode are running two different JVM processes; second, for each task processed in a TaskTracker node, a new JVM instance is launched unless there is an idle JVM which can be reused for the task.

To support these behaviors while coping with the STB limitation of running only one JVM instance, we implemented a new `ProcessBuilder` class that creates a thread group whenever the

launch of a new JVM process is requested. Each thread group provides a distinct set of Hadoop environmental variables which are managed within the threads belonging to a given thread group without interfering with other threads groups. The `ProcessBuilder` class implementation also enables optimizations such as replacing IPC (Inter-Process Call) with method invocations in the same process, and the elimination of local data transfers through sockets with local file-copy operations.

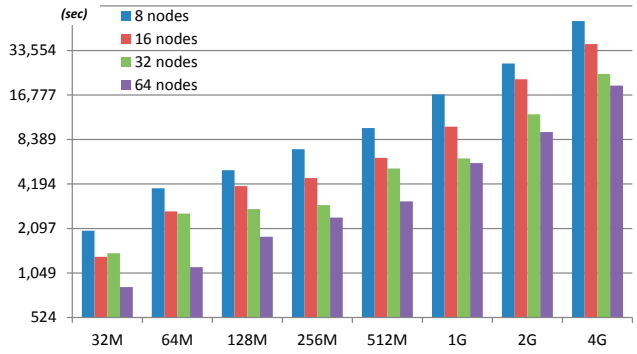
A number of other middleware issues related to porting Hadoop to an embedded device like the STB were discovered and resolved. For example, certain Java classes have bugs that make the application behave improperly, halt, or sometimes fail. In these cases the classes were replaced with better implementations or patched to align with the Hadoop Java class requirements. Also, some configuration changes were made to the system: e.g., the Socket timeout constant had to be slightly extended to account for variations in network response times or delays. Finally, to relieve memory constraints, we reduced the number of threads associated to unimportant services such as the metrics service which profiles the statistics of performance or the web service that provides status information.

6.4 Experiments

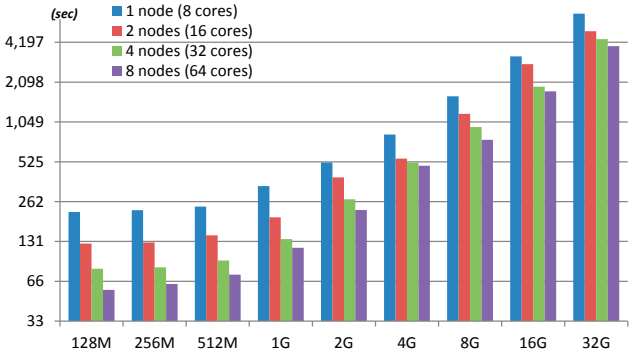
In order to evaluate our embedded Hadoop system for its scalability characteristics and execution performance, we executed a number of MapReduce experimental tests across the Linux Cluster and Set-Top Cluster. All the experiments were performed while varying the degree of parallelism, i.e. by iteratively doubling the number of Hadoop nodes, of each cluster: specifically, from 1 to 8 Linux blades for the Linux Cluster (where each blade contains eight 2GHz processor cores) and from 8 through 64 STBs for the Set-Top Cluster (where each STB contains one 400MHz processor core). The results can be organized in four groups which are presented in the following subsection. We report the average results after executing all tests multiple times.

6.4.1 The WordCount Application

WordCount is a typical MapReduce application that counts the occurrences of each word in a large collection of documents. The results reported in Fig. 6.6(a) and 6.6(b) show that this application scales consistently for both the Set-Top Cluster and Linux Cluster. As the size of the input data



(a) Set-Top Cluster (each node is one STB).



(b) Linux Cluster (each node is a 8-core blade.)

Figure 6.6: *WordCount* execution time as function of problem size (bytes), node count: (a) Set-Top Cluster and (b) Linux Cluster.

increases, the Set-Top Cluster clearly benefits from the availability of a larger number of STB nodes to process larger data sets. The Linux Cluster execution time remains approximately constant for data sizes growing from 128MB to 512MB since these are relatively small, but then it begins to double as the data sizes grow from 1GB to 32GB. In fact, above the 1GB threshold the amount of data that needs to be shuffled in the Reduce task begins to exceed the space available within the heap memory of each node. A similar transition from in-memory shuffling to in-disk shuffling occurs in the Set-Top Cluster for smaller data sets due to the smaller memory available in the STB nodes: specifically, it occurs somewhere between 64MB and 512MB, depending on the particular number of nodes of each Set-Top Cluster configuration.

Table. 6.1 reports the ratios between the execution times of two Set-Top Cluster configurations over two corresponding equivalent Linux Cluster configurations, for large input data sets. [2] The

[2]The values in parenthesis are computed by extrapolating the execution times on the Set-Top Cluster.

	# of Nodes STB / Blade	
Size	8 / 1	64 / 8
1G	49.2	49.4
2G	52.9	41.5
4G	63.9	39.6
8G	(64.5)	(48.8)
16G	(64.5)	(42.1)
32G	(61.3)	(38.3)

Table 6.1: *WordCount* execution time ratio as function of problem size (bytes) and node count.

first column reports the ratio of the configuration with eight STBs over one single blade with eight processor cores; the second column reports the ratio of the Set-Top Cluster configuration (with 64 STBs) over the Linux Cluster configuration (with eight blades for a total of 64 cores.) Across the different data sizes, the performance gap of the Set-Top Cluster relative to the corresponding Linux Cluster with the same number of Hadoop nodes remain approximately constant: it is about 60 times slower for the configuration with 8 nodes and about 40 times slower for the one with 64 nodes. Notice that *these values are the actual measured execution times; they are not modified to account for the important differences among the two systems such as the 5X gap in the processor's clock frequency between the Linux blades and the STBs.* A comprehensive discussion of the reasons behind the performance gap between the two systems and how this may be reduced in the future is given in Section 6.4.5.

6.4.2 HDFS & MapReduce Benchmarks

The second group of experiments involve the execution of a suite of standard Hadoop benchmarks. The goal is to compare how the performance of the Set-Top Cluster and Linux Cluster scales for different MapReduce applications. The execution times of these applications expressed in seconds and measured for different configurations of the two clusters are reported in Fig. 6.7. The numbers next to the application names in the first column denote input parameters, which are specific to each

Benchmarks	8 STBs (8 cores)	64 STBs (64 cores)	1 Blades (8 cores)	8 Blades (64 cores)
Sleep	1285.1	119.6	1223.6	114.5
RandomTextWriter 8	799.6	743.9	177.6	172.0
PiEstimator 1k	461.1	163.5	212.1	52.5
PiEstimator 16k	463.4	474.0	213.7	52.5
PiEstimator 256k	603.6	783.2	214.6	52.4
PiEstimator 4M	1240.9	2048.2	213.9	52.5
PiEstimator 64M	7373.0	10482.5	314.8	58.4
K-Means 1G	3679.2	1149.3	794.7	24.5
Classification 1G	3009.0	784.9	864.7	25.45

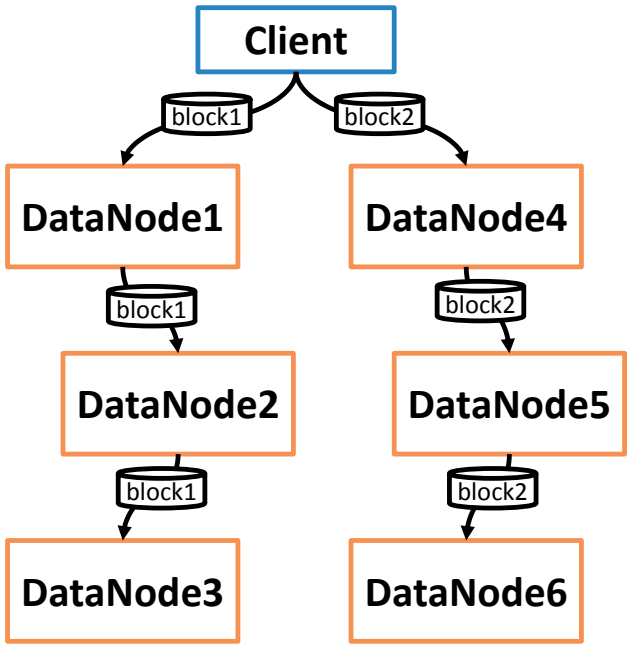
Figure 6.7: Execution times (in seconds) for various Hadoop benchmarks.

application: e.g. "MRBench 16 8" denotes that the MRBench application is running 16 mappers and 8 reducers, while the "Pi-Estimator 1k" denotes that Pi-estimator runs with a 1k sample size.

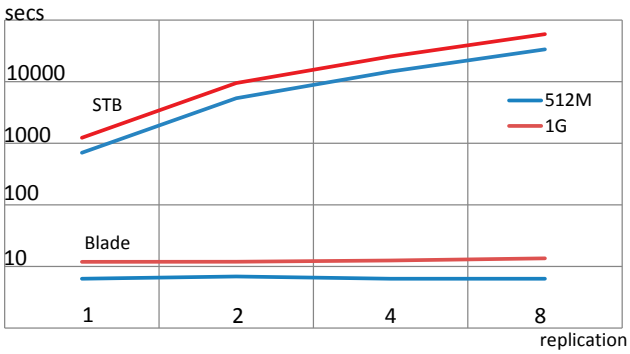
Sleep is a program that simply keeps the processor in an idle state for one second whenever a Map or a Reduce task should be executed. Hence, this allows us to estimate the performance overhead of running the Hadoop framework. For the representative case of running Sleep with 128 mappers and 16 reducers, the Set-Top Cluster and the Linux Cluster performance is basically the same.

RandomTextWriter is an application that writes random text data to HDFS and, for instance, it can be configured to generate a total of 8GB of data uniformly distributed across all the Hadoop nodes. When it is running, eight mappers are launched on each Linux blade, i.e. one per processor core, while only one mapper is launched on each STB node. Since the I/O write operations dominate the execution time of this application, scaling up the number of processor cores while maintaining the size of the random text data constant does not really improve the overall execution time.

Pi-Estimator is a MapReduce program that estimates the value of the π constant using the Monte-Carlo method [12]. For the Linux Cluster, the growth of the input size does not really impact the execution time for a given system configuration, while moving from a configuration with one



(a) Replication pipeline.



(b) Transfer time as function of replication number.

Figure 6.8: HDFS data-replication mechanism (R=3) and replication time.

blade to one with eight blades yields a 4x speedup. For the Set-Top Cluster, in most cases scaling up the number of nodes causes higher execution times because this program requires that during the initialization phase the STBs receive a set of large class files which are not originally present in the Embedded Java Stack. This file transfer, which uses the pipelined mechanism explained in Section 6.4.4, takes a long time that more than cancels out any benefits of increasing the number of Hadoop nodes.

6.4.3 Data Mining Applications

To evaluate the feasibility of utilizing the Set-Top Cluster system for data-mining applications, we performed two experiments based on MapReduce versions of two common algorithms. *K-Means* is a popular data mining algorithm to cluster input data into K clusters: it iterates until the change in the centroids is below a threshold to successively improve the clustering result [45]. *Classification* is a MapReduce version of a classic machine learning algorithm: it classifies the input data into one of K pre-determined clusters [95]. Unlike K-Means, Classification does not run iteratively, and, therefore, does not produce intermediate data.

The last two rows in the table of Fig. 6.7 report the results of running these two applications, each with an input data set of size 1GB. For both applications the results are similar: the execution time when running on the Set-Top Cluster with eight STBs is about four times longer than running in the Linux Cluster with one 8-core blade; furthermore, when both systems are scaled up by a factor of eight, the performance gap grows from four to forty times. The growing gap is mainly due to the fact that scaling up the system parallelism while keeping the input data size constant leads to shuffling a large number of small data sets across the Hadoop nodes. This requires peer-to-peer communication among the nodes, an operation that the DOCSIS network of the Set-Top Cluster does not support as well as the gigabit Ethernet network of the Linux Cluster does. To better evaluate the difference in transfer time between the two networks we complete the following experiment focused on the HDFS data replication, which requires similar peer-to-peer communication among the Hadoop nodes.

6.4.4 Data Replication in HDFS

The Hadoop Distributed File System (HDFS) replicates data blocks through pipelining of DataNodes based on the scheme illustrated in Fig. 6.8(a): for a given replication number R, a pipeline of R DataNodes is created whenever a new block is copied to a DataNode and the data are transferred to the next DataNode in the pipeline until the last one receives it. This mechanism causes a large transfer-time penalty for the Set-Top Cluster due to DOCSIS-network overhead associated with the transfer of data between pairs of Hadoop nodes. Specifically, a DOCSIS network does not support direct point-to-point communications among STBs. Instead, all communications occur between a given STB and the DOCSIS router located in the cable-system head-end: this acts as a forwarding

agent on behalf of the two communicating STBs. Due to this architecture, as we increase the number of STBs in the system (each STB corresponding to one Hadoop node) more slow communications between pairs of STBs occur, thus impacting negatively the overall data-replication time. In contrast, the data replication time spent in the Linux Cluster remains constant as we grow the number of nodes thanks to: (i) the fast communication channels among cores on the same blade and (ii) the gigabit Ethernet network connecting cores across different blades.

6.4.5 Discussion

The performance of executing Hadoop MapReduce applications is influenced by various system properties including: the processor speed, memory, I/O, and networking capabilities of each node. Further, the relative impact of each factor depends on the computation and communication properties of the specific MapReduce application in a way that may vary considerably with the given input problem size and the total number of nodes comprising the Hadoop system. Next, we discuss how the system properties of the Set-Top Cluster compare to those of the Linux Cluster and outline how the technology trends may reduce the gap between the two systems.

Processor performance. In our experimental setup, there is a 5X gap in processor clock frequency between the Set-Top Cluster and Linux Cluster nodes. Further, we empirically noticed another factor of 2X in processing speed which we attributed to the different computer architectures of the 2GHz Xeon and 400MHz MIPS processors. This gap is expected to decrease considerably as next-generation STB devices will incorporate commodity 1GHz+ multi-core processors now found in smartphone and tablets, while it is unlikely that the blade clock frequency will increase much.

I/O Operations. The *RandomTextWriter* benchmark represents many MapReduce applications which execute numerous data-block storage operations. In fact, the Hadoop system itself can be very I/O intensive when performing data replication. We run the *TestDFSIO* test to evaluate the I/O performance of HDFS by reading/writing files in parallel through a MapReduce job. This program reads/writes each file in a separate map task, and the output of the map is used for collecting statistics relating to the file just processed; then, the statistics are aggregated in the reduce task to produce a summary. The results of running TestDFSIO reveal that an STB has 0.115MB/s reading and 1.061MB/s writing speed while the corresponding values for a Linux blade are 68.526MB/s

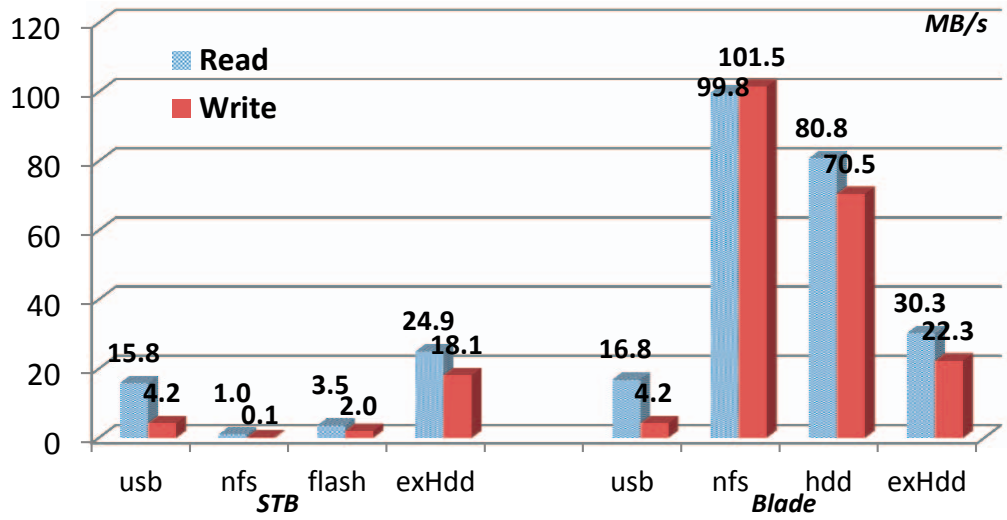


Figure 6.9: Native IO Performance Comparison

and 99.581MB/s. [3] We also run a simple native C program that executes read and write operations using large files on the two clusters with 4 different interfaces: USB, FLASH, NFS, and HDD. The results are reported in Fig. 6.9. We note that the network performance of STB NFS read and write is significantly less, by a factor of nearly 100, than the network performance of the Linux blade server. This gap is primarily due to the DOCSIS network, whose effective transfer rate is limited to 4MB/s compared to 1Gb/s Ethernet network, whose effective maximum transfer rate is closer to 125MB/s. On the other hand, the measured performance of the USB and external hard-drive interfaces on both the STB and Linux blade server is comparable. This is due to the common commodity SoC for USB and disk interfaces used in the design of both the STBs and blades. In our experiments, the Linux blades use an internal hard-drive disk (HDD) while the STBs, which do not contain an internal hard-drive, rely on a USB memory stick whose read performance is 6 times slower (and write performance is 24 times slower) than the HDD when providing HDFS storage. This gap can be reduced by having the STBs use a better file system for the USB sticks than FAT32 such as SFS [94]. Also, as shown in Fig. 6.9, an external USB HDD could provide a 1.5-4.2 speed-up for reading/writing over the USB memory stick. Here, the technology trends should provide next-generation STB devices with HDD and USB 3.0.

[3]The STB shows significant difference between upload and download speed due to the inherently asymmetric and lower transfer rate characteristics of the DOCSIS network.

Networking. The lack of support for peer-to-peer communication among STBs in the DOCSIS network limits considerably the HDFS replication mechanism (as discussed in Section 6.4.4), the Hadoop shuffling operations (as seen for the K-Means, Classification and WordCount programs), and the transfer of large class files during the initialization phase (as in the PI-Estimator). In particular, shuffling generates an implicit all-to-all communication pattern among nodes that is application specific: each node sends its corresponding Map results to other nodes through HTTP, generating $|Node|^2$ communication exchanges, which for the DOCSIS network results in inefficient upstream communication requests as nodes attempt to transfer data blocks from Mappers to Reducers. A similar performance impact occurs during Hadoop replication: for a given replication factor R and a total number of blocks M, the number of DOCSIS upstream communication transfers to complete replication is $M \times (R - 1)$. As the input size increases the number of blocks increases in direct proportion, thus increasing the replication time. The scalability in Set-Top Cluster largely depends on the amount of data to be shuffled generated by the Map tasks and the replication communication overhead. This problem may be addressed in part with the deployment of the higher performance DOCSIS 3.0 standard [40], which supports up to 300 Mb/s upstream bandwidth. Then, opportunities for further improvements include: optimization of the Hadoop scheduling policy, network topology optimization, and leveraging the inherent multi-casting capabilities of DOCSIS to reorder the movement of data blocks among nodes and reduce network contention.

6.5 Related Works

The Hadoop platform for executing MapReduce applications has received great interest in recent years as problems in large-scale data analysis and Big Data have increased in importance. Work in the area of heterogeneous MapReduce computation, however, remains rather limited, notwithstanding the growth of embedded devices interconnected through broadband networking to distributed data centers. Our work is aligned with efforts in the Mobile Space to bridge MapReduce execution to embedded systems and devices. For example, the Misco system implements a novel framework for integrating smartphone devices for MapReduce computation [31]. Similarly, Elespuro *et al.* developed a system for executing MapReduce using smartphones under the coordination of a Web-

based service framework [33]. Besides the fact that our system uses a wired network of embedded stationary devices instead of a mobile network, the main difference with these systems is that we ported the Hadoop framework, including the HDFS, based on the Java programming model. Other related work include utilizing GPU processors to execute MapReduce [46]. While most related work in adapting MapReduce execution to embedded devices has focused on leveraging service-side infrastructure, our work is closer to current research under way for large scale execution of MapReduce applications on the Hadoop platform across Linux blade and PC clusters [113].

6.6 Summary

A heterogeneous system for broadband embedded computing to execute MapReduce applications by leveraging a broadband network of embedded STB devices was developed, implemented and tested. In doing so, we addressed various general challenges to successfully port the Hadoop framework to the embedded JVM environment. We completed a comprehensive set of experiments to evaluate our work by comparing various configurations of the prototype Set-Top Cluster with a more traditional Linux Cluster. First, the results validate the feasibility of our idea as the Set-Top Cluster successfully executes a variety of Hadoop applications. From a performance viewpoint, the Set-Top Cluster typically trails the Linux Cluster, which can leverage more powerful resources in terms of processor, memory, I/O, and networking. On the other hand, for many applications both clusters demonstrate good performance scalability as we grow the number of Hadoop nodes. But a number of problems remain to be solved to raise the performance of executing MapReduce applications in the Set-Top Cluster: in particular, critical areas of improvement include the STB I/O performance and the communication overhead among pairs of STBs in the DOCSIS broadband network. Still, the gap between embedded processors and blade processors in terms of speed, memory, and storage continues to decrease, while higher performance broadband networks are expected to integrate embedded devices into the Cloud. These technology trends hold the promise that future versions of the MapReduce computing system presented in this chapter can help to leverage broadband embedded computing for Internet-scale data-mining and analysis as part of emerging heterogeneous Cloud system platforms.

Chapter 7

Scalable Network Architecture For Broadband Embedded Computing

7.1 Introduction

The communication network among computational nodes is a key building block for all distributed computing and Cloud system infrastructures. Despite the rapid performance improvement of CPU processing power, the communication network is still generally the limiting factor in determining the overall computational system performance [25]. In Chapters 5 and 6, we saw that the inherent characteristics of the DOCSIS broadband network result in a significant difference between the Linux Cluster and Set-Top Cluster performance results, as measured by workload completion time. A second issue, the *process launch problem*, arises when the time taken for launching application processes across a large distributed computing system continues to increase as the number N of compute nodes increases [18]. Here the runtime environment (such as MPI ORTE or the MapReduce Hadoop system) must efficiently manage, in conjunction with the communications network, the distribution and execution of distributed processes in a bounded time window that is small relative to the actual application execution time. Ideally, the runtime environment is able to launch new processes in sub-linear time as the size of the computational cluster increases and for any given message size representing the application executable payload; otherwise as N increases, the runtime system will spend more time initializing and launching application processes in comparison to performing useful computation. The results discussed in Section 7.4.2 will illustrate this point. The

process launch problem is particularly challenging for the implementation of large-scale broadband embedded computing platforms, especially as we scale the computational cluster sizes from just a few hundred or thousand nodes to potentially millions of nodes. Broadband networks of embedded devices on this scale exist today. According to the National Association of Cable Television (NCTA), digital subscriber penetration that includes set-top boxes and other embedded broadband network access devices exceeds 75 million in the Unites States alone [69].

A key issue that has emerged from the work presented in Chapters 5 and 6 is the impact of DOCSIS network communication costs on both runtime system scalability and application execution for large-scale heterogeneous broadband embedded computing systems. This chapter examines the performance of broadband service provider networks, including characteristics of the underlying DOCSIS network, to gain insight to address these challenges. Experiments are developed to evaluate generalized message-delivery performance among DOCSIS device nodes using two application models in conjunction with standard unicast and multicast communication methods. Two network delivery scenarios are evaluated for their scaling properties under different conditions such as traffic load and the persistence of network connections. Experimental testing is performed using both simulation tools and the prototype lab system presented in Chapter 5. Two models, representing the generalized process execution-launch model and the parallel MSA application, are benchmarked to validate both accuracy (simulation versus lab system time measurement comparison) and the potential for large scalability of the number N of computational nodes in the system. Results are presented for DOCSIS network performance under various message size and delivery conditions. Experiments confirm that multicast communication methods minimize message data transmission time, resulting in improved process launch-execution and application completion times. This result is key towards implementing an optimal communications infrastructure for scalable broadband embedded computing. Based on these experimental results, the chapter concludes with a proposed architecture for scalable network architecture for broadband embedded computing.

7.2 Broadband Network Architecture

A typical large-scale broadband service provider system, such as a cable operator system, can be abstracted into the illustration of Fig. 7.1. In this figure, the overall system and broadband network

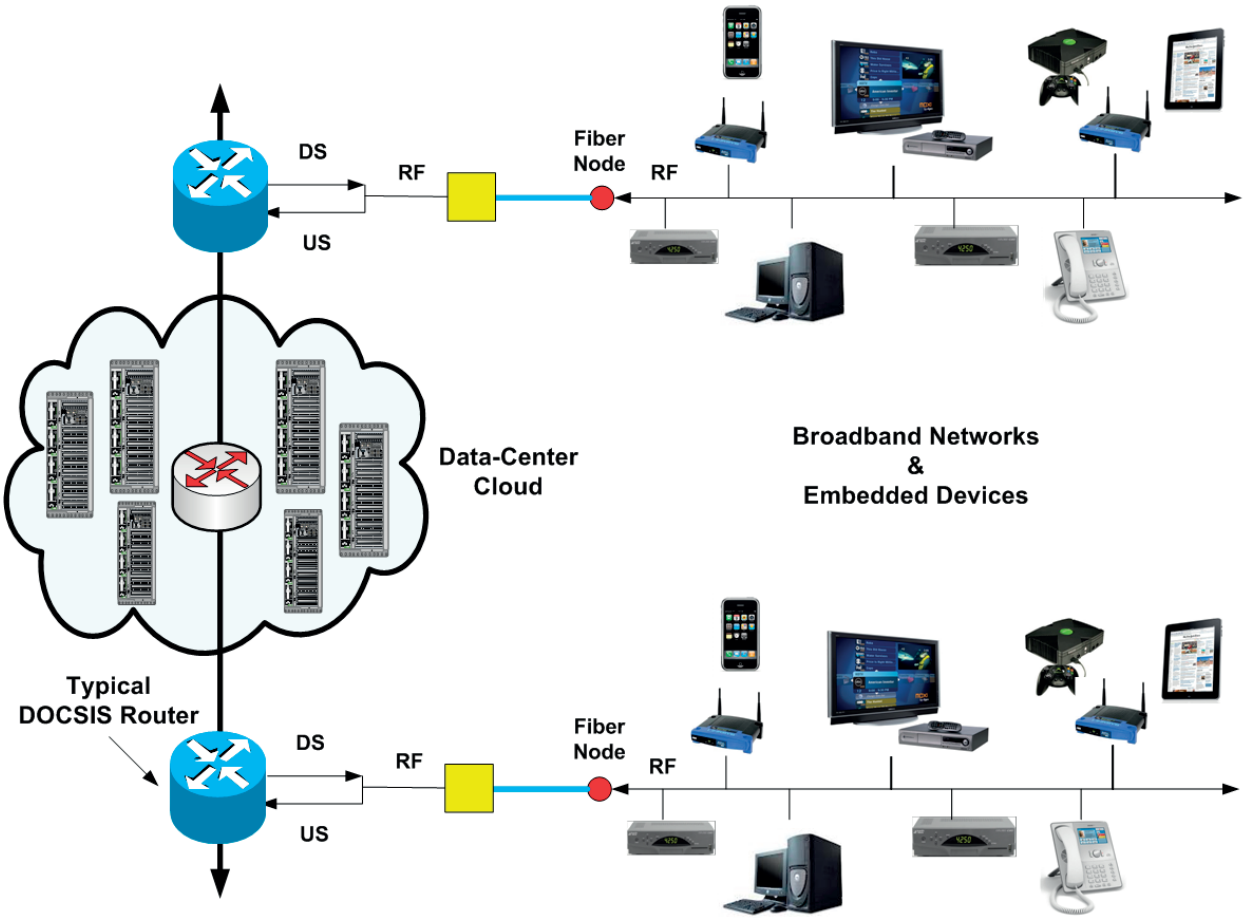


Figure 7.1: Broadband network architecture.

architecture is decomposed as follows: 1) one or more data-center clouds, each containing compute servers and storage, feeding a set of routers that implement a broadband network access based on the DOCSIS standards [57]; 2) one or more broadband networks that support network access to millions of embedded devices including set-top boxes, PCs, game-consoles, and tablets as well as WiFi services. The DOCSIS based broadband network itself provides regional connectivity over RF physical media. However, the RF network is converted to a fiber-based transport at a device known within the cable industry as a *node* (indicated as a red circle in the figure), whose function is back-haul the communications signals to the data-centers and remote hub facilities (refer to Section 2.4.2). Here, these signals are converted back to DOCSIS RF physical layer standards. Note that data communications across DOCSIS networks actually occur bi-directionally using independent downstream (DS) and upstream (US) channels (operating on independent RF frequencies) as

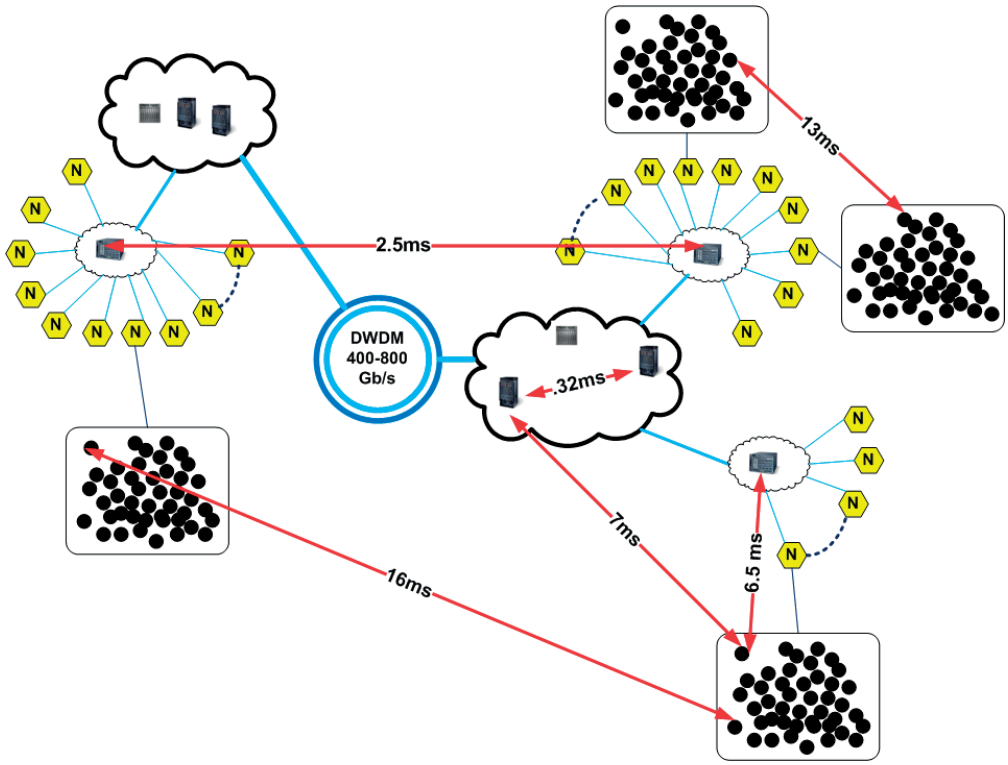


Figure 7.2: Typical round-trip timing for a large service provider broadband network.

indicated in Fig. 7.1. Service provider systems integrate these multiple broadband networks with each other and with centralized managed data-centers using high-speed fiber based technologies, such as dense wavelength division multiplexing (DWDM). This leads to a high-performance wide area network that may span hundreds of miles while connecting millions of embedded devices.

Fig. 7.2 illustrates a high-level broadband network architecture along with typical round-trip ping times between different networks comprising a typical large broadband service provider. Round-trip times were obtained by executing a set of 10 ping operations on an actual service provider system and averaging the set of resulting measurements. The various clouds in Fig. 7.2 represent data-centers or remote-hub facilities that communicate with one another over high-speed fiber transports. These facilities also contain broadband DOCSIS router infrastructure to support the communications of subscriber devices (shown as black dots in Fig. 7.2) with other networks within the system and the Internet. The smaller symbols with the letter N within them indicate the node devices used to convert between RF and fiber physical layer protocols as mentioned above. From Fig. 7.2 we note that the ping times within a data-center are on the order of 0.3ms, whereas

the network delay between data-center systems or remote-hubs is on the order of 2.5ms. Within the DOCSIS network itself, the network delay increases to 6.5ms on average from a typical device node to its closest router. The delay increases to 13ms between any two device nodes within the same DOCSIS router network. The delay is highest, at 16ms, when the device nodes are separated by at least two routers on different DOCSIS networks (DOCSIS networks that are on different remote-hubs). This illustrates the hierarchical nature of a typical service provider network that partitions a large population of broadband devices across multiple DOCSIS networks, each supporting a subset of devices, and a high-speed back-bone network to integrate the broadband networks to each other and the centralized data-centers. The ping times between DOCSIS device nodes from Fig. 7.2 also illustrate the inherent latency that occurs when two DOCSIS devices communicate with one another. Unlike other physical media types, such as Ethernet where two devices communicate directly, DOCSIS-based communication requires that devices communicate through the DOCSIS router. This additional communication cost presents a fundamental performance constraint for broadband networks, effectively doubling the message transfer time required for two DOCSIS devices to exchange messages. This performance constraint is illustrated in Fig. 7.2 where two devices on the same network require 13ms to communicate with one another. DOCSIS networks have the advantage they are fully managed by service providers, hence they are categorized as *managed networks*. A service provider managed network is designed and configured to assure it maintains a minimum performance guarantee to all device nodes, including committed data-rates and bandwidth.

7.2.1 DOCSIS Network Communication Flows

DOCSIS network communications may be described by various logical layer communication flows. As discussed in Section 7.2, the DOCSIS standard defines simultaneous communication flows or transmissions of data in two independent directions; *downstream and upstream*. Downstream refers to all data transfers from the DOCSIS router to a given DOCSIS compatible device node. Whereas upstream refers to the opposite communication flow direction; from a device node to the DOCSIS router. The downstream and upstream communication flows are independent of one another. The DOCSIS standard segments communication across multiple RF frequencies, thereby enabling concurrent upstream and downstream communication channels within the RF broadband physical

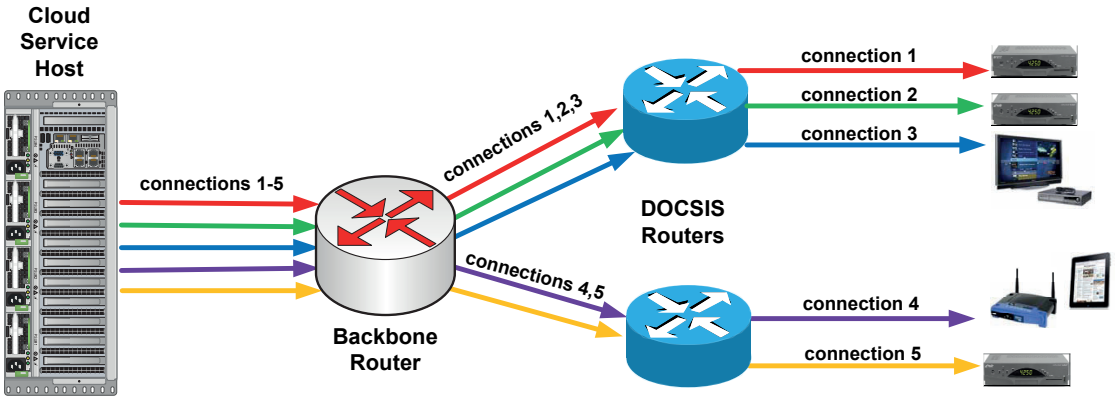


Figure 7.3: Unicast communication flow.

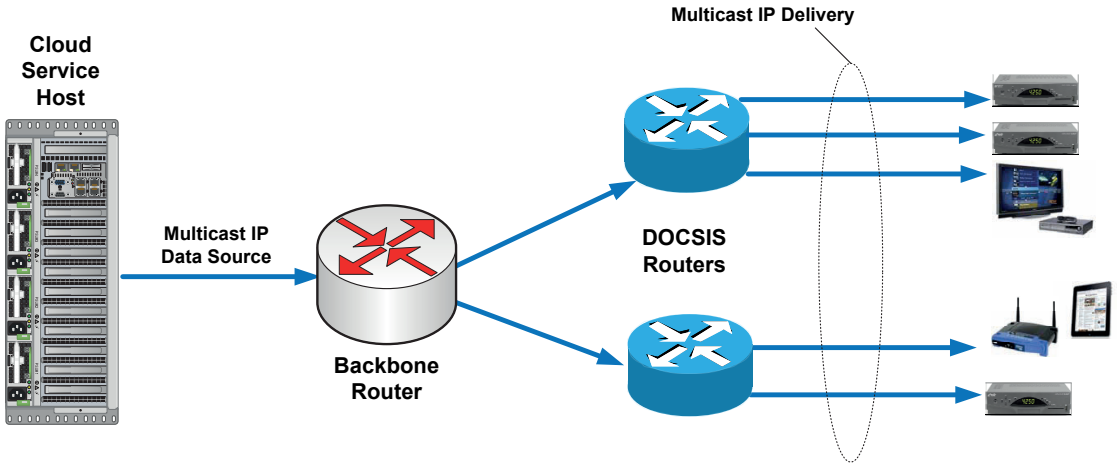


Figure 7.4: Multicast communication flow.

infrastructure. DOCSIS downstream transmission supports both one-to-many and one-to-one communication flows. In contrast, upstream is limited to one-to-one communication flows. This is a characteristic of the DOCSIS standard where upstream transmission opportunities are arbitrated by the DOCSIS router, as defined by DOCSIS TDMA access mechanisms (refer to Section 2.4.1.1). At the logical communications layer, similar to the standard Internet protocol suite defined by

the IETF [5], the DOCSIS network standard includes the definition of both unicast and multicast Internet Protocol (IP) communication flows [57] as illustrated in Figures 7.3 and 7.4, respectively. A unicast flow is defined as a distinct set of transmissions between a single pair of source and destination device nodes. Unicast transmissions are connection oriented, with each transmission requiring an individual copy of data sent to each destination device node, through the TCP/IP delivery mechanism.

In contrast, multicast delivery occurs between a single source host or device node, referred to as the multicast source, and one or more destination hosts or device nodes, defined by a special class of IP addresses and referred to as the multicast group address. Multicasting is far more efficient for scalable data transmission since a single copy of data is transferred to many receivers simultaneously [61]. However, since multicast is based on UDP/IP delivery mechanisms, it lacks reliability assurances like those offered by the TCP/IP protocol. Multicast delivery is considered 'best effort', as it lacks facilities for congestion avoidance, flow-control, or in-order delivery message data guarantees [61], which are instead supported in unicast delivery. Interest in overcoming these weaknesses has resulted in a number of multicast protocol and application-layer library enhancements that include: the NORM protocol specified in IETF RFC-5740 [5], the Reliable Multicast Protocol (RMTP) [101], and the MCL Multicast Library [104]. These are implemented as application layer protocol modules that reside above the UDP/IP multicast transport layer. Multicast IP communications across broadband networks is supported through the *DOCSIS Set-Top Gateway* (DSG) standard [4; 61]. DSG defines a mechanism for IP multicast delivery between the IP multicast source host and DOCSIS-capable device node end-points. A key aspect of the DSG standard is the elimination of the multicast group management protocol known as IGMP requirements [5] for DOCSIS devices. This is accomplished by transferring the multicast IGMP protocol messaging requirements to the DOCSIS router. In this scenario, the DOCSIS router acts as a proxy by managing the required multicast protocol operations [61] on behalf of the DOCSIS broadband network hosts. Using this mechanism, multicast group membership and MAC layer delivery addressing is mapped to one or more statically-defined tunnel addresses, referred to as *DSG tunnels*. DOCSIS devices have the option, through configuration, to bind their low-level multicast address to the appropriate tunnel address during their initialization phase. The DSG standard also requires that multicast IP datagrams match the MTU size of the underlying DOCSIS MAC layer. Therefore DOCSIS routers

do not fragment IP packets that exceed the maximum transmission unit size (no-fragment bit is set in the IP header), which has the effect of requiring both fragmentation and reassembly within the application services layer at the sending and receiving hosts respectively. Finally, DSG supports the notion of an IP broadcast model using a special broadcast tunnel, whose implementation is a generalization of the multicast tunnel framework where the tunnel identifier is set to zero. Similar to IP based broadcast communications, a broadcast tunnel transmits a single message copy to all DOCSIS hosts simultaneously. In summary, in terms of comparing logical Internet Protocol (IP) communication flows with that of DOCSIS, the latter supports equivalent IP models for unicast, multicast and broadcast communication patterns for downstream transmissions, and unicast for upstream transmissions. The choice of which communication flow to utilize is examined in the following sections in order to develop efficient broadband embedded computing software for communication between client and server device nodes. The performance for each of these different flows is evaluated by experiments that measure communication performance when varying message size and the number of nodes.

7.3 Experimental System Environments

Experimental testing for broadband performance evaluation is conducted using both OPNET simulations and a lab system environment consisting of 128 STB nodes. The OPNET simulation tool executes across a cluster of 5 Linux high-performance blade servers, each consisting of 8 2.1GHz cores and 64GB of memory. This enables up to 5 concurrent simulations to execute in parallel. The lab environment utilizes the Open MPI broadband embedded computing system implementation described in Chapter 5. The lab system environment enables the execution of MPI test applications that measure communications and computation performance.

7.3.1 Simulation Environment

The simulation environment is based on OPNET Network Modeler (version 17.1). This is a commercial grade, discrete-event simulation tool, containing advanced network simulation models, including broadband-specific network model support for DOCSIS and DSG standards. The OPNET tool supports simulation of a large network of nodes, thus enabling insight into the scaling capa-

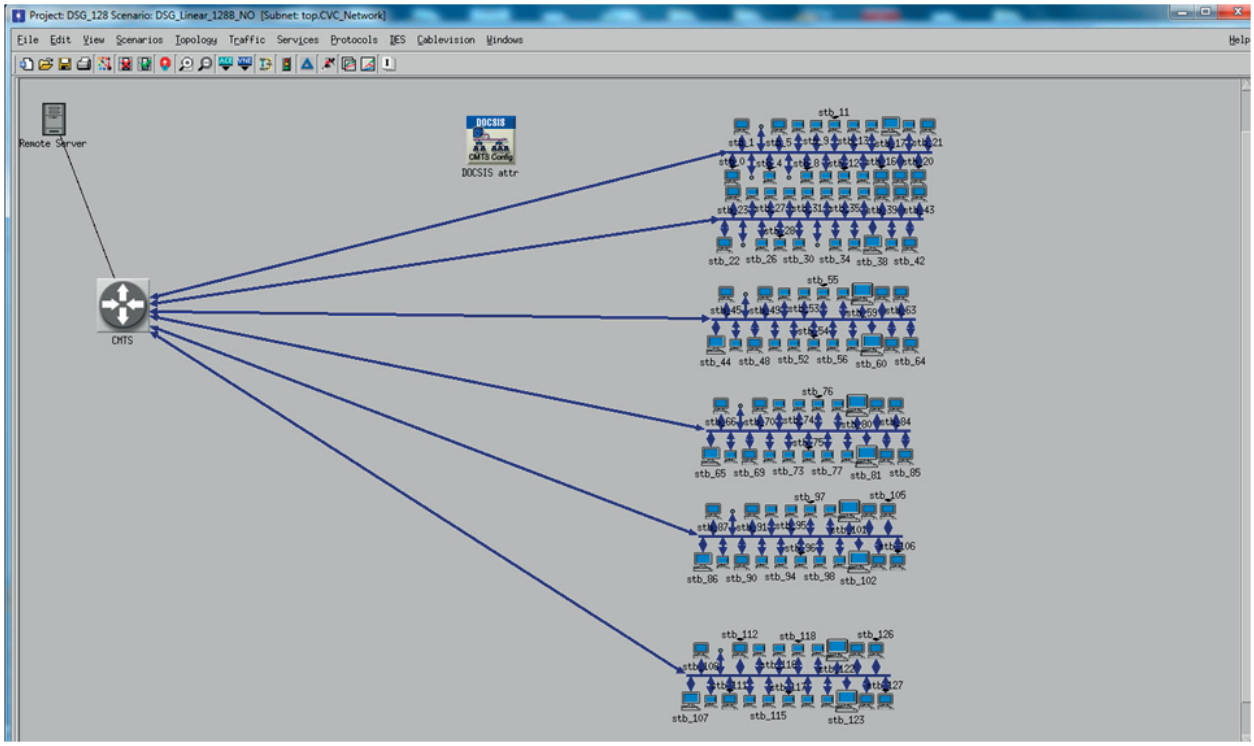


Figure 7.5: OPNET scenario snapshot illustrating server, DOCSIS router, and broadband network of device nodes.

bilities of various DOCSIS broadband network systems, including those with and without external traffic profiles. Experimental simulation instances are defined as Modeler scenarios; these are developed by using either the Modeler GUI or the OPNET automated scenario generation tools (required when the number of nodes is large) to draw interconnections between network nodes such as servers, DOCSIS routers, and one or more DOCSIS device nodes such as cable-modems or STB devices. Fig. 7.5 illustrates a completed scenario design within Modeler for the simulation of a single server, DOCSIS router and a small network of 128 DOCSIS device nodes. Once the network node topology is created, attributes within the nodes such as their respective application model or DOCSIS parameters are modified to establish the simulation execution parameters of the given simulation scenario. Attribute definitions are node-model specific and vary in definition among the numerous node models OPNET supports within Modeler. OPNET supports two types of attribute levels: node and global. When both attributes are defined for the same attribute entity, node attributes take precedence over global attribute values. Node attributes are specific to a given

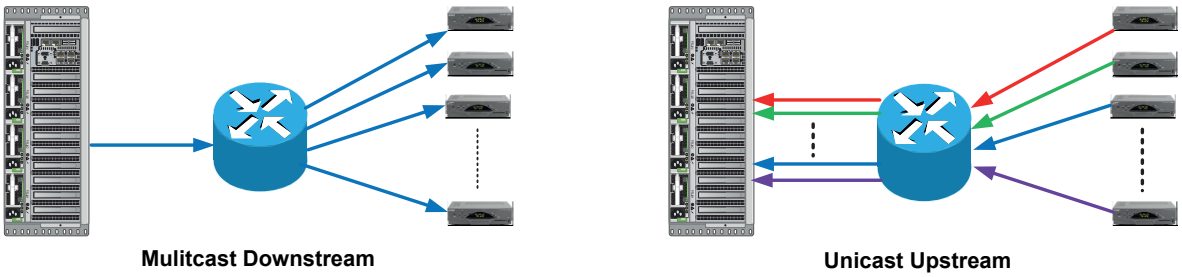


Figure 7.6: Multicast-unicast test configuration.

node instance, whereas global attributes are defined across all nodes within the simulation scenario. As an example, the number of DOCSIS router interfaces is a node-level attribute. This attribute enables one to control the number of downstream and upstream interfaces. An example of a global attribute is the Ethernet Maximum Transmission Unit (MTU) size. In this case, all device nodes are configured with the same Ethernet MTU. Global attributes relieve the user of repetitive effort when configuring an OPNET simulation scenario.

Utilizing global attributes for configuring both the DOCSIS router and the device nodes within simulation enables efficient set-up of a large broadband network of embedded devices. As a specific example, to simulate a service provider remote-hub facility, the total number of device nodes in the experimental OPNET scenario varies between 128 and 8K nodes. In this case, the OPNET DOCSIS router model is configured to include six 5x20 RF line cards, or 30 downstream RF links and 120 upstream RF links. Note that for any given number N of DOCSIS device nodes, the N nodes are distributed evenly across all upstream RF links. For scenarios that require background traffic, OPNET also supports the addition of a background traffic profile attribute. The traffic attribute is a configurable global DOCSIS attribute. This attribute can be configured with a static value or a value taken from a distribution function, applied to either downstream or upstream communication channels.

A global attribute is also used to define communication flow mechanisms between server and device nodes. Two communication flow mechanisms are illustrated in Figures 7.6 and 7.7. Depending

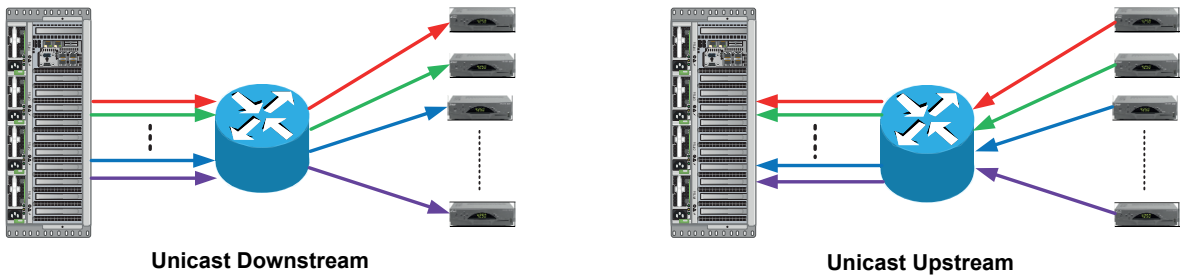


Figure 7.7: Unicast-unicast test configuration.

on the experiment, global attributes are defined for multicast (downstream only) or unicast (downstream or upstream) topologies. The multicast-unicast case illustrated in Fig. 7.6 is based on the multicast capabilities of the DOCSIS network or DSG, as previously defined, where an IP multicast server sends a single copy of message data in the downstream to all device nodes concurrently. In all cases, DOCSIS supports only unicast communications in the upstream flow, therefore multicast may be leveraged only for downstream flows. In the second case illustrated in Fig. 7.7, unicast communication flows are utilized in both downstream and upstream directions. In this case the server is transmitting the same data N times, once to each device node. For unicast flows, network connections may be persistent or non-persistent. Persistent connections are TCP/IP connections between two devices that remain established for the life of a scenario execution. Conversely, non-persistent connections are connections that are created and destroyed one or more times during scenario execution. OPNET provides an attribute to control the connection persistence in the device node model.

In addition to the required DOCSIS interface organization and communication flow model configurations, OPNET models may be extended to contain a task-graph of computation and communication transactions. OPNET models containing a definition of this type are referred to as *application models*. Application models enable the simulation of task-graphs that capture process completion times based on simulated communication flows and configurable compute-time values that may be statically defined or taken from various distribution functions. Section 7.4 describes the application models used in the chapter experiments.

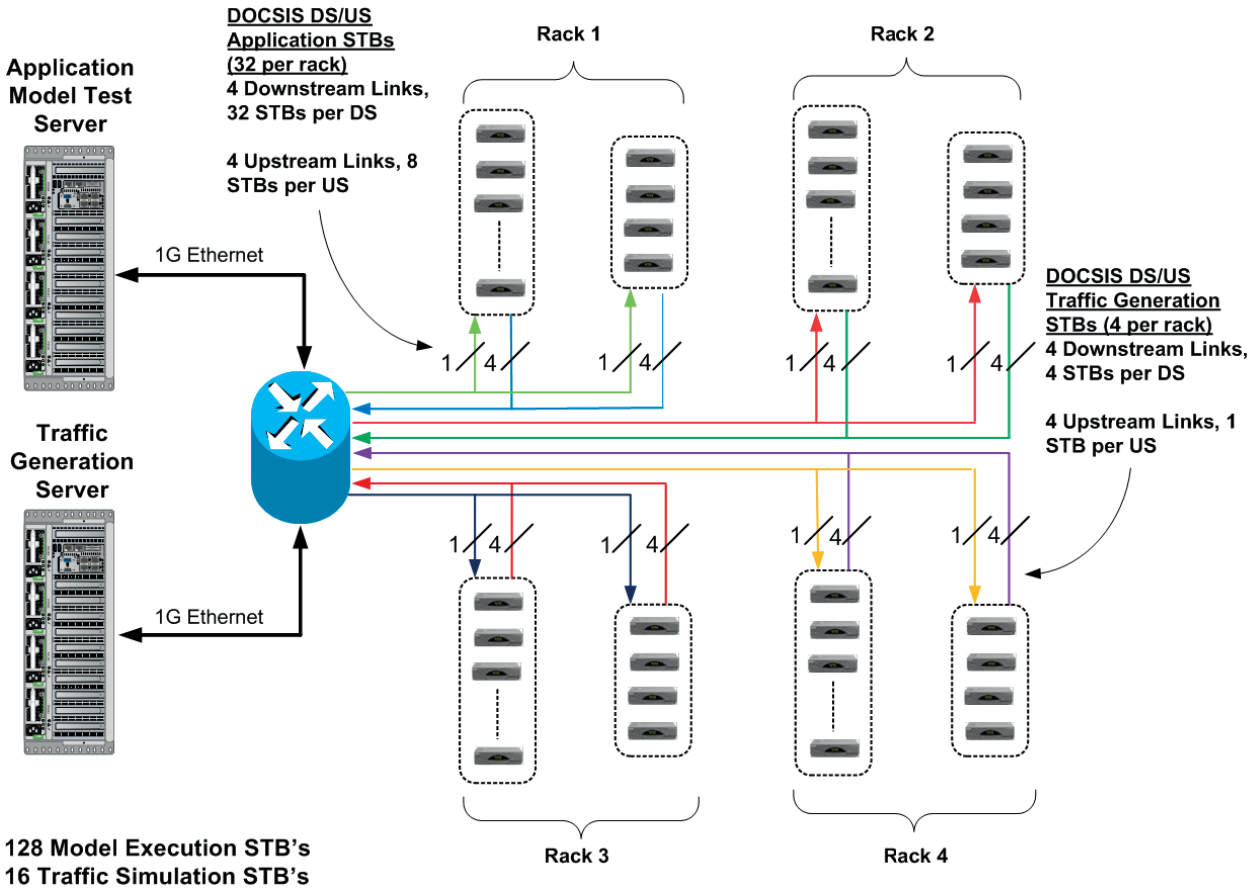


Figure 7.8: Experimental lab system configuration.

7.3.2 Lab System Environment

To evaluate the simulation environment accuracy and gain further insight into factors effecting broadband embedded computing system performance, the 128 node experimental lab system from Chapter 5 is utilized with the enhancement of a traffic-generation system to simulate additional load on the DOCSIS broadband lab network. The traffic-generation system implementation supports the generation of background downstream traffic to match the OPNET simulation environment traffic profile. It consists of 16 additional STB nodes and a traffic generation server integrated into the 128 STB node experimental system to enable realistic test scenarios in the presence of DOCSIS traffic. The block diagram of Fig. 7.8 illustrate the experimental lab system set-up from a DOCSIS physical network perspective. The lab system is partitioned into four racks each containing 32 STBs used for application model execution and 4 STBs to simulate an approximate 20Mb/s DOCSIS downstream

traffic flow profile. The DOCSIS router contains two 2x8 DOCSIS RF line cards. Each line card supports two downstream and eight upstream interfaces. As shown in Fig. 7.8, each rack is wired with dedicated and independent links consisting of one downstream and four upstream RF links. Multiple downstream and upstream links provide for improved traffic distribution, especially for upstream flows where devices gain additional transmission opportunities through the DOCSIS channel arbitration protocols (refer to Section 2.4.1.1). Each rack contains 32 STBs for application model execution; an independent and single downstream RF link feeds each rack group of 32 STBs. Similarly, an independent set of four upstreams RF links feeds each rack, thus one upstream RF link is associated to each set of 8 STBs within the given rack.

Each rack also contains 4 traffic generation STBs (system total of 16 traffic generation STBs) dedicated for experimental DOCSIS traffic scenarios. These 16 STBs share the same downstream and upstream RF links used by the 128 STBs used for application model execution. A single server, based on the Open MPI and OERTE infrastructure implementation from Chapter 5 is extended to support multicast as well as unicast application delivery to execute experiments in both the unicast and multicast experimental scenarios. In the multicast scenario, the application model test server acts as a multicast IP source host in conjunction with DOCSIS DSG tunnels supporting broadband multicast data delivery to all 128 STBs. Experiments that include background traffic utilize a second server that generates a continuous approximately 20Mb/s data flow over the shared upstream and downstream RF links to the 16 STBs described earlier.

A multi-threaded TCP/IP listener process has been developed to implement a 20Mb/s aggregate downstream traffic flow. The TCP/IP listener is a server process that establishes 16 threads, with each thread listening for an incoming STB connection request, one from each of the 16 STBs. The server threads continuously process data-transfer requests from the 16 STB nodes at the rate of 10 requests per second per STB. In other words, each of the 16 server threads continuously sends a 16K data buffer, 10 times per second. For example, using 16 STBs, a 20Mb/s downstream traffic profile is achieved by each server thread, transferring 16,000 bytes/per-sec (20Mb/s = 16 threads x 16,000 bytes x 10 per/sec x 8 bytes bits/sec) spread across the 4 downstream RF channels. Each rack of 32 STB devices sustains a 5Mb/s background downstream traffic profile. The traffic generation is started on each STB device prior to executing any application model experiments on the 128 node

system. Traffic generation is then continuously maintained during the execution of lab experiments whose scenario requires the presence of background traffic.

7.4 Experiments

Experiments consist of 24 independent tests that are performed across both the simulation and lab environments described in Section 7.3. The first experiment consists of 16 tests (8 simulation and 8 lab tests) that are designed to evaluate message delivery performance in addressing the process launch problem described in Section 7.1. The key experimental goals are: 1) understand under what scenarios DOCSIS networks can be leveraged to optimize message delivery performance to a large number of device nodes given various network conditions. 2) utilize experimental results to define an implementation strategy for scalable broadband embedded computing runtime environments.

The 8 simulation tests utilize a task-graph representation for modeling the process launch and execution of distributed application processes across N device nodes, where N varies from 128 to 8K. This model is referred to as the *process launch-execution application model*. The corresponding 8 lab tests utilize a MPI test application that emulates the process launch-execution model, but is fully instrumented to capture timing information when executed across the lab environment. In each set of process launch-execution model experiments, network attributes for communication flow, background traffic, and connection persistence are varied under multicast-unicast or unicast-unicast scenarios, generating all 8 test combinations for lab and simulation, respectively. To further clarify, DOCSIS communication flows are tested using both multicast-unicast and unicast-unicast configurations. For each communication flow, the impact of a continuous stream of 20Mb/s background downstream traffic is evaluated, with and without traffic. The methodology for generating a background DOCSIS network traffic is described in Section 7.3. Finally, the effect of unicast network connection persistence is evaluated for each communication flow with and without traffic. Typically, applications establish network connections on-demand using the standard TCP/IP 3-way handshake protocol. Network connections created in this manner are considered non-persistent connections. For example, TCP/IP sockets in this mode are opened, utilized, then closed with each network operation. However, persistent connections are those where all network connections are pre-established and remain established during the lifetime of the network application process. In

this case, TCP/IP 3-way handshakes are eliminated and the network initialization delay overhead associated with opening and closing TCP/IP socket connections is minimized [110]. Persistent and non-persistent connections are also referred to as *prewired and not-prewired* respectively. This terminology is used throughout the reminder of this chapter.

The second experiment includes 8 tests, consisting of 4 simulation and 4 lab tests, based on the execution of the parallel MSA algorithm presented in Section 4.5. An important goal of this experiment is to: 1) compare the application execution performance results from simulation with those taken from lab system execution in order to validate the accuracy of simulation results with respect to those obtained through lab measurements, and 2) gain insight into a key open question on the potential of scaling the execution of the parallel application across a large-scale broadband embedded computing system. Each of the 4 simulation tests approximate the execution performance of parallel MSA as the number of device nodes is increased, providing insight to this open issue. The parallel MSA simulation and lab tests include 4 test combinations each: DOCSIS multicast-unicast and unicast-unicast scenarios are evaluated with and without background traffic generation. Prewired and not-prewired use-cases are not evaluated. The 4 lab tests utilize an instrumented version of parallel ClustalW based on MPI that implements the parallel MSA algorithm. The lab MSA test application is based on the same parallel ClustalW MSA implementation discussed in Section 4.5. However, it is modified to generate application-level computational durations, network-transfer delays, and message sizes for various input sequence data sets. All results are recorded for later use within simulation. The 4 simulation tests are based on an extension of the basic process launch-execution model discussed previously for the first experiment that captures additional computation and communication operations necessary to accurately represent the parallel MSA algorithm. Measurements from lab testing are back-annotated to the parallel MSA application model in order to configure its attribute values for simulation.

The set of 24 tests comprising the process launch and parallel MSA applications experiments fully characterize the most important message-delivery characteristics of managed DOCSIS broadband networks. In the following sections, both experimental models are described and experimental results are presented.

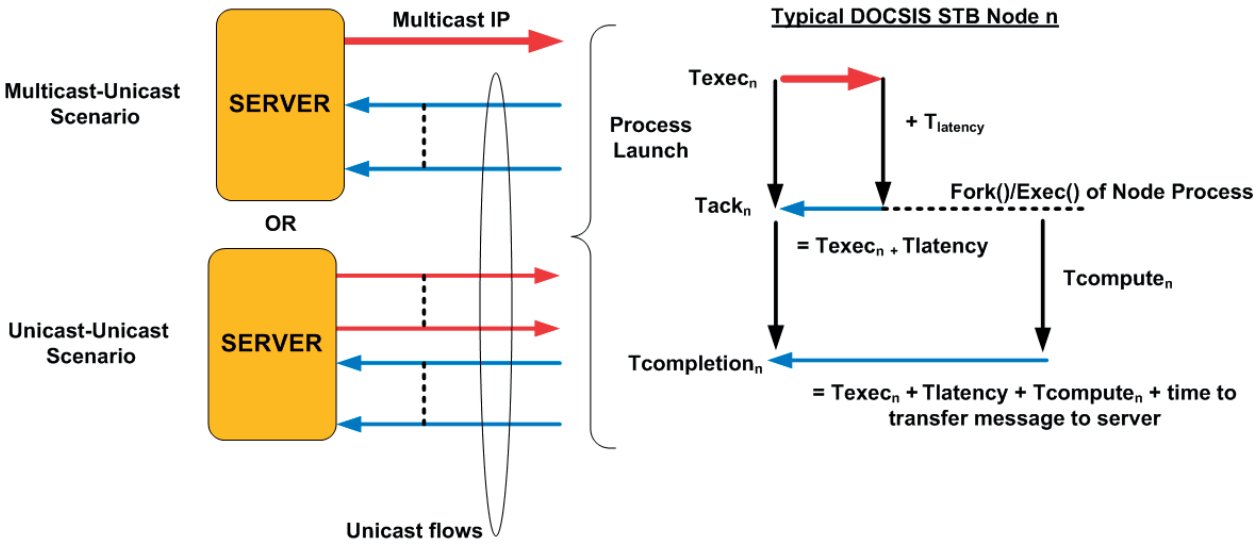


Figure 7.9: Generalized process launch-execution model.

7.4.1 Process Launch-Execution Application Model

The process launch-execution application model illustrated in Fig. 7.9 provides an equivalent task-graph representation of a network application process launch and execution. The model utilizes either multicast-unicast or unicast-unicast DOCSIS communication mechanisms, between the server and n = 1..N embedded device nodes, as discussed in Section 7.2.1. Model notation includes the use of lowercase n, which refers to an individual device node; whereas the use of uppercase N refers to the total number of devices for a given test configuration.

The process launch-execution model provides three important attributes or timing parameters, Texec$_n$, Tack$_n$, and Tcompletion$_n$. Each timing attribute is equal to a message transfer time duration associated with a variable-size message transmission between the server and a unique device node. Timing attributes correspond to a single request-response communication operation. Two additional model attributes shown in Fig. 7.9, Tlatency and Tcompute$_n$, represent internal time duration state within a given DOCSIS device node n. Tlatency is a measure of internal runtime overhead or time taken to respond from receipt of a given Texec$_n$ message to initiating the transmission operation of an acknowledgment message. Tcompute$_n$ provides a measure of internal model computation. Tcompute$_n$ is a configurable attribute within simulation, while it is a measured value during lab MPI test application execution. During lab execution of this model,

Tcompute is measured before the MPI_Init() function and includes the time to load and complete the initialization of the MPI application. Measurement of Tcompute completes after MPI_Finalize() including the exit time of the MPI test application, and the additional time required to issue the final Tcompletion operation.

Texec_n	Time for transferring message from server to DOCSIS client device node n measured at client device node n.
Tlatency	Internal model message processing overhead measured from receipt of Texec message to transmission of acknowledgment message at DOCSIS client device node.
Tack_n	Texec_n time + Tlatency time + time for DOCSIS client device node n to transmit acknowledgment message to server measured at server.
Tcompute_n	Computation time duration attribute representing process execution time within DOCSIS client device node n.
Tcompletion_n	Texec_n time + Tlatency time + Tcompute_n time + time for DOCSIS client device node n to transmit a completion message to server measured at server.

Table 7.1: Timing parameters for the process launch-execution model.

The process launch-execution application model parameters are summarized in Table 7.1. The model is characterized by a sequence of 5 main steps:

1. The server node transmits (downstream using multicast or unicast flow types) a configurable message size, corresponding to a process launch message, to all DOCSIS device nodes. Texec_n is defined then as the time duration required for a given node n to receive a launch message sent by the server node;

2. Each node n initiates an upstream acknowledgment message to the server node after time Tlatency from the instant it receives its Texec_n message;

3. Tack_n is defined as the time for node n to receive its launch message (Texec_n duration), plus Tlatency delay, plus the time required to transmit an acknowledgment message upstream and received at the server node. This message is always a unicast communication flow;

4. Application process computation is modeled as Tcompute_n duration within node n. This includes any initialization and finalization overhead as would be the case under lab conditions;

Attribute	Simulation	Lab
#Nodes	128,512,1K,2K,4K,8K	32,64,128
Texec_n transmission message size (Bytes)	128,512,2^k for k = 10..20	512,2^k for k = 10..20
Tlatency	0us	N/A
Tack_n response message size (Bytes)	128	128
Tcompute_n	10us	N/A
Tcompletion_n response message size (Bytes)	1K	1K

Table 7.2: Process launch-execution model attribute configuration.

5. A final completion message transmission is sent from node n to the server node to signal the completion of one model cycle for node n. This final message is called Tcompletion_n and is defined as the total time from process launch-execution or Texec_n to the time the completion message is received at the server node from node n. Therefore, it is the sum of multiple time durations or $\text{Tcompletion}_n = \text{Texec}_n + \text{Tlatency} + \text{Tcompute}_n$.

In both the experimental simulation and lab execution case, individual results for Texec_n, Tlatency, Tack_n, Tcompute_n and Tcompletion_n are collected. However, to evaluate the worst-case performance behavior of the overall network topology, maximum measured values are defined as Texec_N, Tack_N, and Tcompletion_N. With this interpretation, 1) Texec_N is the time duration required for all N nodes to receive their process launch-execution message; 2) Tack_N is defined as the time taken for all N nodes to deliver their acknowledgment messages as measured at the server node; 3) Tcompletion_N is the completion of a launch-process execution cycle across all N nodes as measured at the server node. Finally, to consider Tcompute_n values across all N nodes, the average of all Tcompute_n values is computed for all N measured device nodes. This is denoted Tcompute_{avg}.

7.4.2 Process Launch-Execution Application Model Experiments

The experiments for Texec_N, Tack_N and Tcompletion_N are executed and measured 3 times, taking the average, for each scenario with messages sizes and other initial conditions for the process launch-execution application model attributes as specified in Table 7.2. Note that Tlatency for this experiment is neglected and set to 0us for simulation. For lab tests, it is also assumed small

and therefore neglected (indicated as non-applicable or N/A in Table 7.2). Tcompute$_n$ attribute is set to 10us for simulation, and measured (indicated as N/A in Table 7.2) for lab experiments. In each experimental scenario, Texec$_N$, Tack$_N$, and Tcompletion$_N$ times are measured under the two 20 Mb/s background traffic profiles defined in Section 7.3. Both persistent (wired) and non-persistent (not-prewired) connections are considered as well as multicast-unicast or unicast-unicast communication flows described in Section 7.3. Simulation results are shown in appendix Figures A.1 through A.8. Lab results are shown in appendix Figures A.9 through A.16.

7.4.2.1 Process Launch-Execution Multicast-Unicast Results

An important result across both simulation and lab multicast-unicast scenario executions is the near constant performance of multicast downstream communications performance for n = 1..N. This result is observed as uniform Texec$_N$ time measurements independent of the number of device nodes N in the corresponding experiment. Results for simulation are shown in appendix Figures A.1 through A.4. Lab execution results are shown in appendix Figures A.9 through A.12. The figures show that the increase in Texec$_N$ time is consistent across both simulation and lab tests, as message size increases, and remains constant as the number of nodes increases from n = 128 to 8K. Tack$_N$ and Tcompletion$_N$ follow a linear increase as both message size and the number of device nodes increases from n = 128 to 8K. This is due to the required unicast communication pattern which is linear in n for all upstream communications. The amount of transmission time required is directly proportional to message size (increasing data size and consequently number of transmissions) or number of device nodes. Results for the no-prewired, without traffic, multicast-unicast experimental simulation scenario are listed in Table 7.3. The table shows the uniform performance at .07ms in each case for Texec$_N$ and the linear increase (45ms to 484ms at 128B message size as an example) for unicast Tack$_N$ and Tcompletion$_N$ times where n varies between 128 and 8K device nodes. As message size increases from 128 bytes to 1MB, all timing parameters increase in proportion to the message size. We note negligible differences in simulated results between Tack$_N$ and Tcompletion$_N$ as Tcompute is set to 10us within simulation.

To evaluate the change in Texec$_N$, Tack$_N$, and Tcompletion$_N$ times as the number of device nodes increases from 128 to 8K, we take the ratio of the measured values at both N = 8K and N = 128 device nodes. Ratios are computed for Texec$_N$, Tack$_N$, and Tcompletion$_N$ and listed

#Nodes	Msg Size	$Texec_N$ (ms)	$Tack_N$ (ms)	$Tcompletion_N$ (ms)
128	128B	.07	45	45
8K	128B	.07	484	484
128	1MB	280	322	322
8K	1MB	287	716	724

Table 7.3: $Texec_N$, $Tack_N$ and $Tcompletion_N$ time simulation measurements under multicast-unicast, not-prewired no-traffic scenario.

Msg Size	$Texec_{8K}/Texec_{128}$ (ms)	$Tack_{8K}/Tack_{128}$ (ms)	$Tcompletion_{8K}/Tcompletion_{128}$ (ms)
128B	1	11	11
1M	1	2	2

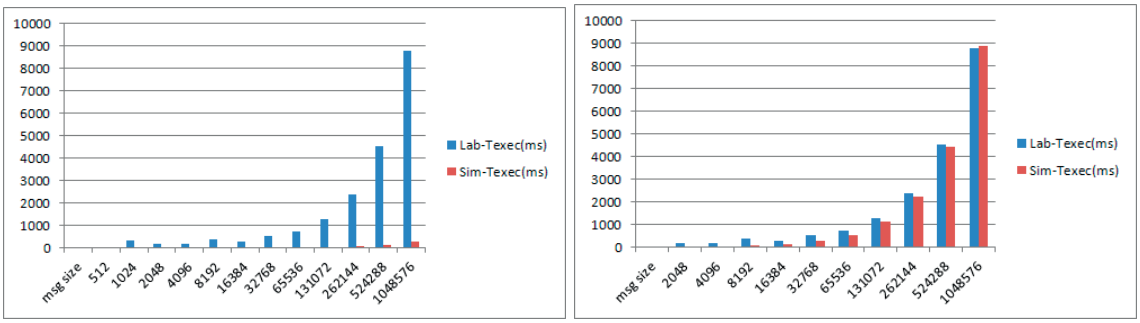
Table 7.4: $Texec_N$, $Tack_N$ and $Tcompletion_N$ time ratios (case N=8K over N=128) under multicast-unicast, not-prewired no-traffic simulation scenario.

in Table 7.4. $Texec_N$ ratio remains equal to 1 independently from message size, highlighting the efficiency of multicast communications. In contrast, $Tack_N$ and $Tcompletion_N$ ratios equals 11 for 128 byte messages and 2 for 1MB messages respectively. This indicates that in comparison to multicast communications, unicast upstream communications becomes less efficient as the number of nodes increases. The ratio decreases from 11 to 2 as the message size increases from 128 bytes to 1MB. This also indicates that it is more efficient to transmit larger messages then smaller messages, since 1MB messages only take twice as long to deliver as we increase the number of nodes from 128 to 8K. In contrast, 128 byte messages take 11 times as long for the same increase in device nodes. This result is due to the additional DOCSIS protocol contention overhead as the number of device nodes increase. In addition, there are fewer TCP/IP acknowledgments required for larger message sizes in comparison to smaller messages. Therefore, it is more efficient to transmit larger messages during unicast TCP/IP communications.

Lab results are shown in appendix Figures A.9 through A.12; graphs of $Texec_N$, $Tack_N$ and $Tcompletion_N$ are similar to simulation cases, but with overall higher time duration values respectively. The difference in comparative performance results between simulation and lab execution is a consequence of the DOCSIS broadband multicast network and MPI test application environment. Primarily: 1) data transmission reliability issues with the DOCSIS multicast delivery that

Msg Size	#Nodes	Texec$_N$ (ms)	Tack$_N$ (ms)	Tcompute$_{avg}$ (ms)	Tcompletion$_N$ (ms)
512B	128	10	74	637	670
	64	10	42	348	402
	32	12	47	212	255
1MB	128	4363	4399	592	4981
	64	4364	4400	377	4770
	32	4365	4403	275	4635

Table 7.5: Texec$_N$, Tack$_N$, Tcompute$_{avg}$, and Tcompletion$_N$ time measurements for Lab execution tests under multicast-unicast, not-prewired no-traffic scenario.



(a) without compensation factor

(b) with compensation factor

Figure 7.10: Lab versus simulation Texec$_N$ multicast results with and without 10ms per-transmission compensation factor.

require the addition of a small delay between transmissions to ensure reliable data delivery; 2) additional Tcompute$_{avg}$ overhead due to the second-generation system (see Chapter 5) lab MPI runtime environment that contributes additional time delays during MPI test application execution.

Impact on lab results due to STB multicast reliability. Initial lab testing exhibited data loss during multicast transmission indicating the presence of data transmission reliability issues. This may be due to two reasons: 1) there exists an intrinsic software architecture issue within the test STB device networking stack where a time delay is required to transfer a message from the network driver layer to the operating system TCP/IP stack. 2) since the STB software stack does not provide a reliable multicast communication stack [5] as discussed previously, data integrity is not assured. A number of tests conducted show that if a delay is implemented in the server send operation for each transmission, data loss at the device node is eliminated. Any delay less

than 10ms resulted in data loss at the device node. Therefore 10ms was chosen empirically as a correction time factor to ensure reliable multicasting during lab tests. However, ensuring reliability impacts network performance. This is confirmed by the longer Texec times observed in the lab test results compared to simulation results.

In order to compare Texec_N simulation results with lab measurements, simulation results for Texec_N are adjusted by applying a correction time factor. The correction time factor is computed by simply multiplying the number of transmissions required for a given Texec_N message size by 10ms and adding this to Texec_N. The number of transmissions is approximated by dividing the Texec_N message size by the size of the DOCSIS packet PDU or 1280 bytes. To evaluate the impact of the transmission delay compensation time factor on lab versus simulation accuracy, Figure 7.10 illustrates the change in performance differential with and without the 10ms per transmission correct time factor. The left side of the figure illustrates the actual difference between simulation and lab results, without network time compensation. The right side of Figure 7.10 shows a much improved accuracy for lab-versus-simulation results once the 10ms per transmission compensation time factor is included into the Texec_N results.

Impact on lab results due to MPI system overhead. The MPI test application and underlying MPI runtime environment require multiple MPI application and stack-level initialization steps that are included in lab Tcompute_n measurements. For example, required MPI libraries and MPI initialization operations within MPI_Init() (which is the first MPI executable statement for any MPI based application) generate low-level MPI network stack operations between the STB client devices and server. These upstream unicast requests add additional delays as a function of STB, server, and network storage services (see the system architecture from Section 6.2). To illustrate this point, Table 7.6 contains the average of all Tcompute_n values or Tcompute_{avg} timing results where N = 128, 64 and 32 lab STB device nodes when executing under the multicast-unicast, not-prewired, no-traffic scenario corresponding to Figure A.9. For a given number of device nodes there is a relative difference less than 9% in Tcompute_{avg} time duration when comparing the smallest and largest Texec_N message size. Tcompute_{avg} itself, however, increases with scaling from 32 to 128 device nodes. This is due to the additional unicast MPI test application initialization server and storage requests as the number of STB device nodes increases. Results indicate an approximately 5% to 8% variation in Tcompute_n values across STB device nodes for 32, 64 and 128 device

#Nodes	Msg Size	Tcompute$_{avg}$ (ms)	Std Deviation (ms)
128	512B	637	26
128	1MB	592	24
64	512B	348	20
64	1MB	377	23
32	512B	212	21
32	1MB	275	13

Table 7.6: Lab measurements illustrating variation in Tcompute$_{avg}$.

nodes. Table 7.6 lists the standard deviation for typical results. Variation in Tcompute$_n$ may be explained by systemic MPI initialization overhead for loading both MPI runtime environment and libraries. STB nodes whose unicast requests complete their initialization process earlier have lower Tcompute$_n$ values in comparison to those STB nodes that complete their requests later and exhibit higher Tcompute$_n$ values. The larger STB Tcompute$_n$ values impact Tcompletion$_N$ time (worst case) because the process-launch execution cycle is not complete until the server receives Tcompletion$_N$ (final message from Tcompletion$_n$).

Referring to appendix Figures A.9 through A.12 note that Tack$_N$ and Tcompletion$_N$ appear nearly constant across all lab results. This effect is due to the lab environment where insufficient number of nodes exist to measure linear performance characteristics as observed in Tack$_N$ and Tcompletion$_N$ simulation results, particularly for unicast upstream communication flows. The presence of traffic has minimal effect on both simulated and lab Texec$_N$, Tack$_N$ and Tcompletion$_N$ performance. The lack of contention in the presence of traffic is due to the abundant number of DOCSIS downstream and upstream channels available. Recall from Section 7.3 that the DOCSIS router configuration includes six 5x20 DOCSIS interfaces or 30 downstream and 120 upstream channels matching a large broadband service provider configuration where the impact of DOCSIS traffic is similarly minimized.

The performance for prewired communication (persistent network connections) scenarios improves both Tack$_N$ and Tcompletion$_N$ time across simulation and lab environments over not-prewired (non-persistent network connections or those created on-demand) scenarios. Table 7.7 compares prewired and not-prewired results taken from simulation and lab multicast-unicast experiments. The effect of prewired topologies is indicated in Tack$_N$ and Tcompletion$_N$ columns,

Environment	Pre-Wired	Msg Size	Texec$_N$ (ms)	Tack$_N$ (ms)	Tcompletion$_N$ (ms)
Simulation	Y	512B	.17	25	25
	Y	1M	280	303	303
	N	512B	.2	44	44
	N	1M	280	323	323
Lab	Y	512B	5	21	617
	Y	1MB	8652	8670	9264
	N	512B	10	74	670
	N	1MB	4313	4399	4981

Table 7.7: Comparing prewired and not-prewired Texec$_N$, Tack$_N$ and Tcompletion$_N$ time measurements under multicast-unicast, no-traffic scenarios for N = 128 nodes.

#Nodes	Msg Size	Texec$_N$ (ms)	Tack$_N$ (ms)	Tcompletion$_N$ (ms)
128	128B	24	46	46
8K	128B	392	549	551
128	1MB	2690	2711	2711
8K	1MB	124016	124036	124036

Table 7.8: Texec$_N$, Tack$_N$ and Tcompletion$_N$ time simulation measurements under unicast-unicast, not-prewired no-traffic scenario.

where time values are higher for not-prewired cases as compared to prewired for equivalent message sizes. This is expected as both Tack$_N$ and Tcompletion$_N$ communication flows are unicast upstream flows, where any elimination of TCP/IP 3-way handshakes reduces the overhead associated to socket operations as described earlier in Section 7.4.

In summary, the analysis of the multicast-unicast experimental results indicates that the key properties of multicast distribution are consistent between lab and simulation experiments. This holds true despite differences in absolute timing results comparing the lab to simulation environments. For unicast upstream communications, prewired connections improve the overall Tack$_N$ and Tcompletion$_N$ performance. The presence of background traffic within the lab and simulation experiments is not a significant factor due to the contention free DOCSIS router configuration in both environments.

Msg Size	$\text{Texec}_{8K}/\text{Texec}_{128}$ (ms)	$\text{Tack}_{8K}/\text{Tack}_{128}$ (ms)	$\text{Tcompletion}_{8K}/\text{Tcompletion}_{128}$ (ms)
128B	16	12	12
1M	46	46	46

Table 7.9: Texec_N, Tack_N and Tcompletion_N time ratios (case N=8K over N=128) under unicast-unicast, not-prewired no-traffic simulation scenario.

7.4.2.2 Process Launch-Execution Unicast-Unicast Results

Unicast-unicast scenarios consistently follow a linear performance curve for Texec_N, Tack_N, and Tcompletion_N timing measurements across all message sizes as N increases, in both simulation and lab experiments. Results are shown in appendix Figures A.5 through A.8 for simulation and Figures A.13 through A.16 for lab tests. A sample of measured values for Texec_N, Tack_N and Tcompletion_N for the scenario in Figure A.5 are listed in Table 7.8. A key result deduced from the figures and Table 7.8 is the significant performance cost of unicast message delivery across large numbers of device nodes for a given message size. Table 7.8 illustrates this across Texec_N, Tack_N and Tcompletion_N. Texec_N time for 128B communications increases from 24ms to 392ms, a 16 times increase. The efficiency decreases even further for larger message sizes. For example with 1MB message size, Texec_N increases from 2690ms to 124016ms, a 46 times increase. Results are comparable for each unicast communication as indicated in Table 7.8. The computed ratios comparing 8K to 128 nodes for each case are listed in Table 7.9.

Simulation results shown in appendix Figures A.5 through A.8 indicate at n = 4096 nodes, the presence of a positive inflection in the slope, where Texec_N, Tack_N and Tcompletion_N time durations increase faster above 4K nodes. This is due to DOCSIS contention effects from downstream and upstream TCP/IP acknowledgment traffic [90] when the topology size hits a threshold for the given CMTS model configuration. This effect is reduced in pre-wired scenarios Figures A.7 and A.8. Here, TCP/IP 3-way handshakes are minimized by virtue of pre-established persistent connections.

Appendix Figures A.5 through A.16 show the results for prewired versus not-prewired scenarios. Prewired scenarios across both lab and simulation outperform not-prewired scenarios as expected. In unicast-unicast scenarios, Tcompletion_N time is positively impacted when utilizing prewired connections as protocol overhead is reduced in each model unicast direction (Texec_N, Tack_N and

#Nodes	Msg Size	Prewired Tcompletion$_N$ (ms)	Not-prewired Tcompletion$_N$ (ms)	Improvement
8K	128B	243	484	2.0
8K	1MB	496	724	1.5
4K	128B	133	258	1.9
4K	1MB	392	490	1.3
2K	128B	80	145	1.8
2K	1MB	345	396	1.1
1K	128B	58	93	1.6
1K	1MB	329	360	1.1
512	128B	46	74	1.6
512	1MB	318	341	1.1
128	128B	23	45	1.9
128	1MB	302	322	1.1

Table 7.10: Comparing unicast-unicast prewired versus not-prewired Tcompletion$_N$ simulation performance results.

#Nodes	Msg Size	Texec$_N$ with traffic (ms)	Texec$_N$ without traffic (ms)
128	128B	25.4	24.3
128	1MB	2733	2690
8K	128B	396	392
8K	1MB	117981	124015

Table 7.11: Comparing unicast-unicast, not-prewired, Texec$_N$ time simulation measurements with and without traffic.

Tcompletion$_N$) for all DOCSIS devices nodes. Table 7.10 lists measured Tcompletion$_N$ values simulation comparing prewired to not-prewired. In each result, Tcompletion$_N$ time is lower for prewired test cases compared to not-prewired cases. We note that the improvement factor increases as the number of device nodes increases and, conversely, it decreases as message size increases. This result is consistent with the additional overhead cost incurred for TCP/IP session set-up relative to both the number of device nodes and a reduction in amortized cost of this overhead as the amount of data transmitted in increased.

The effect of traffic in all unicast-unicast experiments is small. For example, Table 7.11 illus-

trates the impact of traffic for not-prewired simulation scenario for Texec_N. Interesting, in the case of 8K nodes, the presence of traffic may actually lead to a small improved performance. This condition may occur when upstream traffic requests are randomly distributed in such a way that the DOCSIS back-off algorithm reduces contention. Under these conditions, the DOCSIS scheduler spreads out the upstream device-node requests over time such that contention effects are reduced and overall bandwidth across all device nodes is increased. The minor impact of traffic in these experiments is primarily due to production scale sizing of the DOCSIS CMTS model as configured in simulation. That is, there are sufficient CMTS line cards (6 5x20 interfaces) to support the upstream DOCSIS node transmission activity without high-contention due to traffic. Appendix Figures A.6 and A.8 illustrate this comparison graphically for the model time attribute.

In summarizing the unicast-unicast experimental results, a key property of unicast communication is the linear degradation in measured performance as we increase the size of the network and message length. Simulation and lab results are consistent. However lab results are further impacted by differences in experimental model attribute values such as $Tcompute_n$ *and overhead associated to MPI system initialization as discussed in multicast-unicast experiments. Similar to multicast-unicast scenarios, experimental results of prewired scenarios demonstrate positive performance impact, particularly as we increase the number of device nodes. The improvement rate decreases as the message size increases. In this experimental set-up, traffic generation was not a significant factor. In the next section, efficiency of multicast-unicast is compared to unicast-unicast communication flow scenarios.*

7.4.2.3 Comparing Process Launch-Execution Multicast Versus Unicast Results

In this section, multicast and unicast communication flows are compared relative to number of nodes, message size, delivery performance and efficiency. Results from experiments described in the previous section are utilized to evaluate when it is best to utilize multicast or unicast communication flows for message delivery.

Comparing multicast versus unicast message delivery performance and efficiency. Table 7.12 compares downstream multicast versus unicast message delivery performance in terms of Texec_N time, for message distribution to N nodes, where the same message is delivered to each node. For the same N, multicast, which has a constant time at nearly .07ms for 128B messages and

#Nodes	Msg Size	Multicast Texec$_N$ (ms)	Unicast Texec$_N$ (ms) Not-Prewired	Unicast Texec$_N$ (ms) Prewired
8K	128B	.07	392	92
8K	1MB	286	124016	99883
4K	128B	.07	209	42
4K	1MB	281	46038	46312
2K	128B	.07	121	21
2K	1MB	280	22257	22199
1K	128B	.07	72	10
1K	1MB	280	11174	11179
512	128B	.07	53	5
512	1MB	280	5730	5719
128	128B	.07	24	1.4
128	1MB	280	2690	2650

Table 7.12: Multicast versus unicast Texec$_N$ simulation performance results.

280-286ms for 1MB messages, is more efficient compared to unicast communications. Multicast Texec$_N$ times remain nearly constant for any fixed message size independent of N, whereas unicast increases linearly. Unicast prewired results indicate consistent improvement in comparison with not-prewired results in each case.

Results for multicast and unicast Texec$_N$ for N = 128 and 8K nodes versus message size are plotted in Fig. 7.11, illustrating multicast efficiency compared to unicast. Referring to the figure, results are consistent with the values from Table 7.12. Texec$_N$ duration versus message size curves (for N = 128 and 8K nodes) remain nearly constant (.07ms for 128B and 286ms for 1MB at 8K nodes) until message size grows to 1MB; unicast curve for N = 8K grows very quickly as the message size increases towards 1MB. For example, at N=128 nodes Texec$_N$ is 2690ms for not-prewired unicast case. Texec$_N$ increases to 124016ms at N=8K nodes; a 46X increase. For small N, unicast and multicast are comparable. Note that multicast efficiency actually increases as the number of nodes and the message size increases relative to unicast, as indicated by the widening Texec$_N$ gap between multicast and unicast for N = 8K nodes.

If each device node requires a unique message, how does multicast delivery compare to unicast?

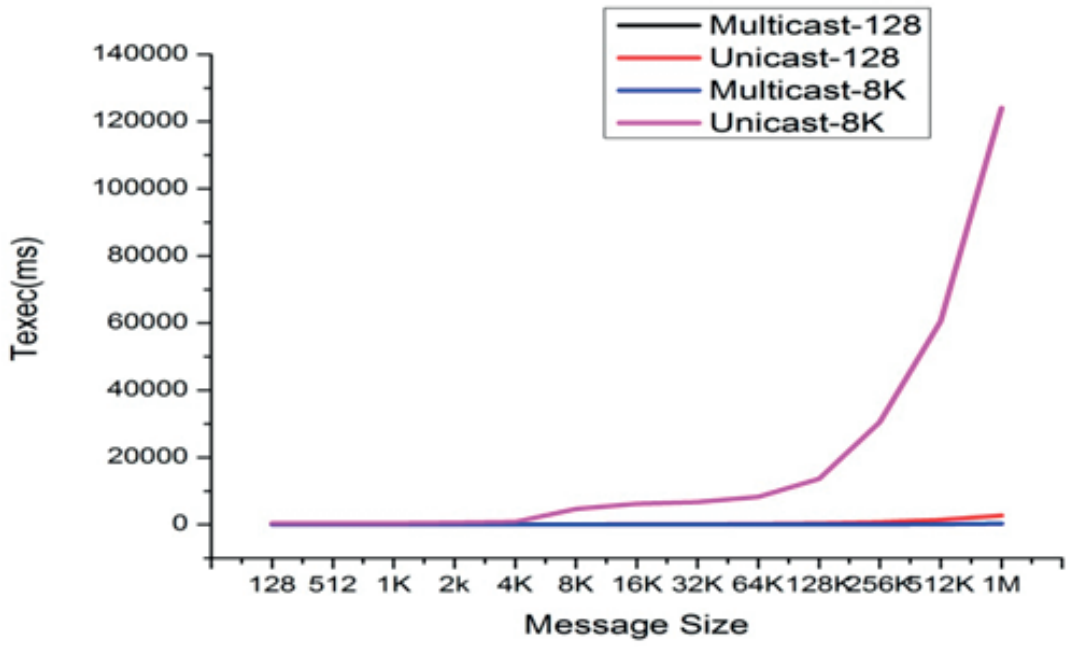


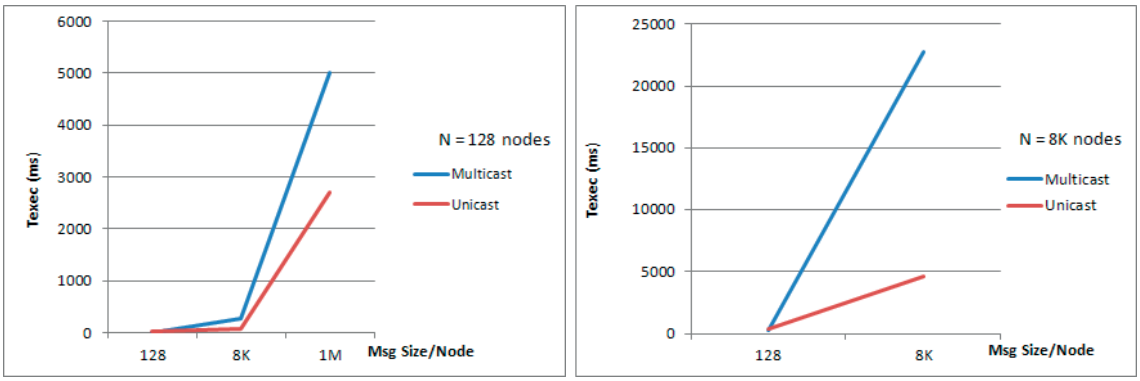
Figure 7.11: Multicast versus unicast delivery performance for various message sizes (where all messages are the same for all nodes).

Msg size/node	Multicast Texec$_N$ (ms)	Multicast Message Size	Unicast Texec$_N$ (ms)	Unicast Message Size
128	4.5	16KB	24	128B
8K	280	1MB	70	8K
1M	5000	128MB	2690	1MB

Table 7.13: Multicast versus unicast message delivery efficiency simulation for N = 128 nodes.

Msg size/node	Multicast Texec$_N$ (ms)	Multicast Message Size	Unicast Texec$_N$ (ms)	Unicast Message Size
128	287	1MB	392	128B
8K	22784	64MB	4619	8KB

Table 7.14: Multicast versus unicast message delivery efficiency simulation for N = 8K nodes.



(a) (b)

Figure 7.12: Multicast versus unicast delivery efficiency where each node receives a unique message.

Is it more efficient to send a single large multicast message that clients filter their unique message, compared to many smaller individual unicast messages? Simulation results in Tables 7.13 and 7.14 provide insight to answer these questions using Texec_N time to evaluate delivery efficiency. Results in Table 7.13 correspond as follows; column one is the effective unique message size per node for each client, evaluating 128B, 8K, and 1MB per node. The next two columns include values for multicast Texec_N time corresponding to the transmitted Texec_N message size in column three. That is, Texec_N message size is the total message delivered to 128 nodes, such that each node can filter a unique 128B, 8K, and 1MB message respectively. Table 7.14 provides similar information, but for N = 8K nodes. In analyzing both tables, the results indicate that only when the message size is small, does multicast outperform unicast for delivery of individual unique messages per device.

Fig. 7.12 further illustrates the benefit of utilizing unicast delivery for unique message delivery over multicast. The left-side figure shows the gap between multicast and unicast for 128 nodes, increasing once the per-device message size exceeds 4KB. For larger number of nodes, the gap occurs almost instantly with message sizes over 128B per node. This is illustrated on the right side of Fig. 7.12.

Comparing multicast versus unicast process completion performance. Texec_N performance data indicate that multicast downstream transmission is optimal where single-copy message delivery for large numbers of devices is required. Multicast-unicast offers an ideal solution for addressing large-scale process or application launch-execution within distributed broadband com-

#Nodes	Msg Size	Multicast-Unicast Tcompletion$_N$ (ms)	Unicast-Unicast Tcompletion$_N$(ms)	Performance Ratio
128	128B	45	46	1
8K	128B	484	551	1.1
128	1MB	322	2711	8.4
8K	1MB	724	124036	171

Table 7.15: Comparing multicast-unicast and unicast-unicast Tcompletion$_N$ simulation times under not-prewired no-traffic scenarios.

puting environments. However, utilizing multicast-unicast under this scenario also improves the overall completion time when compared to unicast-unicast flows where both message size and number of nodes increase (assuming equivalent Texec$_N$ message data and other model attributes such as Tcompute$_{avg}$ values for example). Simulation results for Tcompletion$_N$ in Table 7.15 validate and illustrate the overall improvement. For small message sizes there is negligible comparative improvement in utilizing multicast-unicast as performance ratios equal 1.0 and 1.1 respectively (message size is 128B). The Tcompletion$_N$ performance ratio for multicast-unicast versus unicast-unicast increases to 8.4 for 1MB message sizes when using N=128 device nodes. Reduction in Tcompletion$_N$ time, and hence overall process completion time, improves as the network increases in size. For example, referring to Table 7.15, at N=8K device nodes and 1MB message size, the performance ratio increases to 171; where Tcompletion$_N$ for the multicast-unicast scenario is 724ms compared to 124036ms for unicast-unicast scenario.

In summary, simulation results indicate a clear performance improvement when using multicast downstream data delivery compared to unicast delivery. This improvement is directly proportional to the number of device nodes and transmitted message size.

7.4.3 Parallel MSA Application Model

This section describes a second model that extends the process launch-execution model from Section 7.4.1 to enable simulation of the parallel MSA application execution from Section 4.5. To simulate parallel MSA, the algorithm from Section 4.5 is converted into an equivalent task-graph model representation. Fig. 7.13 illustrates the MSA task-graph model. Timing attributes Texec$_n$, Tack$_n$, Tcompletion$_n$ and their overall (worst case) case versions, Texec$_N$, Tack$_N$, and Tcompletion$_N$ are

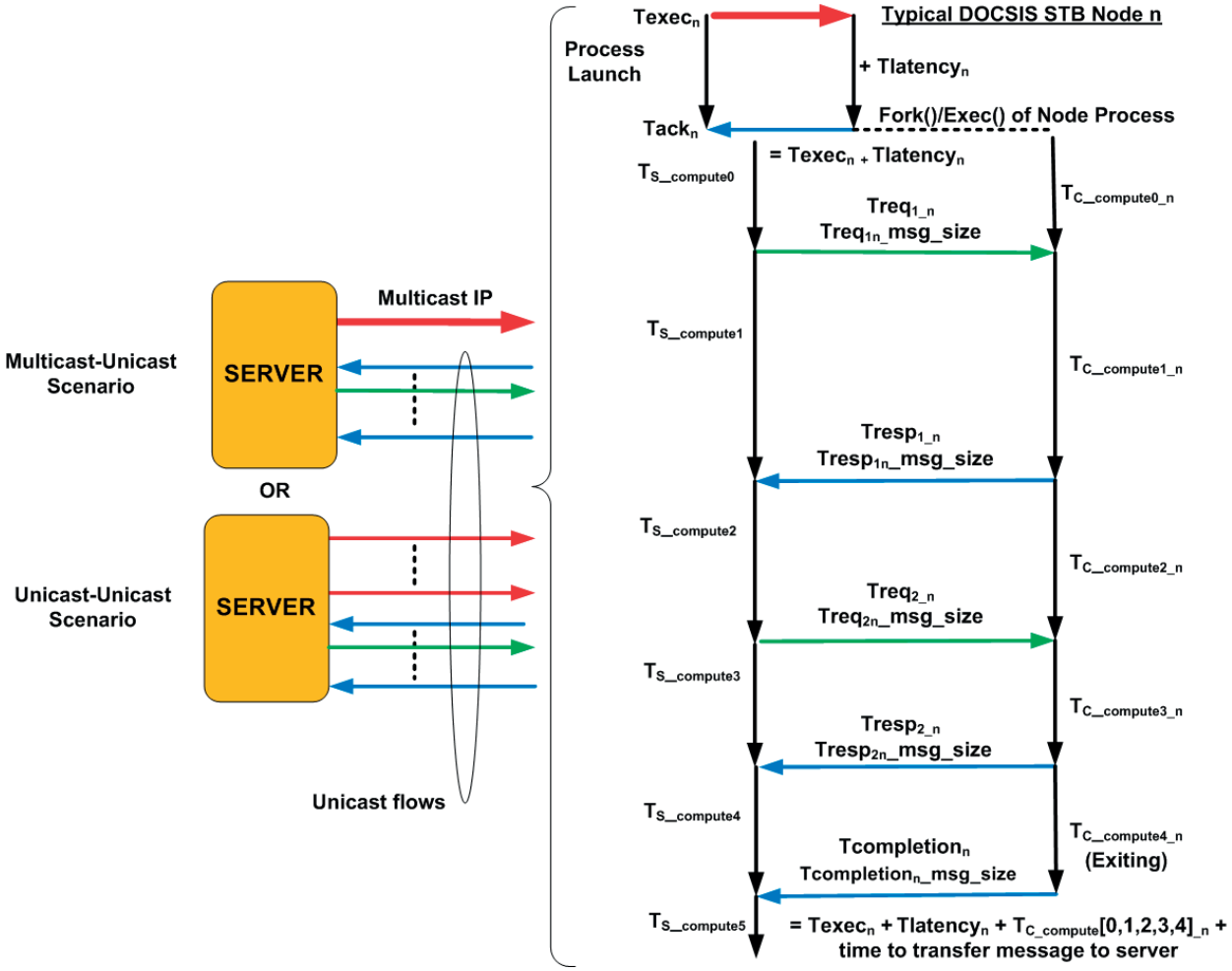


Figure 7.13: Parallel MSA application model.

defined as previously in Section 7.4.1. Tlatency is now captured for each device node $_n$ and is defined as Tlatency$_n$. The average value computed across all N device nodes is defined as Tlatency$_{avg}$. Model attributes shown in Fig. 7.13 capture additional computation and communication operations between the server and the N device nodes. Model notation and usage is as follows: Computation is modeled at the server as a sequence of 5 compute cycles, labeled T$_{S_compute\{1..5\}}$, with interleaving request and response message cycles between server and device nodes. In a similar fashion, device node computation is represented as a sequence of 4 compute cycles, labeled T$_{C_compute\{1..4\}_n}$, at the nth device node. Worst-case device node computation (i.e. the time required for all device nodes to complete a given computation cycle) across all device nodes is defined as T$_{C_compute\{1..4\}_N}$.

Communication flows from the server to individual device nodes are unicast and require 2 request message cycles labeled T_{req1_n} and T_{req2_n}, illustrated in green in Fig. 7.13. Communication flows from individual device nodes to the server are also unicast and require 2 response message cycles. These messages are labeled T_{resp1_n} and T_{resp2_n} respectively, illustrated in blue in Fig. 7.13. The parallel MSA algorithm includes multiple nested request and response communication cycles. In this analysis, all request and response communication timing results are reported as worst-case, or overall message communication times across all N device nodes. Worst-case overall request message cycles are defined as T_{req1_N} and T_{req2_N}. Similarly, overall time values for device-node-to-server response messages (worst-case response message operations time across all N device nodes to the server) are labeled and reported as T_{resp1_N} and T_{resp2_N}, respectively. In cases where individual request or response message times are reported, $T_{req1\_n\_avg}$ and $T_{req2\_n\_avg}$ indicate average time for server-to-device request message-transmissions. Similarly, $T_{resp1\_n\_avg}$ and $T_{resp2\_n\_avg}$ indicate average individual device-node-to-server message transmissions.

Request and response message sizes are defined as $T_{req1,2n\_msg\_size}$ and $T_{resp1,2n\_msg\_size}$ where the shorthand notation corresponds to the first and second request or response messages between the server and n^{th} device node, respectively. Finally, average request or response message sizes between server and device nodes across all nodes are defined as $T_{req1\_msg\_size\_avg}$, $T_{req2\_msg\_size\_avg}$ and $T_{resp1\_msg\_size\_avg}$ $T_{resp2\_msg\_size\_avg}$, respectively.

As mentioned earlier, a final completion message, $Tcompletion_n$, with a corresponding message size $Tcompletion_{n\_msg\_size}$, provides final acknowledgment for each device node process completion to the server node. The $Tcompletion_n$ message is sent from each device node after $T_{C\_compute4\_n}$ completes. The model execution terminates at the server at the end of $T_{S\_compute5}$ time. This final server-side computation is initiated after the server receives the final $Tcompletion_n$ message or $Tcompletion_N$ (worst case or overall reported time). This final computation at the server marks the completion of the overall model execution cycle.

Fig. 7.14 illustrates the mapping for the MSA task-graph model in Fig. 7.13 to the master-worker computational pattern representing parallel MSA as described in Section 4.5. The algorithm is based on the MASON implementation of CLUSTALW for execution on distributed memory parallel systems using MPI [26; 32]. The parallel MSA algorithm and corresponding task-graph model representation proceed through nine steps after initialization as follows:

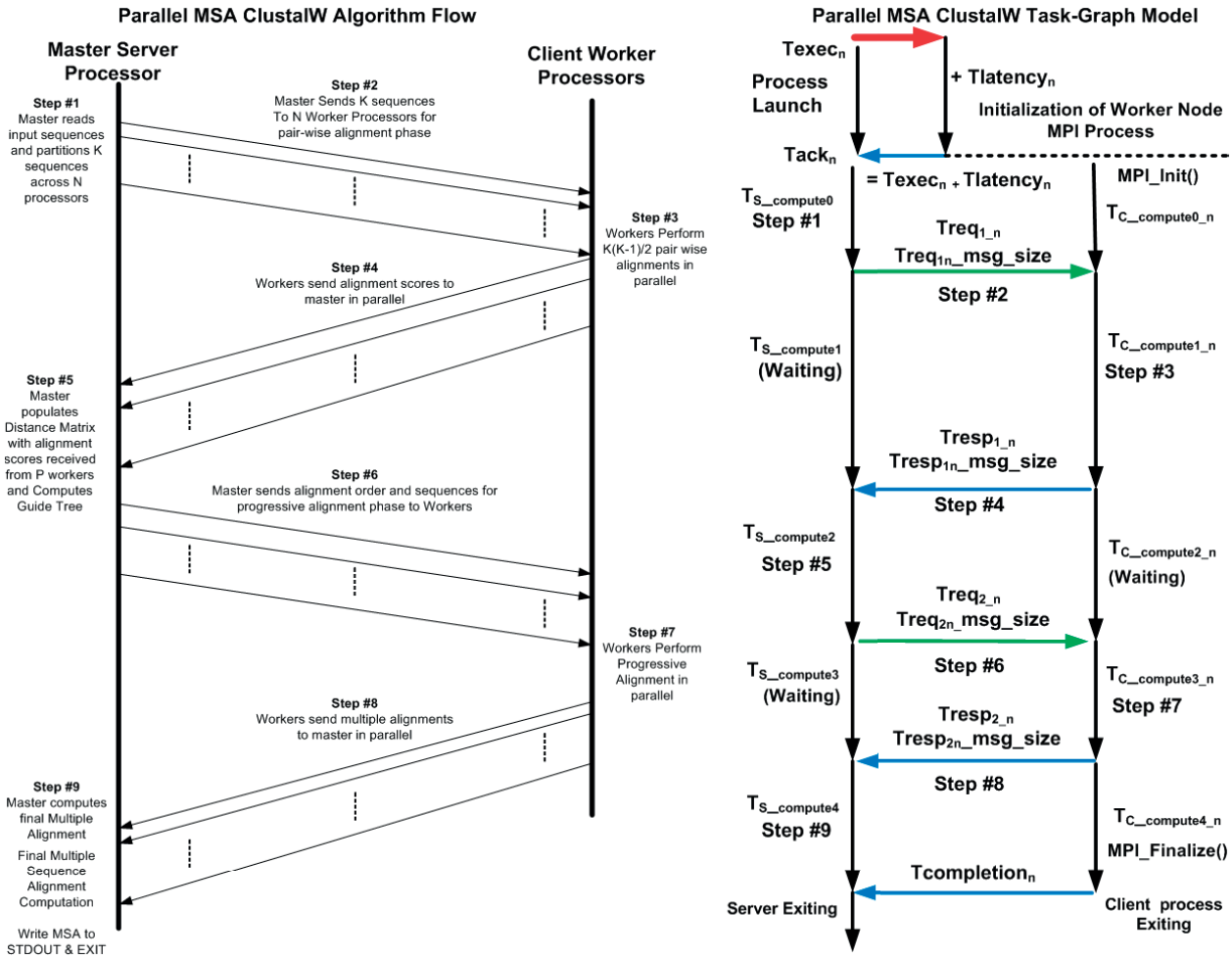


Figure 7.14: Parallel ClustalW MSA algorithm comparison to Parallel MSA application model.

1. The master host processor reads the input file containing K sequences and allocates a $K \times K$ distance matrix structure that is used to hold all optimal alignment scores between all permutations of K sequences. The master also partitions the distance matrix by dividing up the sequences among the N worker processors (device nodes) to distribute the workload, network, and resource requirements during the alignment step. This step corresponds or takes a time equal to $\mathrm{T}_{S\_compute1}$ in the MSA model task-graph.

2. The master sends all required sequences to the N worker processors based on the distance matrix partitioning scheme computed in Step 1. Each processor has a fraction $K_f = K/N$

of the total number of K sequences to align. The overall communication cycle time required from server to all N device nodes is T_{req1_N}.

3. The N worker processors compute their $\frac{K_f \cdot (K_f - 1)}{2}$ alignments in parallel using the pairwise alignment dynamic programming algorithm. This step corresponds to $T_{C\_compute1\_n}$ for each client. Pairwise computation for all N workers must complete before proceeding to the next MSA algorithm step. Worst-case pairwise completion time across all N device nodes is $T_{C\_compute1\_N}$. Since over 96% (see Section 4.5) of the MSA algorithm is spent in this computation step, $T_{C\_compute1\_N}$ has an important impact on overall parallel MSA performance.

4. All worker processors send their alignment results back to the master in parallel. The overall communication cycle time required from all N devices nodes to the server is T_{resp1_N}. The average message size sent from device nodes to the server is $T_{resp1\_msg\_size\_avg}$.

5. After receiving all scores and completing the distance matrix the master builds the guide tree. The MSA application model represents this computational step as $T_{S\_compute2}$.

6. The master computes and sends an alignment order along with the respective sequences required for progressive alignment to all worker processors. The overall communication cycle time required from server to all N device nodes is T_{req2_N}. The average message size sent from the server to all N device nodes is $T_{req2\_msg\_size\_avg}$.

7. All worker processors perform progressive alignment in parallel. This computation step occurs at or takes $T_{C\_compute3\_n}$ in the MSA application model.

8. All worker processors send their partial multiple alignments back to the master in parallel. This message cycle corresponds to T_{resp2_N}. The average message size sent from all N device nodes to the server is $T_{resp2\_msg\_size\_avg}$. After this step, each device node completes its execution during the final computation stage $T_{C\_compute4\_n}$. For lab experiments, this phase corresponds to MPI finalization.

9. The master progressively completes the remaining multiple alignment pairs to produce the final result. This final server computation step is defined as $T_{S\_compute4}$.

Attribute	Simulation	Lab
#Nodes	128,512,1K,2K,4K	32,64,128
Texec$_n$ transmission message size (Bytes)	87078	87078
Tack$_n$ response message size (Bytes)	128	128
Tcompletion$_n$ response message size (Bytes)	1K	1K

Table 7.16: MSA model message size attribute settings.

The server and device node initialization time is not included in the MSA algorithm execution time. However, the initialization time duration is captured in the MSA application model as $T_{S\_compute0}$ and $T_{C\_compute0\_n}$ computation phases respectively. Additionally, some computation phases represent wait states durations, typically while some communication tasks complete on the server or device nodes. In these cases, phases are indicated in Fig. 7.14 as *Waiting*. Model termination occurs at the conclusion of the final server computation in Step 9. In this final phase, the server waits for a Tcompletion$_N$ message indicating that all device nodes have sent their final message. At this point the MSA application model has completed a full execution cycle.

7.4.4 Parallel MSA Application Model Experiments

This section describes experiments to evaluate broadband embedded computation and communication performance trade-offs by analyzing the execution of the parallel MSA algorithm across lab and simulation environments. The application model described in Section 7.4.3 supports parallel MSA simulation, whereas the lab system executes the parallel ClustalW MPI MSA application. Experiments are performed across lab and simulation environments to confirm the accuracy of simulation in comparison to results derived from the lab system. Simulation is used to evaluate large-scale application launch performance and broadband network scalability. Results are then utilized in simulation to examine the scaling potential of broadband embedded computing for executing distributed parallel computing applications across large-scale broadband network infrastructures. The experimental environment is based on the same simulation and lab system configuration described in Section 7.3.

The experiments consist of 4 parallel MSA simulation test scenarios and 4 parallel MSA lab test scenarios using 7 DNA input sequences for each test. Seven sequences are exactly the same

as those defined in Section 4.5. A sequence is defined by the length and base pair encoding *Sx-Ly*, that denotes a data set of x DNA sequences with y base pairs (e.g. *S100-L1500* means "100 DNA Sequences with 1500 base pairs"). Average sequence length is denoted L_{avg}. Simulation and lab system experiments consist of tests to measure the performance impact of multicast-unicast and unicast-unicast DOCSIS communication flows under traffic and no-traffic test scenarios. Therefore, simulation and lab experimental scenarios include a total of 4 test combinations as follows: 1) multicast-unicast with and without traffic, and 2) unicast-unicast with and without traffic. In both simulation and lab experiments, each test is repeated 3 times and the average results reported.

Parallel MSA application model attributes are defined in Section 7.4.3. Values for $Texec_n$, $Tack_n$ and $Tcompletion_n$ message size attributes are defined in Table 7.16. $Texec_n$ message size is fixed and corresponds to the actual ClustalW MPI application binary image size. $Tcompletion_N$ defines the completion of the parallel MSA model execution corresponding to the MSA sequence generation. Simulation experiments make use of lab results including computation and communication message sizes measured from lab tests. Measurements from lab execution are averaged, then utilized to predict important attribute values for various N. The predicted attribute values are then back-annotated to the simulation scenario model attributes corresponding to the equivalent model attributes in Fig. 7.13. To evaluate the impact of DOCSIS network performance in comparison to device node computation, communication and computation results from lab execution and simulation are summarized in the following sections.

7.4.4.1 Parallel MSA Launch-Execution and Completion Time Lab Results

Results for $Texec_N$ and $Tcompletion_N$ from parallel MSA application model lab execution across 128, 64 and 32 device nodes are shown in Tables 7.17 and 7.18 respectively. Table 7.17 presents $Texec_N$ and $Tcompletion_N$ time measurements for multicast-unicast and unicast-unicast scenarios under the no-traffic scenario. Table 7.18 presents the results for the same set of experiments executed in the presence of traffic.

$Texec_N$ results are consistent and uniform averaging 369ms multicast delivery time performance as the number of device nodes is increased from 32 to 128. Averaging 680ms across all experiments, the delivery time $Texec_N$ for unicast is nearly twice as long as the delivery time for multicast. This confirms the results of earlier process launch-execution experiments from Section 7.4.2.1, where

#Nodes	Input Sequence	Multicast-Unicast $Texec_N$ (ms)	Multicast-Unicast $Tcompletion_N$ (ms)	Unicast-Unicast $Texec_N$ (ms)	Unicast-Unicast $Tcompletion_N$ (ms)
128	S100-L1500	370	390512	633	391275
	S200-L300	369	110776	629	112631
	S300-L200	369	44123	624	44721
	S500-L200	368	207330	624	207979
	S200-L500	368	154016	620	154862
	S500-L1100	368	2327106	626	2327044
	S1500-L100	368	1267784	621	1276006
64	S100-L1500	370	558804	613	561195
	S200-L300	369	209471	623	212313
	S300-L200	369	65967	618	65037
	S500-L200	368	378900	615	381729
	S200-L500	368	258018	633	258834
	S500-L1100	369	4323279	619	4319816
	S1500-L100	369	1828766	608	1835764
32	S100-L1500	371	1035962	623	1037729
	S200-L300	370	277618	638	278815
	S300-L200	369	101951	619	103218
	S500-L200	368	592751	653	589761
	S200-L500	368	464835	605	465778
	S500-L1100	370	6681769	636	6677896
	S1500-L100	370	2287596	612	2278596

Table 7.17: Parallel MSA Lab results for $Texec_N$ and $Tcompletion_N$ no-traffic scenario.

multicast message delivery outperformed unicast message delivery for any given message size. Lab results also confirm that multicast delivery provides higher performance (less time) for parallel MSA application delivery compared to the unicast based approach. The $Texec_N$ variation across N (128, 64, and 32 device nodes) for either multicast-unicast or unicast-unicast is small at less than 1%. Indeed, as a consequence of the small lab network environment, measured results are similar for both traffic and non-traffic scenarios. The average measured $Tlatency_{avg}$ across all multicast-unicast tests is 41ms. However, the average measured $Tlatency_{avg}$ increases to 57ms for unicast-unicast tests due to added device node network I/O overhead associated to unicast operations. Specifically, additional time is required within the lab STB test device nodes during socket open and close operations in the unicast-unicast scenario.

#Nodes	Input Sequence	Multicast-Unicast $Texec_N$ (ms)	Multicast-Unicast $Tcompletion_N$ (ms)	Unicast-Unicast $Texec_N$ (ms)	Unicast-Unicast $Tcompletion_N$ (ms)
128	S100-L1500	370	389292	680	392737
	S200-L300	370	110926	685	112816
	S300-L200	369	40845	672	41894
	S500-L200	369	207088	656	208738
	S200-L500	368	153384	678	155165
	S500-L1100	368	2327639	687	2328387
	S1500-L100	369	1264625	677	1267905
64	S100-L1500	370	558731	661	565196
	S200-L300	370	209616	686	211724
	S300-L200	371	64595	692	66877
	S500-L200	371	379358	680	381902
	S200-L500	371	259292	675	259867
	S500-L1100	372	4321019	673	4323933
	S1500-L100	371	1837264	672	1836102
32	S100-L1500	371	1036113	695	1037047
	S200-L300	370	277788	677	279052
	S300-L200	370	102553	701	105391
	S500-L200	370	585142	677	592398
	S200-L500	369	466109	664	467842
	S500-L1100	371	6673171	676	6678858
	S1500-L100	370	2289438	702	2292539

Table 7.18: Parallel MSA Lab results for $Texec_N$ and $Tcompletion_N$ traffic scenario.

$Tcompletion_N$ times are consistent across all experiments with $Tcompletion_N$ decreasing (performance improving nearly linearly) as the number of device nodes increases. $Tcompletion_N$ results in Tables 7.17 and 7.18 show a small improvement for multicast-unicast cases over unicast-unicast results. This improvement is primarily due to application launch-execution efficiency impact from lower $Texec_N$, $Tlatency_{avg}$ values and internal device node overhead.

Results are shown in appendix Figures A.17 through A.20 with additional results for $Tack_N$ included. As expected, $Tack_N$ values are higher compared to $Texec_N$ yet uniform across all N. In the traffic scenario test case, results indicate a linear increase in $Tack_N$ at N = 128 nodes. This is consistent with the previous results from Section 7.4.2.2 where $Tack_N$ as unicast operation begins

Input Sequence	Multicast-Unicast Speedup	Unicast-Unicast Speedup
S100-L1500	2.7/2.7	2.7/2.6
S200-L300	2.5/2.5	2.5/2.5
S300-L200	2.3/2.5	2.3/2.5
S500-L200	2.9/2.8	2.8/2.8
S200-L500	3.0/3.0	3.0/3.0
S500-L1100	2.9/2.9	2.9/2.9
S1500-L100	1.8/1.8	1.8/1.8

Table 7.19: Lab speedup results for multicast-unicast and unicast-unicast with no-traffic/traffic scenarios.

to increase linearly as the number of device nodes increases with slightly higher Tack_N values under lab traffic scenarios.

7.4.4.2 Parallel MSA Execution Time Speedup Lab Results

Parallel MSA execution speedup results for both traffic and no-traffic lab tests are shown in Table 7.19 (shown as no-traffic/traffic values). Results show a near linear speedup as the number of device nodes increases from 32 to 128. Parallelization speedup is computed as the ratio of measured Tcompletion_N values using 32 device nodes (or Tcompletion_{32}) divided by Tcompletion_N measurements using 128 device nodes (or Tcompletion_{128}). Speedup is comparable in all cases and varies between 1.8 and 3.0. Multicast-unicast speedup is comparable and consistent to unicast-unicast speedup measurements from the experiments in Section 4.5. Similarly there is a small difference in results due to traffic between multicast-unicast and unicast-unicast test scenarios. This is consistent with the small difference of less than 1% variation in Tcompletion_N values presented in Section 7.4.4.1. This small difference in results is due to the parallel MSA application implementation, which utilizes unicast communication flows (between server and device nodes as illustrated in Fig. 7.14) during program execution. This comprises the majority of parallel MSA execution time.

7.4.4.3 Parallel MSA Communications Lab Results

Lab measurements for overall communication cycle time results are shown in Tables 7.20 and 7.21. All attribute values in Tables 7.20 and 7.21 correspond to the parallel MSA task-graph model attributes described in Section 7.4.3. Lab results are shown for 128, 64, and 32 device nodes for

#Nodes	Input Sequence	T_{req1_N} (ms)	T_{resp1_N} (ms)	T_{req2_N} (ms)	T_{resp2_N} (ms)
128	S100-L1500	1140	98108	45	86803
	S200-L300	1185	55521	54	17403
	S300-L200	1136	14741	87	8190
	S500-L200	1195	81967	82	10852
	S200-L500	1556	48650	99	21785
	S500-L1100	1347	992582	89	189224
	S1500-L100	1262	243294	185	44622
64	S100-L1500	698	112116	45	85176
	S200-L300	675	123393	53	17301
	S300-L200	659	24404	49	7050
	S500-L200	687	175341	72	11044
	S200-L500	681	80255	98	21627
	S500-L1100	849	1924671	70	193066
	S1500-L100	787	574168	135	38604
32	S100-L1500	394	255615	44	88916
	S200-L300	380	128884	42	18029
	S300-L200	382	35148	44	4903
	S500-L200	417	228577	54	17326
	S200-L500	368	143938	47	22729
	S500-L1100	543	2188748	77	206722
	S1500-L100	509	544015	117	38878

Table 7.20: Parallel MSA Lab overall request-response communication cycle time results.

each input sequence, and executed under the multicast-unicast, no-traffic scenario. Table 7.20 lists overall communication cycle time measurements (T_{req1_N} and T_{req2_N}) corresponding to request cycles from server to device nodes across all sequence tests. Overall response message cycle times (T_{resp1_N} and T_{resp2_N}) from devices nodes to the server are also shown in Table 7.20. Corresponding message sizes associated to each of the request and response message phases are shown in Table 7.21.

Results show that T_{req1_N} increases by an average factor of 2.7X and $T_{req1\_msg\_size\_avg}$ message sizes decrease an average of 2.0X, as the number of device nodes varies between 32 to 128. As an illustration, results for S100-L1500 taken from Tables 7.20 and 7.21 show T_{req1_N} increasing from 394ms to 1140ms as the number of device nodes increases from 32 to 128. Correspondingly,

$T_{req1\_msg\_size\_avg}$ decreases from 36859 to 18834 bytes. T_{req1_N} time increases are due to the time required by the server node to distribute partitioned sequence data over an increasing number of unicast connections between server and devices nodes as N increases. These results are consistent with the unicast-unicast process launch-execution results from Section 7.4.2.2 where transmission time increased linearly with N for a given message size. Results for T_{req2_N} follow a similar trend as T_{req1_N}. However, T_{req2_N} increases by a factor of only 1.3X due to greater variability in $T_{req2\_msg\_size\_avg}$ message sizes across all sequence tests. For example, $T_{req2\_msg\_size\_avg}$ message sizes decrease by only a factor of 1.1X for S100-L1500. In comparison, $T_{req2\_msg\_size\_avg}$ message sizes decrease by a factor of 4.0X for S1500-L100. Results from Tables 7.20 and 7.21 confirm that T_{req2_N} values are impacted by message-size variation across input sequence tests. This is due to the greater communication cost impact from message size increases relative to the impact from increasing the number of lab device nodes from 32 to 128.

Response message timing results for T_{resp1_N} and T_{resp2_N} are shown in Table 7.20. Both T_{resp1_N} and T_{resp2_N} generally increase as the number of device nodes decreases from 128 to 32 but at different rates. To illustrate, values for T_{resp1_N} increase by a factor of 4.0X as N decreases for test input S1500-L100 and 3.3X for test input S100-L1500. In comparison, T_{resp2_N} values increase by only a factor of 1.1X across N for S100-L1500 and 4.0X for sequence test S1500-L100. Except for the possible effects of DOCSIS upstream communication, the behavior of T_{resp1_N} and T_{resp2_N} is similar to T_{req1_N} and T_{req2_N} where variation in message sizes impact transmission performance. Specifically, the combination of message size and DOCSIS upstream contention effects (both influenced by message size and the number of device nodes) are contributing factors to the performance of the communication from device node to server. The effect of contention in downstream server to device node request messages is not a significant factor as discussed in Section 2.4.1.1. As the number of nodes are increased, message sizes generally decrease, because the total data-set is partitioned across additional device nodes. However, the effects of DOCSIS upstream contention also increases. These effects can be observed from results in Table 7.20 and 7.21 using S100-L1500 as an example: T_{resp1_N} increases from 98108ms to 255615ms or a factor of 2.6X as we decrease N from 128 to 32 device nodes. At the same time, $T_{resp1\_msg\_size\_avg}$ message sizes increase by a factor of 3.3X (223 bytes to 728 bytes). In comparison, T_{resp2_N} increases only from 86803ms to 88916ms or a factor of 1.02 while $T_{resp2\_msg\_size\_avg}$ message sizes increases by only

#Nodes	Input Sequence	$T_{req1\_msg\_size\_avg}$ (Bytes)	$T_{resp1\_msg\_size\_avg}$ (Bytes)	$T_{req2\_msg\_size\_avg}$ (Bytes)	$T_{resp2\_msg\_size\_avg}$ (Bytes)
128	S100-L1500	18834	224	14296	43948
	S200-L300	8298	728	5938	17153
	S300-L200	5721	1224	2566	7550
	S500-L200	13572	3968	6139	16987
	S200-L500	12902	624	6661	18209
	S500-L1100	47871	4224	25551	66741
	S1500-L100	25036	32040	19887	52674
64	S100-L1500	26636	360	14296	43948
	S200-L300	11310	1520	6582	19008
	S300-L200	7999	2400	4123	12121
	S500-L200	18550	8280	12255	33892
	S200-L500	17122	1368	7901	21589
	S500-L1100	66088	8648	50998	133137
	S1500-L100	33903	68120	39609	104893
32	S100-L1500	36859	728	16044	49246
	S200-L300	16871	2600	13142	37875
	S300-L200	11492	4488	8186	24060
	S500-L200	26928	15128	24364	67287
	S200-L500	25296	2400	15706	42937
	S500-L1100	95114	16128	101586	264653
	S1500-L100	48547	127448	78620	208018

Table 7.21: Parallel MSA Lab execution request-response message size results.

a factor of 1.1. In the case of T_{resp1_N}, both message size and upstream contention impact performance. However, since $T_{resp1\_msg\_size\_avg}$ message sizes are smaller compared to $T_{resp2\_msg\_size\_avg}$, upstream contention is the more significant factor. In comparison, since $T_{resp2\_msg\_size\_avg}$ message sizes are much larger, the effect of message size on T_{resp2_N} is greater on overall performance, even though the relative difference in message size changes is small across 128 and 32 nodes.

Parallel MSA request-response message size lab results. Table 7.21 shows the message size results associated to each Parallel MSA unicast communication operation shown in Table 7.20. For any given test input consisting of K sequences, the total amount of message data transferred during T_{req1_N} cycles, is approximately equal to the number of K sequences multiplied by average sequence length. As mentioned, results show that message size variations follow similar trends to

Input Sequence	$T_{C\_compute1\_32}$ (ms)	$T_{C\_compute1\_64}$ (ms)	$T_{C\_compute1\_128}$ (ms)
S100-L1500	928898	455482	285416
S200-L300	254509	186764	87180
S300-L200	90866	52315	28355
S500-L200	541050	335122	225791
S200-L500	432893	226585	121608
S500-L1100	6400948	4058831	2067877
S1500-L100	1536862	1070978	501753

Table 7.22: Parallel MSA lab results for alignment computation phase.

message delivery time measurements in Table 7.20. As the number of nodes increases from 32 to 128, $T_{req1\_msg\_size\_avg}$ size generally decreases, since the message data is partitioned and spread across more available device nodes. $T_{req1\_msg\_size\_avg}$ is also directly proportional to the size and length of the test input sequence data, delivered to device nodes during the T_{req1_N} communication phase. Both T_{req1_N} and $T_{req1\_msg\_size\_avg}$ are higher for sequence inputs that either contain a larger number of sequences (S1500-L100), or have a longer sequence length (S500-L1100). In comparison, test inputs that contain less sequences or sequences that are shorter in length (S300-L200) exhibit lower T_{req1_N} values and smaller $T_{req1\_msg\_size\_avg}$ message sizes. Results for $T_{req2\_msg\_size\_avg}$ follow a similar trend as $T_{req1\_msg\_size\_avg}$. The $T_{req2\_msg\_size\_avg}$ message sizes increase as we decrease the number of device nodes.

To summarize, consistent with results from process launch-execution experiments, results of executing parallel MSA across 128 STB device nodes confirm the trend that unicast communication flows follow a linear performance curve as the number of device nodes and message sizes increase. The actual performance achieved is a function of the number of device nodes and associated message sizes, that in the case of MSA, are determined by the size of the input sequence.

7.4.4.4 Parallel MSA Computation Time Lab Results

Table 7.22 shows parallel MSA computation results for $T_{C\_compute1\_N}$ corresponding to the time required for the overall alignment computation phase for N = 32, 64, and 128 device nodes, respectively. Results are for the multicast-unicast, no-traffic lab scenario. Since 96% of all parallel MSA computation (see Section 4.5) is spent in Step 3 (pairwise alignment computation phase), only results for $T_{C\_compute1\_N}$ are reported. The remaining parallel MSA model computation at-

Rank	Input Sequence	$(L_{avg})^2$ (Bytes)	#Alignments per-Device Node	Computation Time per-Alignment (ms/Bytes^2)	$T_{C\_compute1\_32}$ (ms)
1	S500-L1100	615597	4032	.0028	6400948
2	S1500-L100	18851	31862	.0028	1536862
3	S100-L1500	2473700	182	.0023	928898
4	S500-L200	49240	3782	.0031	541050
5	S200-L500	287725	600	.0027	432893
6	S200-L300	116896	650	.0037	254509
7	S300-L200	29998	1122	.0029	90866

Table 7.23: Alignment computation values for N = 32 device nodes.

tributes and corresponding results do not significantly impact the completion time results. The measurements in Table 7.22 are ranked from highest to lowest $T_{C\_compute1\_N}$ values. Table 7.22 also includes a column labeled ${L_{avg}}^2$, containing values of average sequence length squared. ${L_{avg}}^2$ is a measure of the average number of operations required to align two sequences and corresponding data structure size.

As the number of device nodes varies from 32 to 128, the compute time decreases by 3.3 times for the longest sequence input test (S100-L1500) and 3.1 times for the sequence input test (S1500-L100) with the largest number of sequences. The performance improvement is due to the distribution of computational work across additional device nodes, each performing a smaller subset of the total MSA computation. As the input test is varied, either in number of sequences or their length, there is a corresponding impact on the total computation required for performing MSA. This is observed in the ranking of sequence test inputs by $T_{C\_compute1\_N}$ results, where the three highest $T_{C\_compute1\_N}$ values are associated to sequence inputs S500-L1100, S1500-L100, and S100-L1500 respectively. The lowest two $T_{C\_compute1\_N}$ values are associated to sequence inputs S200-L300 and S300-L200 respectively. Results show that computational time increases in direct relation to the combination of total number of sequences and their average length L_{avg}. As the number of sequences increase for a given test input, the table shows the number of pairwise alignments assigned to each node also increases. The number of sequences for each device node to process is directly proportional to $\frac{K_f \cdot (K_f - 1)}{2}$, or approximately the square of the fraction of total sequences K_f assigned to a given device node. $T_{C\_compute1\_N}$ values are highest for test inputs with the

largest number of sequences, except for those cases where $L_{avg}{}^2$ (average sequence length squared) are highest. The computational sensitivity to L_{avg} is due to the MSA algorithm itself, where each pairwise alignment computation requires $L_{avg}{}^2$ operations.

Table 7.22 lists the average computation time required for a device node to process a single alignment operation between two sequences. Results were computed using N = 32 device nodes to evaluate computation under worst-case conditions since under this scenario each device node is required to complete the largest number of computations for each sequence test input. Alignment computation results are computed by dividing the measured $T_{C\_compute1\_N}$ (or total alignment computation time for all sequences assigned to the device node) by the product of $L_{avg}{}^2$ and number of alignments assigned per device node. This product is a measure of the total *work* performed by a given device node. The average alignment compute time, defined C_A, across all test inputs is 0.0027ms/Bytes^2 with 0.0014 ms/Bytes^2 variation between highest and lowest measured values. This result is consistent across all sequence test inputs, and is due to the reciprocal relationship exhibited among the sequence test inputs; where those with larger values for L_{avg} (more work) also contain less number of sequences in the test input (less work).

In summary, lab results from parallel MSA testing confirm the benefit of multicast delivery for distributed application launch compared to unicast based application delivery, validating the generalized process launch-execution model results presented in Section 7.4.2.2. Parallel MSA speedup is consistent with earlier lab experiments from Section 4.5, where results show comparable MSA completion time performance improvement as the number of device nodes is increased. Measurements of device node alignment computation show that overall MSA execution time is sensitive to both the number of test input sequences and their average length.

7.4.4.5 Large-Scale Parallel MSA Simulation

This section describes the methodology used to configure OPNET parallel MSA model attributes for simulating parallel MSA execution across a large-scale broadband network of up to 4K device nodes. The parallel MSA model from Fig. 7.13 requires server and device node attribute definitions prior to simulation. Lab measurements from Section 7.4.4.3 and 7.4.4.4 are utilized to define both server and device node model attribute values. Parallel MSA server model attribute values are dependent on sequence test inputs and the number of device nodes. Since parallel MSA server

#Nodes	Input Sequence	#Alignments per-Device Node	Predicted $T_{C\_compute1\_N}$ (ms)	Predicted $T_{req1\_msg\_size\_avg}$ (Bytes)	Predicted $T_{resp1\_msg\_size\_avg}$ (Bytes)
128	S100-L1500	56	369333	22019	224
	S200-L300	182	56722	8889	728
	S300-L200	306	24474	5889	1224
	S500-L200	992	130230	13758	3968
	S200-L500	156	119670	12874	624
	S500-L1100	1056	1733182	50214	4224
	S1500-L100	8010	402585	24439	32040
4K	S100-L1500	6	39571	6291	24
	S200-L300	12	3740	2051	48
	S300-L200	12	960	1039	48
	S500-L200	42	5514	2663	168
	S200-L500	12	9205	3218	48
	S500-L1100	42	68933	9415	168
	S1500-L100	272	13671	4394	1088

Table 7.24: Predicted MSA simulation model attribute values for N = 128 and 4K.

computation is a small percentage (less than 4%) of the total parallel MSA completion time, server model attributes are configured using measured 128 device node lab results directly.

Parallel MSA simulation attributes for device nodes contain both fixed and variable attribute values. Fixed attributes contain static values that are independent of the number of sequences or device nodes, and do not change during simulation. Variable attributes contain values that depend on both sequence test input data, and the number of device nodes N. Fixed attribute values associated with parallel MSA process-launch and execution are defined in Table 7.16. Variable attributes associated with parallel MSA algorithm computation at the device node and communications between server and device nodes include: $T_{C\_compute1\_N}$, $T_{req1\_msg\_size\_avg}$, $T_{resp1\_msg\_size\_avg}$, $T_{req2\_msg\_size\_avg}$, and $T_{resp2\_msg\_size\_avg}$. A simple predictive model is developed to define values for each of these computation and communication attributes.

Predicting parallel MSA computation. Parallel MSA device node computation is predicted based on the lab results from Sections 7.4.4.3 and 7.4.4.4 as well as results from Datta [26; 32]. These results show that the majority of parallel MSA computation occurs during parallel MSA algorithm Steps 3 corresponding to $T_{C\_compute1\_N}$ time. Worst-case computation for each

device node is therefore predicted as $T_{C\_compute1\_N} = C_A \times {L_{avg}}^2 \times$ #alignments per device node, where C_A is defined as the average computation required per alignment operation in milliseconds. From Section 7.4.4.4, C_A is calculated and equal to .0027 for N = 32 device nodes. The alignment computation factor C_A can be thought of as the approximate time required for a device node to compute a single pairwise alignment. ${L_{avg}}^2$ defined previously in Section 7.4.4.4 refers to the average input sequence length squared. When ${L_{avg}}^2$ is multiplied by the number of alignments per device node, the product ${L_{avg}}^2 \times$ #alignments represents the total amount of computational work to perform on the given device node. Computational accuracy is compared with lab measurements for N = 32 nodes. Average $T_{C\_compute1\_N}$ accuracy is within $\pm 10\%$ across all test inputs for the predicted simulation model values versus the actual lab measurements.

Predicting parallel MSA communications. In addition to device node computation, simulation attribute values for parallel MSA request and response message sizes must be configured for each experimental scenario. With communication message sizes defined, simulation can generate individual request timing results for $T_{req1\_n\_avg}$ and $T_{req2\_n\_avg}$ as well as overall forms T_{req1_N} and T_{req2_N} defined in Section 7.4.3. Recall from Section 7.4.3 that the overall timing result refers to the time duration for a complete communication request or response cycle across N device nodes. Similarly, individual response timing results for $T_{resp1\_n\_avg}$ and $T_{resp2\_n\_avg}$, along with overall forms T_{resp1_N} and T_{resp2_N} respectively, are obtained through simulation. Values for the MSA communication attribute $T_{req1\_msg\_size\_avg}$ are predicted based on the product of L_{avg} and the number of sequences K_f sent to each device node. Recall K_f is approximately K total sequences divided by the number N of available device nodes. $T_{resp1\_msg\_size\_avg}$ is predicted based on the number of alignments processed from each device node or approximately ${K_f}^2$. Both $T_{req1\_msg\_size\_avg}$ and $T_{resp1\_msg\_size\_avg}$ can be computed without any knowledge other than the total number of sequences in the test input. The accuracy of the attribute values for message sizes $T_{req1\_msg\_size\_avg}$ and $T_{resp1\_msg\_size\_avg}$ is based on the analysis of simulation. Predicting attribute values for $T_{req2\_msg\_size\_avg}$ and $T_{resp2\_msg\_size\_avg}$ is more challenging since they cannot be calculated directly from the input test sequence properties. In this case, a worst-case approach is taken, where attribute values are configured for all N by utilizing measured $T_{req2\_msg\_size\_avg}$ and $T_{resp2\_msg\_size\_avg}$ values from the 128 device node lab results. Utilizing message size results for 128 device nodes results in an upper bound on message sizes values for N as it approaches 4K device

nodes. Results for T_{req2_N} and T_{resp2_N} are worst-case under these assumptions but consistent across all N devices.

For similar reasons, the accuracy of T_{resp1_N} and T_{resp2_N} as well as T_{req1_N} and T_{req2_N} is impacted as a result of two additional effects. First, server and device node computational time attribute error is introduced for N >128 nodes. This is a result of utilizing attributes values for $T_{S\_compute\{1..5\}}$ and $T_{C\_compute\{1..4\}}$ directly from N = 128 lab measurements. Secondly, variances in communication times exist simply from typical network packet loss and TCP/IP retransmission effects. For these reasons, simulation results for average individual timing attributes $T_{req1\_n\_avg}$, $T_{req2\_n\_avg}$,$T_{resp1\_n\_avg}$, and $T_{resp2\_n\_avg}$ are reported. Individual timing results measure the average communication time between server and device nodes, and provide a lower bound on the communication costs. With both an upper and lower bound in communication performance, simulation provides very useful results in evaluating large-scale broadband DOCSIS performance.

Table 7.24 lists the computed values for various predicted model attributes for N = 128 and 4K device nodes. These values suggest significant scaling potential as the number of device nodes increases from 128 to 4K, even under the worst-case assumptions previously described. For example, as the number of device nodes increases from 128 to 4K, the number of alignments per device node decreases for each sequence test input case. Values for $T_{C\_compute1\_N}$ also decrease. Consequently, the amount of computation required per device node decreases as N increases. Similarly, the average amount of message data $T_{req2\_msg\_size\_avg}$ and $T_{resp2\_msg\_size\_avg}$ that must be transferred decreases as N increases from 128 to 4K during the pairwise alignment communication phases. These values are consistent with those measured from Section 7.4.4.3.

7.4.4.6 Parallel MSA Launch-Execution and Completion Time Simulation Results

Results for $Texec_N$ and $Tcompletion_N$ from parallel MSA application model simulation execution across 128 and 4K device nodes are shown in Tables 7.25 and 7.26 respectively. Table 7.25 presents $Texec_N$ and $Tcompletion_N$ time measurements for multicast-unicast and unicast-unicast scenarios under the no-traffic scenario. Table 7.26 summarizes the timing results for the same set of experiments executed in the presence of traffic.

$Texec_N$ results show uniform multicast performance averaging 24ms for multicast delivery as we increase the number of device nodes from 128 to 4K in both traffic and no-traffic scenarios shown

#Nodes	Input Sequence	Multicast-Unicast Texec$_N$ (ms)	Multicast-Unicast Tcompletion$_N$ (ms)	Unicast-Unicast Texec$_N$ (ms)	Unicast-Unicast Tcompletion$_N$ (ms)
128	S100-L1500	24	384007	396	375041
	S200-L300	24	109846	397	109371
	S300-L200	24	45814	398	50170
	S500-L200	24	267629	396	202748
	S200-L500	24	150201	398	150031
	S500-L1100	24	2268941	397	2271284
	S1500-L100	24	1246176	395	1252049
4K	S100-L1500	23	115847	4037	123535
	S200-L300	23	32419	4024	36356
	S300-L200	23	21170	4036	26496
	S500-L200	23	55094	4029	58149
	S200-L500	23	39367	4037	45033
	S500-L1100	23	334009	4026	350339
	S1500-L100	23	780543	4032	784438

Table 7.25: Parallel MSA simulation results for Texec$_N$ and Tcompletion$_N$ no-traffic scenario.

#Nodes	Input Sequence	Multicast-Unicast Texec$_N$ (ms)	Multicast-Unicast Tcompletion$_N$ (ms)	Unicast-Unicast Texec$_N$ (ms)	Unicast-Unicast Tcompletion$_N$ (ms)
128	S100-L1500	24	372532	423	385382
	S200-L300	24	109672	430	111996
	S300-L200	24	46011	431	50259
	S500-L200	24	270857	425	203598
	S200-L500	24	151944	425	152035
	S500-L1100	24	2286755	417	2227447
	S1500-L100	24	1249449	423	1259953
4K	S100-L1500	24	115974	4093	127552
	S200-L300	23	32390	4101	36427
	S300-L200	23	21165	4104	26698
	S500-L200	23	55119	4093	58290
	S200-L500	23	37968	4102	45033
	S500-L1100	24	334275	4081	350410
	S1500-L100	23	792039	4089	784535

Table 7.26: Parallel MSA simulation results for Texec$_N$ and Tcompletion$_N$ traffic scenario.

in Tables 7.25 and 7.26. However, unicast Texec_N delivery time increases 10 times between 128 and 4K devices. As shown in Table 7.25, unicast Texec_N averages 396ms at 128 device nodes and averages 4032ms for 4K device nodes. Results for traffic scenarios shown in Table 7.26 are similar. Simulation results confirm the earlier process launch-execution experiments from Section 7.4.2.1, where multicasting outperforms unicast based message delivery for any given message size. Simulation results confirm that multicasting provides higher performance (less time) for parallel MSA application delivery compared to the unicast based approach. Texec_N variation across N (128 through 4K) for either multicast-unicast or unicast-unicast is consistent with lab results and small at less than 1%. Results from Tables 7.25 and 7.26 show that there is no significant impact due to background traffic profiles within simulation. This is consistent with lab results since both lab and simulation traffic configurations are similar as defined in Section 7.3. Tlatency_{avg} values are configured as lab using 41ms for multicast-unicast tests and 57ms for unicast-unicast tests.

Tcompletion_N time results are consistent across all experiments with Tcompletion_N decreasing (performance improving nearly linearly) as the number of device nodes increases. Tcompletion_N results in Tables 7.25 and 7.26 show an improvement for multicast-unicast cases over unicast-unicast results as the number of device nodes increases to 4K. This is primarily due to improved process launch-execution efficiency impact from lower Texec_N in the multicast-unicast scenarios and the negative impact of higher Tack_N values in unicast-unicast scenarios.

All simulation results for Texec_N, Tack_N and Tcompletion_N are illustrated graphically in Figures A.21 through A.24. Texec_N is uniform for all N in the multicast-unicast scenarios, and increases linearly for the unicast-unicast scenarios. Tack_N values are higher compared to Texec_N as expected and increases linearly as the number of device nodes increases. In the traffic scenario test case, results indicate a further increase in Tack_N as the number of device nodes increases. This is consistent with the previous results from Section 7.4.2.2 where Tack_N as a unicast operation begins to increase linearly as the number of device nodes increases, with slightly higher Tack_N values under lab traffic scenarios. Tcompletion_N performance for all 7 input sequence tests are shown. As the number of device nodes increase, Tcompletion_N performance improves linearly. However, the improvement is greatest up to 512 device nodes, while the rate of improvement decreases as the number of nodes is increased up to 4K. This effect indicates either: 1) a limit on the available data parallelism beyond 512 device nodes for the given set of input sequence tests, or 2) that the commu-

Rank	Input Sequence	Multicast-Unicast Speedup	Unicast-Unicast Speedup
1	S500-L1100	6.5/6.7	6.4/6.3
2	S500-L200	4.9/4.8	3.5/3.4
3	S200-L500	3.9/3.8	3.4/3.2
4	S200-L300	3.4/3.5	2.9/2.9
5	S100-L1500	3.1/3.2	2.9/2.9
6	S300-L200	2.4/2.3	1.8/1.5
7	S1500-L100	1.6/1.6	1.6/1.6

Table 7.27: Simulation speedup results for multicast-unicast and unicast-unicast with no-traffic/traffic scenarios.

nication overhead (especially upstream unicast communication flows) is limiting the Tcompletion_N performance improvement rate as the number of device nodes increases, relative to computation.

To summarize key results of this section: 1) Results are consistent with simulation results from Section 7.4.2.2 confirming the performance advantage of multicast over unicast for large-scale network application delivery. 2) The impact of upstream unicast communication, especially as the number of upstream connections increases in direct relation to the number of device nodes confirms the cost associated with unicast communication flows. The performance impact is primarily due to upstream DOCSIS contention effects as discussed in Section 7.4.2.2 resulting in a linear time decrease in message delivery performance.

7.4.4.7 Parallel MSA Execution Time Speedup Simulation Results

Parallel MSA simulation speedup results for both multicast-unicast and unicast-unicast scenarios, with traffic and without are listed in Table 7.27. Speedup improvement is defined as the ratio of MSA execution times calculated by dividing Tcompletion_{4K} (4K device nodes) by Tcompletion_{128} (128 device nodes). Average speedup varies between 1.6 and 6.7 depending on the sequence test case. Results are comparable across traffic and non-traffic cases. Results suggest that DOCSIS unicast communications overhead has significant impact on speedup. This can be illustrated by analyzing the S1500-L100 test case whose speedup ranks lowest at 1.6 across all test scenarios. S1500 contains the largest set of input sequences (1500) and, therefore, requires the longest time duration (number of alignments is proportional to square of number of sequences in input) for device node T_{resp2_N} responses. S1500-L100 also requires the least amount of device node computation

#Nodes	Input Sequence	T_{req1_N} (ms)	T_{resp1_N} (ms)	T_{req2_N} (ms)	T_{resp2_N} (ms)
128	S100-L1500	6179	285448	164	103595
	S200-L300	6132	87205	1342	19663
	S300-L200	6123	28381	3859	8896
	S500-L200	6158	225830	26541	14434
	S200-L500	6169	121639	1259	26108
	S500-L1100	6338	2067930	29502	222725
	S1500-L100	6203	501890	712658	41844
4K	S100-L1500	6311	43169	150	103476
	S200-L300	6118	4647	1337	19700
	S300-L200	6109	1408	3860	8955
	S500-L200	6127	7233	26536	14450
	S200-L500	6228	10534	1256	26144
	S500-L1100	6492	75108	29480	222618
	S1500-L100	6260	19035	712637	41664

Table 7.28: Parallel MSA simulation overall request-response communication cycle time results.

with an average sequence length of 100. The remaining 5 input tests in Table 7.27 show speedup improvements that average between 2.4 and 4.9. These input tests differ in speedup by one 1, and contain sequences that exhibit reciprocal properties in terms of both number and average length of sequences relative to one another. For example, S300-L200 and S200-L300 have speedups of 2.4 and 3.4, respectively. S200-L500 and S500-L200 have speedups of 3.9 and 4.9, respectively. These results suggest that there exists an optimal balance between the length and number of sequences such that speedup results are maximized. The largest speedup improvement occurs in test input S500-L1100 with an average speedup of 6.7 across all scenarios. This suggests that as the number of device nodes is increased, speedup improvement gain is optimized for input tests that contain both a large number of sequences and, simultaneously, a longer average length.

Finally, multicast-unicast scenarios show a small speedup improvement compared to unicast-unicast scenarios. This may be explained due to the higher efficiency of multicast-unicast during launch-execution compared to unicast-unicast as we increase the number of device nodes. This result is consistent with similar results from Section 7.4.2.2, where it was shown that Tcompletion_N time is positively impacted in the multicast delivery scenario.

#Nodes	Input Sequence	$T_{req1\_n\_avg}$ (ms)	$T_{resp1\_n\_avg}$ (ms)	$T_{req2\_n\_avg}$ (ms)	$T_{resp2\_n\_avg}$ (ms)
128	S100-L1500	86	36	16	169
	S200-L300	38	36	8	101
	S300-L200	29	37	1	76
	S500-L200	60	57	6	101
	S200-L500	62	36	7	102
	S500-L1100	212	56	24	234
	S1500-L100	108	144	21	191
4K	S100-L1500	217	167	2	167
	S200-L300	23	165	3	163
	S300-L200	14	163	2	173
	S500-L200	29	166	1	167
	S200-L500	122	168	3	167
	S500-L1100	366	170	2	173
	S1500-L100	164	167	2	166

Table 7.29: Parallel MSA simulation individual request-response communication cycle time results.

7.4.4.8 Parallel MSA Communications Simulation Results

Parallel MSA simulation request and response communication results are shown in Tables 7.28 and 7.29 for multicast-unicast, no-traffic experimental scenarios. The tables report overall communication nested cycles and individual or single transmission times between server and device nodes, respectively. Experiments are executed for both traffic and no-traffic scenarios with no significant difference (less than 3%) noted in results. Results for unicast-unicast experiments are also comparable to the results in Tables 7.28 and 7.29: They are not reported because they differ by less than 1%. The lack of difference between multicast-unicast and unicast-unicast scenarios is due to the common unicast model communication flows during the parallel execution of the MSA algorithm.

Request communication cycles. Referring to Table 7.28, T_{req1_N} and T_{req2_N} overall request times for downstream communications (from server to all N device nodes) show a small change as the number of device nodes increase from 128 to 4K. This is due to two reasons:

1. $T_{req1\_msg\_size\_avg}$ and $T_{req2\_msg\_size\_avg}$ message sizes decrease as the number of device nodes increase, which tends to offset one another. For a given number of K sequences to distribute among N device nodes, the corresponding $T_{req1\_msg\_size\_avg}$ and $T_{req2\_msg\_size\_avg}$ message sizes decrease as N increases. For example using results of S100-L1500, as we increase the number

of devices nodes from 128 to 4K, $T_{req1\_msg\_size\_avg}$ message sizes decrease from 18834 bytes to 6291 bytes or a factor of 3X. For S1500-L100, $T_{req1\_msg\_size\_avg}$ message sizes decrease from 25036 bytes to 4394 bytes or a factor of 5.7X. Under the same test cases, message size reduction factor is even higher for $T_{req2\_msg\_size\_avg}$ with results for S100-L1500 computed as 47X and S1500-L100 equal to 52X respectively.

2. As described in Section 7.4.4.5, simulation accuracy is impacted when utilizing lab experiment values within simulation. The impact occurs for all timing simulations where N is greater than 128 device nodes.

To better evaluate actual device node to server transmission time communication performance, Table 7.29 shows results for individual average timing values $T_{req1\_n\_avg}$ and $T_{req2\_n\_avg}$. Individual request-response timing results show the average time required to transmit a single message between server and device nodes. Referring to the table values, note that message delivery time trends upward as we increase the number of device nodes from 128 to 4K, especially for larger sequence test cases S100-L1500, S200-L500, S500-L1100, and S1500-L100. As an example from Table 7.29, S100-L1500 increases from 86ms to 217ms as we vary between 128 and 4K device nodes. This is consistent with results from Section 7.4.2.2 where unicast message transmission time increases in direct relation to any increase in message size for a given number of device nodes. Results for $T_{req1\_n\_avg}$ and $T_{req2\_n\_avg}$ are averaged across all server to device node transmissions. Standard deviation is 4 ms.

Response communication cycles. Results for overall upstream communications (device nodes to server) are characterized by simulation timing values for T_{resp1_N}, and T_{resp2_N}, corresponding to response message delivery time across all N device nodes to the server. Results for $T_{resp1\_n\_avg}$ and $T_{resp2\_n\_avg}$ are reported in Table 7.29 to evaluate the communication performance of individual device node to server node transmission time. These results indicate that T_{resp1_N} performance improves as we increase the number of device nodes, especially when the number of input test sequences is large. For instance, in test input S1500-L100, T_{resp1_N} decreases from 501890ms to 19035ms as the number of device nodes vary from 128 to 4K. This is a factor of 26X improvement in performance. T_{resp1_N} improvement is due to two main reasons:

1. As we increase the number of device nodes from 128 to 4K, computation time at each node

is shorter resulting in a reduction in time required, or turn-around time per device node to complete all N device node transmissions. Lower turn-around time is directly related to a reduction in computation time as N increases. The reduction in device node computation is determined by the number of alignments per device node or $K_f{}^2$. The number of alignments processed per device node decreases as the number of device nodes increase.

2. As the message size $T_{resp1\_msg\_size\_avg}$ decreases, (S100-L1500 response message size is 224 bytes for 128 device nodes versus 24 bytes for 4K device nodes) there is simply less message data per device node to transmit to the server. Results show that T_{resp1_N} is impacted by increases in average sequence length as well as number of sequences. This is illustrated by the transmission time required for large sequence input test cases S100-L1500, S500-L1100, and S1500-L100. Individual timing results $T_{resp1\_n\_avg}$ show a significant increase in transmission time as N varies between 128 and 4K. Using S100-L1500 as an example, $T_{resp1\_n\_avg}$ increases by a factor of 4.7X (167ms versus 36ms) as the number of device nodes increase from 128 to 4K. Standard deviation is 16 for 128 device nodes and 113 for 4K device nodes. The large variation in standard deviation is due to the impact of DOCSIS upstream contention effects.

Results for T_{resp2_N} indicate that overall response cycle time remains uniform across 128 and 4K device nodes, increasing or decreasing as the sequence alignment problem varies in both number and length of sequence inputs. This result is similar to $T_{req1\_msg\_size\_avg}$, where the message size for $T_{resp2\_msg\_size\_avg}$ also decreases as we increase N. However, for $T_{resp2\_msg\_size\_avg}$, the reduction factor is approximately 47X across all sequence test inputs. The concurrent decrease in message size and increase in number of nodes offset one another resulting in a uniform set of T_{resp2_N} values for all N. Individual results shown in Table 7.29 support this. Simulation values show uniform variation in $T_{resp2\_n\_avg}$ across N device nodes. $T_{resp2\_n\_avg}$ standard deviation is computed as 48ms for 128 device nodes and 114ms for 4K device nodes. Both $T_{resp1\_n\_avg}$ and $T_{resp2\_n\_avg}$ values vary considerably, which is consistent with a large standard deviation. These results further confirm the impact of DOCSIS upstream contention effects and experiments in Section 7.4.2.2. The large variation in results does not impact interpretation of key results, however. Results in Tables 7.28 and 7.29 are consistent within a constant factor for all Parallel MSA simulation measurements.

#Nodes	S100-L1500	S200-L300	S300-L200	S500-L200	S200-L500	S500-L1100	S1500-L100
128	1.85/1.96	1.51/1.55	1.12/1.05	1.24/1.26	1.48/1.47	2.07/2.06	.32/.32
512	1.36/1.36	1.02/1.02	.62/.62	.58/.58	1.05/1.06	1.21/1.23	.12/.12
1024	1.29/1.29	.87/.88	.56/.48	.40/40	.96/.96	1.05/1.07	.08/.08
2048	1.41/1.38	.85/.86	.46/.46	.32/.32	.94/.94	1.14/1.11	.05/.05
4096	1.37/1.36	.84/.83	.41/.41	.28/.28	.93/.93	1.07/1.08	.04/.04

Table 7.30: Computation/Communication ratio for simulation multicast-unicast results for no-traffic/traffic scenarios.

#Nodes	S100-L1500	S200-L300	S300-L200	S500-L200	S200-L500	S500-L1100	S1500-L100
128	1.75/1.76	1.59/1.55	1.15/.92	1.14/1.12	1.56/1.41	2.01/2.04	.33/.32
512	1.35/1.36	1.02/1.03	.73/.73	.58/.58	1.04/1.04	1.21/1.21	.12/.12
1024	1.31/1.28	.89/.89	.62/.62	.39/.39	.95/.95	1.05/1.05	.08/.08
2048	1.36/1.36	.86/.86	.61/.60	.31/.31	.94/.93	1.08/1.09	.05/.05
4096	1.36/1.36	.84/.84	.57/.57	.27/.27	.93/.93	1.08/1.08	.04/.04

Table 7.31: Computation/Communication ratio for simulation unicast-unicast results for no-traffic/traffic scenarios.

This is because all attribute value configurations including predicted values are uniformly applied across all experiments.

To summarize this section, parallel MSA communication simulation results indicate that request and response message times are highly dependent on the number of nodes and message size variations. This is consistent with experiments from Section 7.4.2.2. Simulation results show that transmission times increase as the number of nodes increases, especially for larger messages. Upstream communications are further impacted by variability introduced by DOCSIS scheduling and contention effects. These effects become increasingly impactful as the number of nodes are increased.

7.4.4.9 Comparing Parallel MSA Computation Versus Communication Ratios

To gain additional insight into parallel MSA scaling potential, computation and communication simulation results are used to compute computation-to-communication ratios. Computation-to-communication ratios for multicast-unicast experiments are shown in Table 7.30. Similarly, ratio values for unicast-unicast experiments are shown in Table 7.31. Each table indicates both no-traffic and traffic scenarios (indicated as no-traffic/traffic in tables). Overall, resulting ratio values are

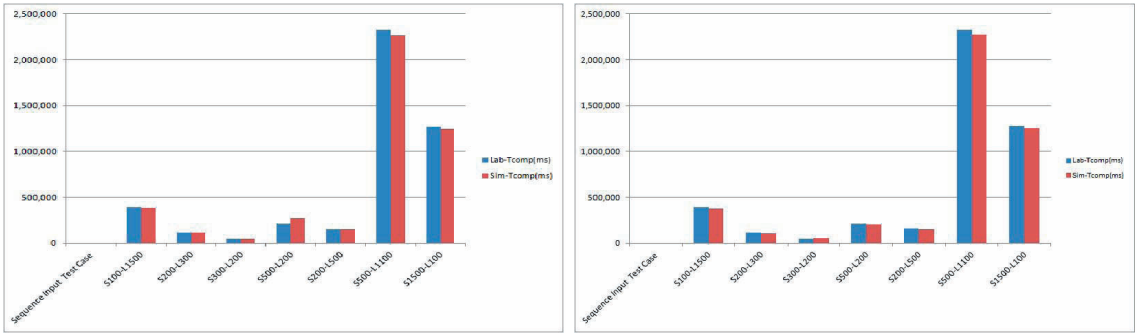
comparable for both multicast-unicast and unicast-unicast tables as well as traffic and no-traffic scenarios.

Results shown in both tables confirm previous results from Sections 7.4.4.7 and 7.4.4.8: the impact of communications increases as the number of device nodes increases. From Section 7.4.4.8, results for T_{resp1_N} and T_{resp2_N} showed that upstream communications had the largest impact on overall performance. Therefore, upstream unicast device node-to-server messaging contributes the most to the communications component of the computation-to-communication ratio. A key result is that all ratio values decrease as N increases for each parallel MSA experiment. For example, the ratio for S1500-L100 in Table 7.30 decreases from .32 to .04 as we increase from 128 to 4K device nodes. However, experiments where the average sequence length is considerable longer (S100-L1500 and S500-L1100), the ratio decrease is much smaller as the amount of computation overcomes the communication overhead as reflected in the table ratio values. For example, ratio values for S100-L1500 in Table 7.30 decrease from 1.85 to only a minimum of 1.29 at 1024 device nodes. Since S100-L1500 requires higher computation during the MSA device node alignment phase, and there are 100 input sequences, the resulting computation-to-communication ratio increases at 4K device nodes to 1.37.

The ratio results from Tables 7.30 and 7.31 further confirm the analysis in Section 7.4.4.7: optimizing parallel MSA speedup requires consideration of input problem size (number of sequences) and computational complexity (average length of sequences) in relation to the computational capabilities, and number of device nodes within the communications network.

7.4.4.10 Comparing Parallel MSA Simulation Versus Lab Results

A comparison of lab and simulation parallel MSA execution completion times for both multicast-unicast and unicast-unicast, no-traffic scenarios for N = 128 device nodes is shown Fig. 7.15. Lab and simulation results are very similar, within 10% accuracy across all sequence input tests. The high degree of accuracy is due to the majority of computation time during the device node pairwise alignment phase or $T_{C\_compute1\_N}$. Additionally, model attribute values across lab and simulation are essentially equivalent further improving upon simulation accuracy. Results in Fig. 7.15 illustrate the accuracy of simulation versus lab results confirming simulation methods for evaluating scalability and potential of large-scale broadband embedded computing systems.



(a) Multicast-unicast scenario

(b) Unicast-unicast scenario

Figure 7.15: Comparison of Lab versus simulation parallel MSA completion time results.

Fig. 7.15 also illustrates any differences between the multicast-unicast and unicast-unicast scenarios as shown on the left and right sides of the figure. Results show no significant completion time improvement between multicast-unicast and unicast-unicast parallel MSA launch-execution phases. This suggests that short running application processes benefit greater from multicast based launch methods compared to long running application processes. However, from Section 7.4.2 and 7.4.4.6, application process execution through multicasting is optimal over unicast methods overall. Therefore, for any size process launch-execution, there is potential improvement of launch throughput where many application processes must be initialized and launched concurrently. This experimental work is outside the scope of this dissertation and is part of future research.

In concluding this section, parallel MSA lab versus simulation completion time results are similar, demonstrating the accuracy of the parallel MSA simulation in comparison to real world parallel MSA execution. This confirms the validity of utilizing simulation to explore the capabilities of large-scale broadband networks for broadband embedded computing.

7.4.5 Scalable Network Architecture For Broadband Embedded Computing

Results from Sections 7.4.2.1 and 7.4.4 show the performance advantages of multicasting over unicast communication mechanisms, especially as the number of device nodes increase. This confirms the feasibility of implementing scalable runtime systems that efficiently manage, distributed process launch-execution across millions of device nodes by utilizing multicasting technologies. Broadband multicast-unicast network architectures are key in the implementation of heterogeneous Cloud sys-

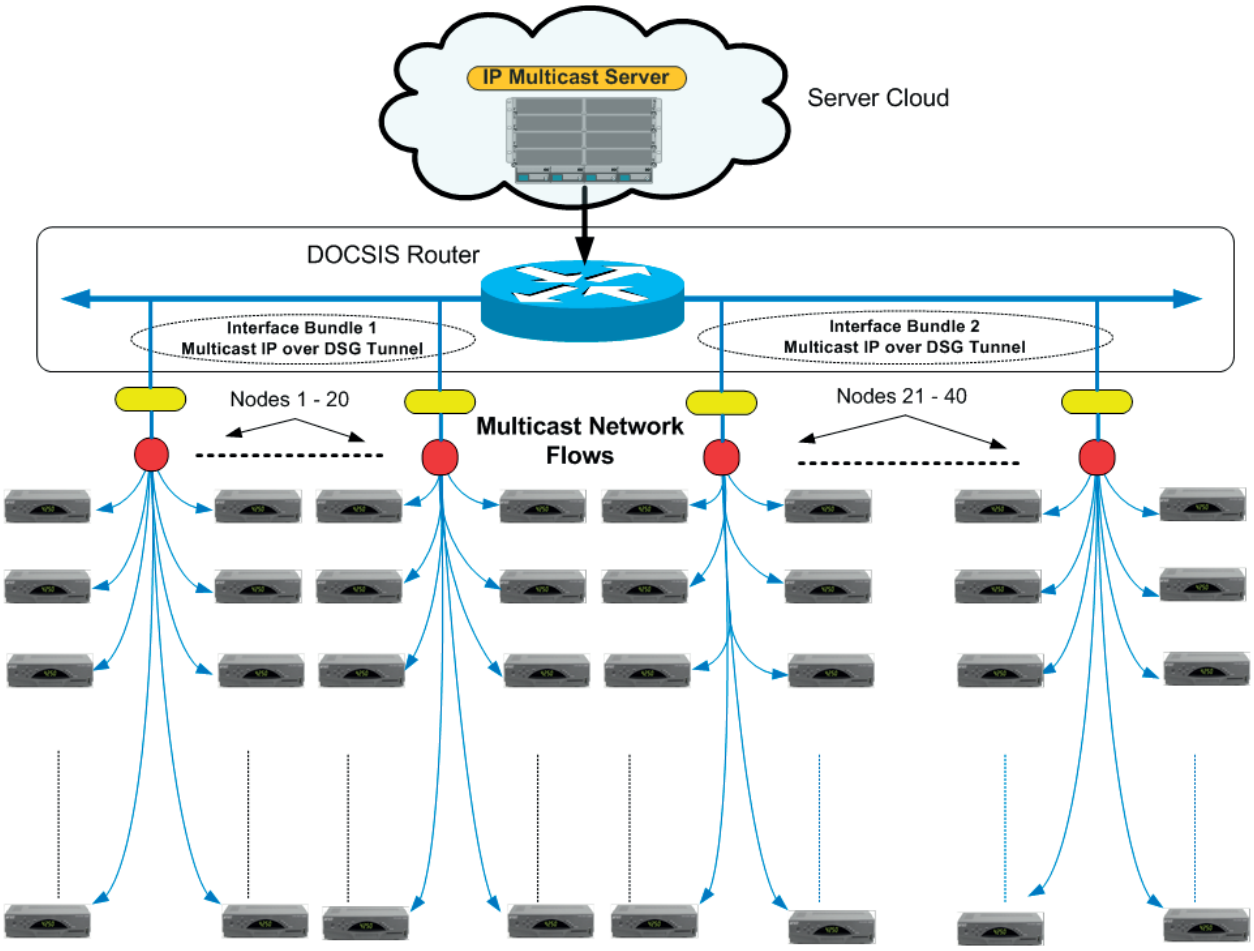


Figure 7.16: Scalable broadband network architecture based on DOCSIS multicast.

tems based on broadband embedded computing that leverage distributed computing technologies such as MPI and Map-Reduce.

Fig. 7.16 illustrates the architecture for a multicast-unicast network infrastructure instance typical in a large service provider Cloud data-center and remote-hub facility. Recall from Section 2.4.2 that remote-hub facilities are geographically dispersed, and contain numerous DOCSIS routers required to delivery network services to users. The figure represents one instance of a Cloud computing facility containing a single multicast server system. For example, the multicast server may provide runtime launch-execution services described in Section 5.4.2 to deliver, through multicasting, applications to tens of thousands of devices. Each multicast server is associated to one or more

DOCSIS routers shown in Fig. 7.16. The DOCSIS router converts multicast messages received over gigabit Ethernet or fiber interfaces to DOCSIS based DSG multicast messages over broadband RF media. Distribution over the broadband network (described earlier in Section 7.2.1) across multiple multicast DSG tunnel interfaces (indicated as Interface Bundle 1 and Interface Bundle) is shown in Fig. 7.16. The DOCSIS router associates one or more multicast DSG tunnels to physical interfaces within each router. Each router interface feeds from 1 to 20 RF nodes as described in Section 2.4.2. Through this partitioning, multicast scalability is achieved as represented in the lower portion of Fig. 7.16 as numerous multicast network flows from router and received at all device nodes. In Chapter 8, the described network architecture is utilized to propose a complete heterogeneous Cloud system based on large-scale broadband embedded computing.

7.5 Related Works

There is extensive work in the analysis of broadband communication networks that focus on distribution of video and audio, as well as performance characteristics as they relate to voice-over-IP service delivery. Harte *et al.*, describes multicast methods for optimal transmission of video and audio content to many broadband receivers [44]. Shah *et al.*, contributes a study to analyze DOCSIS physical layer performance, identifying performance improvements through protocol enhancements [89; 109]. Harte *et al.*, analyzed the impact to TCP/IP application performance due to various factors such as congestion and number of nodes [90; 20]. Similar to Harte *et al.*, DOCSIS performance is evaluated through lab and simulation, however this chapter presents simulation results up to 8K nodes and experiments that enable the analysis of parallel application execution across a broadband network. Futher, this work places greater emphasis on measuring the communications performance specific to application distribution over broadband DOCSIS networks for broadband embedded computing. Finally, while prior related work for DOCSIS performance evaluation is based on DOCSIS 1.1 standard experimentation [89; 109]. This work is the first extensive experimental study involving application distribution across a large number of DOCSIS devices nodes in conjunction with DOCSIS 2.0 simulation and system environments.

7.6 Summary

This chapter examined two key communication methods, multicasting and unicast, for broadband network computing. Experimental models based on process launch-execution, and a real world parallel application (Multiple Sequence Alignment) are tested through simulation and lab execution. The comparative analysis of results from lab experiments and simulation confirm that multicasting is the optimal communication method for large-scale data delivery. This result is confirmed across small and large message sizes and as the number of broadband devices is increased up to 8K devices. Results also suggest that multicasting improves the overall completion time for short-running application processes because it minimizes the launch initialization overhead, thus allowing the overall network of devices to begin computation almost immediately. However, it is also shown that in cases where messages are unique for each device, unicast communication methods excel over multicast for all sizes, except for very small messages. This leads to the conclusion that multicasting is best utilized for implementing runtime environments that manage application process distribution, and unicast for cases where unique messages must be transmitted to device nodes. Numerous applications require unicast communications flows. Experimental execution and simulation of parallel MSA confirmed the performance impact due to unicast message size, number of device nodes, and DOCSIS contention effects. Therefore, practical heterogeneous Cloud systems built utilizing broadband networks require a mixture of both multicast and unicast communication methods, where the particular method utilized is selected to minimize total communication costs. Future work shall further optimize communication costs associated to broadband embedded computing applications by utilizing policy based management of network traffic. The DOCSIS standard discussed in Section 2.4.1.3 provides mechanisms for establishing traffic priorities among multiple unicast communication flows. Finally, an architecture for multicast-unicast based communications over DOCSIS was described for implementing a broadband embedded computing system. This will be leveraged in Chapter 8 as the broadband network architecture for implementing a highly-scalable heterogeneous Cloud system utilizing broadband embedded computing.

Chapter 8

Heterogeneous Cloud Systems Based On Broadband Embedded Computing

8.1 Introduction

Chapters 5 and 6 presented experimental systems that support the message passing computational model using Open MPI, and the MapReduce computational model using Hadoop. Both experimental system platforms demonstrated a unique heterogeneous systems implementation that combines traditional data-center class compute cluster servers with distributed embedded devices. Chapter 7 provided an in-depth analysis of the broadband communication characteristics of a DOCSIS network of embedded devices. Results confirmed the need for communication mechanisms for optimal massive-scale distributed process launch-execution - a key challenge to overcome in the implementation of any large-scale heterogeneous computational system with millions of computational devices. Chapter 7 also presented results to optimize embedded devices communications with data-center servers such as those found in centralized Cloud facilities.

Service provider systems such as MSOs manage heterogeneous environments consisting of thousands of data-center class servers, peta-bytes of storage, high-performance fiber-optic, and interconnection networks that integrate large-scale broadband networks with millions of embedded devices. These embedded devices will continue to expand in variety, increase in number, performance and decrease in size as technology continues to advance.

Based on previous chapters, a proposed service provider architecture for heterogeneous Cloud

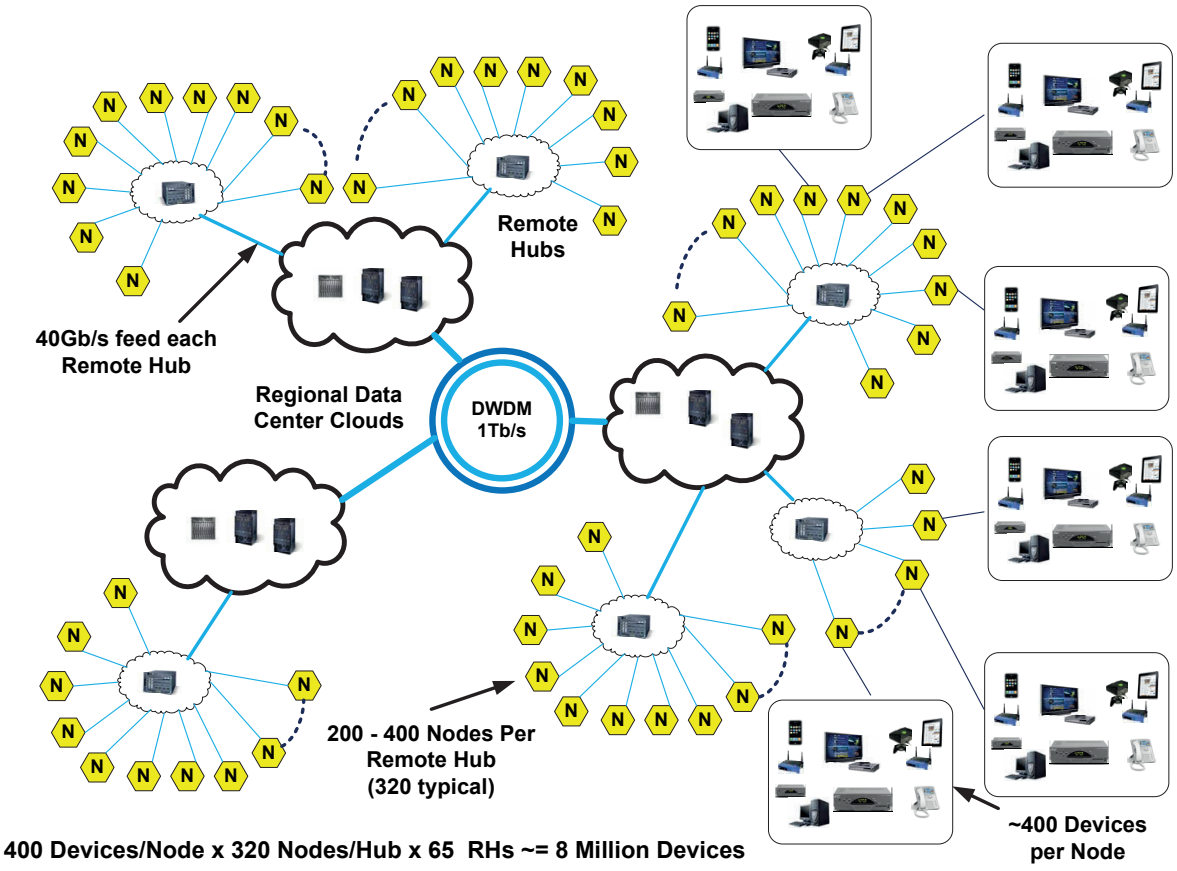


Figure 8.1: Large-scale MSO service provider Cloud.

systems utilizing broadband embedded computing is presented in this chapter. The architecture leverages key results from the experimental systems built and tested in Chapters 5 and 6, which described how to implement scalable heterogeneous computing by integrating broadband embedded devices with centrally managed compute clusters. The system network architecture is based on results from Chapter 7, where experiments confirmed optimal scenarios for scalable distributed runtime environments and data distribution across broadband networks. A description of four proposed heterogeneous Cloud application scenarios that leverage broadband embedded computation concludes the chapter.

8.2 Scalable Broadband System Architecture

Fig. 8.1 illustrates the architecture of a typical MSO service provider system capable of supporting 8 million embedded devices. A typical system is highly distributed with one or more regional data-center Cloud facilities providing managed Cloud services. This includes distribution of multimedia content, storage, home security, health monitoring, and IP network data to multiple broadband-connected device types as shown at the right of the figure. In Fig. 8.1 three regional data-centers are illustrated that provide distribution of multimedia content and data to 60-70 remote-hub facilities. Each remote-hub facility contains 5-20 DOCSIS routers that provide broadband connectivity for 100K to 250K broadband devices. Data-centers and remote-hub facilities are all interconnected with high-bandwidth fiber that utilizes dense wavelength division multiplexing (DWDM) switch technologies (up to 1Tb/s communications links). Remote hubs are connected using fiber-optic cable to RF translation devices called *nodes*. Nodes are passive devices that convert between a fiber-optic network and an RF broadband cable network that residential and business broadband devices attach to. In addition to residential connected devices, MSO service providers also offer large-scale WiFi connectivity through thousands of WiFi access points to each of the remote-hub facilities. In this manner, MSO providers are capable of offering a rich mix of content and network services from their own data-center Cloud and those of other service provider Clouds through peering agreements. Due to its hierarchical structure, the broadband system architecture is highly scalable and can support millions of devices. As an example, shown in Fig. 8.1, eight million broadband connected devices are supported as follows. Each node or RF cable section supports approximately 400 consumer embedded devices. A typical MSO service provider may have 320 nodes or more throughout their service region that may span hundreds or thousands of miles in diameter. Each remote-hub supports approximately 128K homes or access points. Assuming approximately 65 remote-hubs, the system scales as $400 \times 320 \times 65$ or approximately 8 million devices as illustrated in Fig. 8.1.

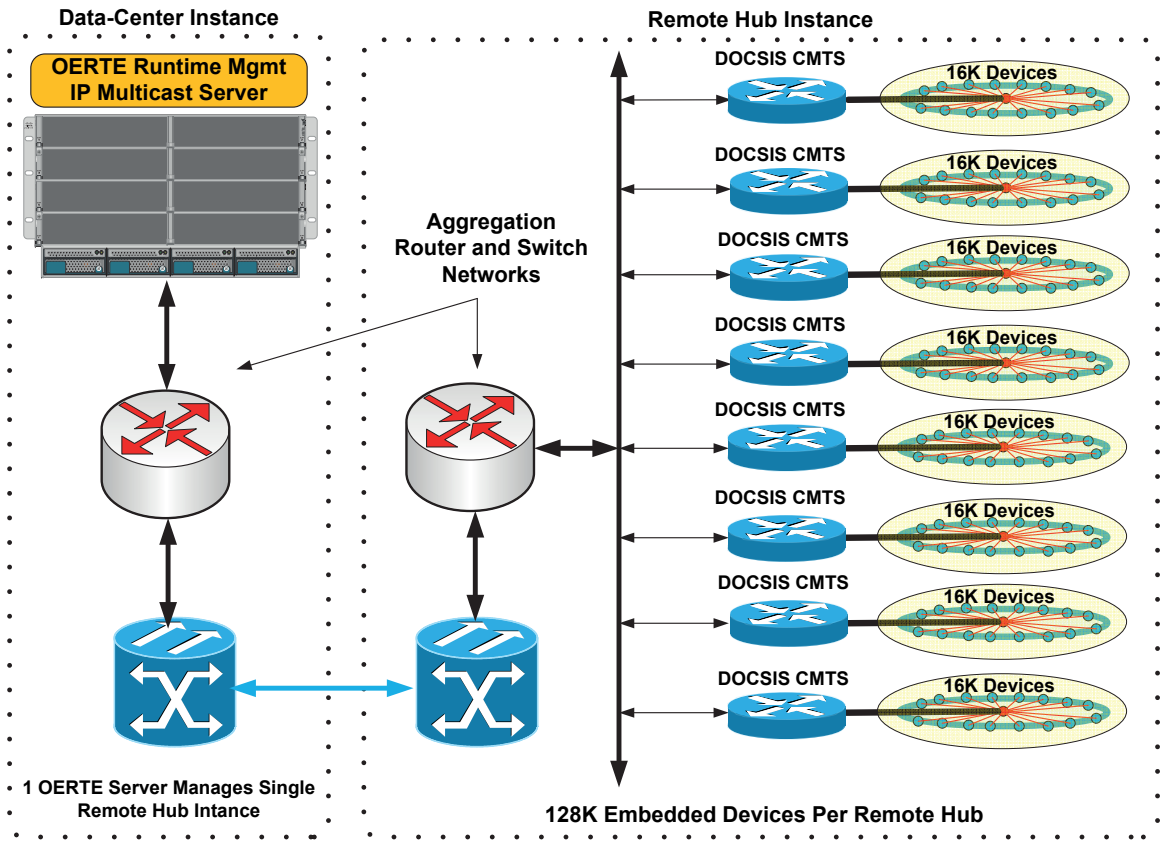


Figure 8.2: Remote hub unit as an instance of runtime environment scalability.

8.3 Scalable System Architecture for Cloud Systems Based on Broadband Embedded Computing

In order to implement a system architecture for broadband embedded computing that supports millions of devices, scalable methods for distributed process management and communications is required. In addition, standardized runtime libraries that support distributed programming models such as MPI and MapReduce must be implemented on both data-center and embedded devices. MPI and MapReduce represent one class of distributed programming models for execution across a heterogeneous Cloud system. However, other models of distributed computation are also possible as discussed in Section 2.2.

Chapter 5 described a scalable runtime system called *Open Embedded Runtime Environment* or OERTE that supports process life cycle management for distributed embedded systems. OERTE supports, but is not limited to, process launch and execution of MPI and MapReduce applications across a broadband system with millions of embedded devices. OERTE achieves high scalability by virtualizing the embedded device nodes into the runtime environment space of standard data-center runtime systems such as the ORTE system [18]. Through this virtualization model, Cloud computing clusters can be expanded to a much larger heterogeneous Cloud of both data-center servers and embedded devices. Using OERTE, a Cloud server complex can launch processes across the Cloud cluster and the millions of embedded devices concurrently within seconds. Section 5.4.2 describes in detail the implementation of the OERTE system consisting of server and client software components. To support heterogeneous computation in a distributed environment, runtime libraries must be available across both data-center and embedded devices software environments. Chapters 5 through 7 described MPI and MapReduce implementations that support distributed programming models for heterogeneous computation, and confirmed multicasting as the most efficient mechanism for application launch-execution across N device nodes. In the proposed scalable system architecture, multicasting is integrated into the OERTE software architecture to solve the process-launch execution problem for managing execution of applications across large-scale broadband embedded computing systems. Communications performance of the multicasting OERTE server is discussed in the parallel MSA lab experiments in Section 7.4.4.1. Fig. 8.2 illustrates an instance of a single remote-hub with connection back to the Cloud data-center and OERTE multicasting process launch-execution runtime management server. In Fig. 8.2, each remote-hub supports 128K broadband devices that are partitioned across 8 DOCSIS routers where each router supports 16K devices. The 8 routers are aggregated across a single high-performance LAN switch that interfaces directly with the DWDM fiber switch. In the proposed architecture, each Cloud data-center has multiple OERTE servers supporting the respective population of remote-hubs associated to a given data-center. Based on the MSO broadband system described in Section 2.4.2, an 8 million device system would result in approximately 65 OERTE servers spread across the 3 regional data-centers. As illustrated in Fig. 8.1, multiple remote-hubs correspond to a given Cloud data-center facility. Indeed, MSO system scalability is based on the multi-level partitioning that exists between Cloud data-centers and remote-hub facilities.

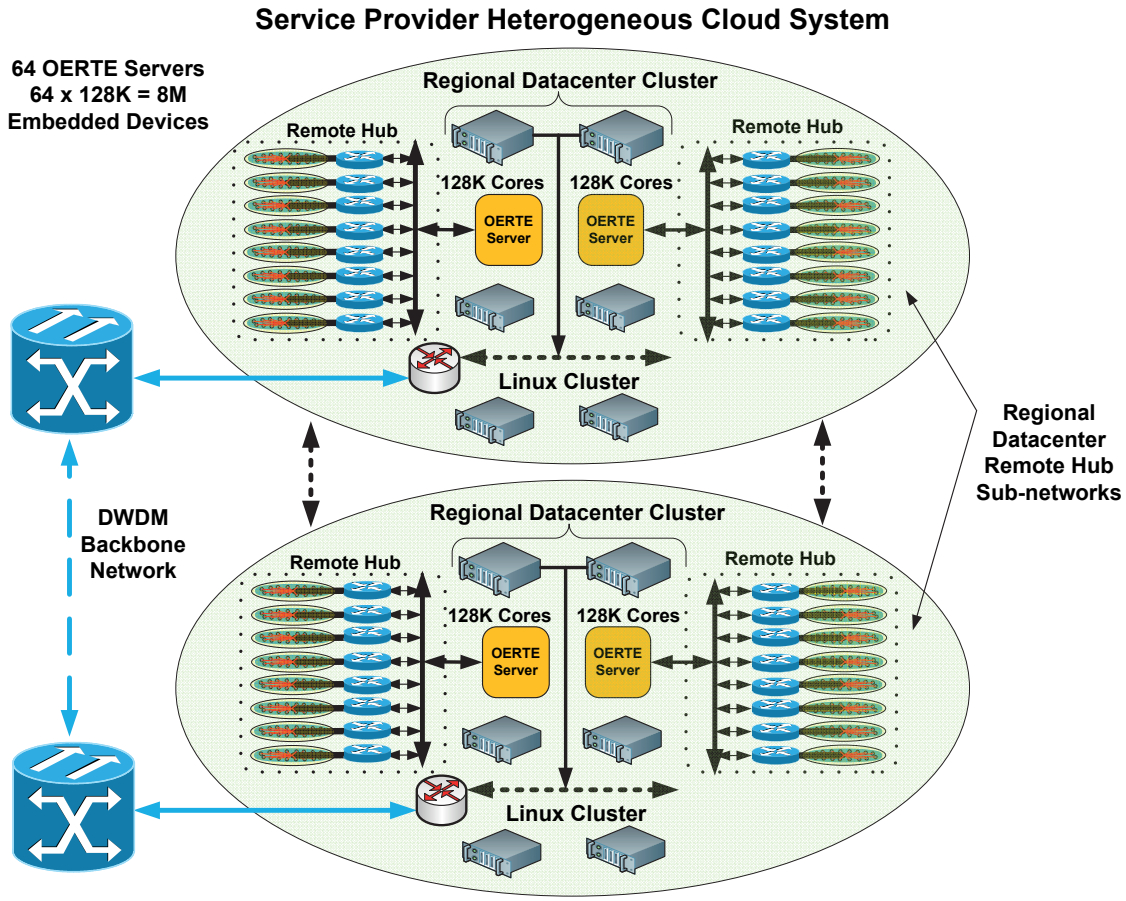


Figure 8.3: Heterogeneous Cloud system architecture utilizing broadband embedded computing.

Fig. 8.3 illustrates the system and network architecture view of the proposed heterogeneous Cloud system. In Fig. 8.3, each regional data-center is shown with multiple OERTE servers mapped to one or more remote-hub instance across the dedicated sub-network connection detailed in Fig. 8.2. The OERTE server is also connected to the Linux data-center cluster through a second high-performance network. Fig. 8.3 illustrates how these multiple regional data-center Clouds, are interconnected over a DWDM backbone.

To implement a heterogeneous Cloud system based on broadband embedded computing, the system and network architecture must supports a number of key properties, including:

1. Device heterogeneity.

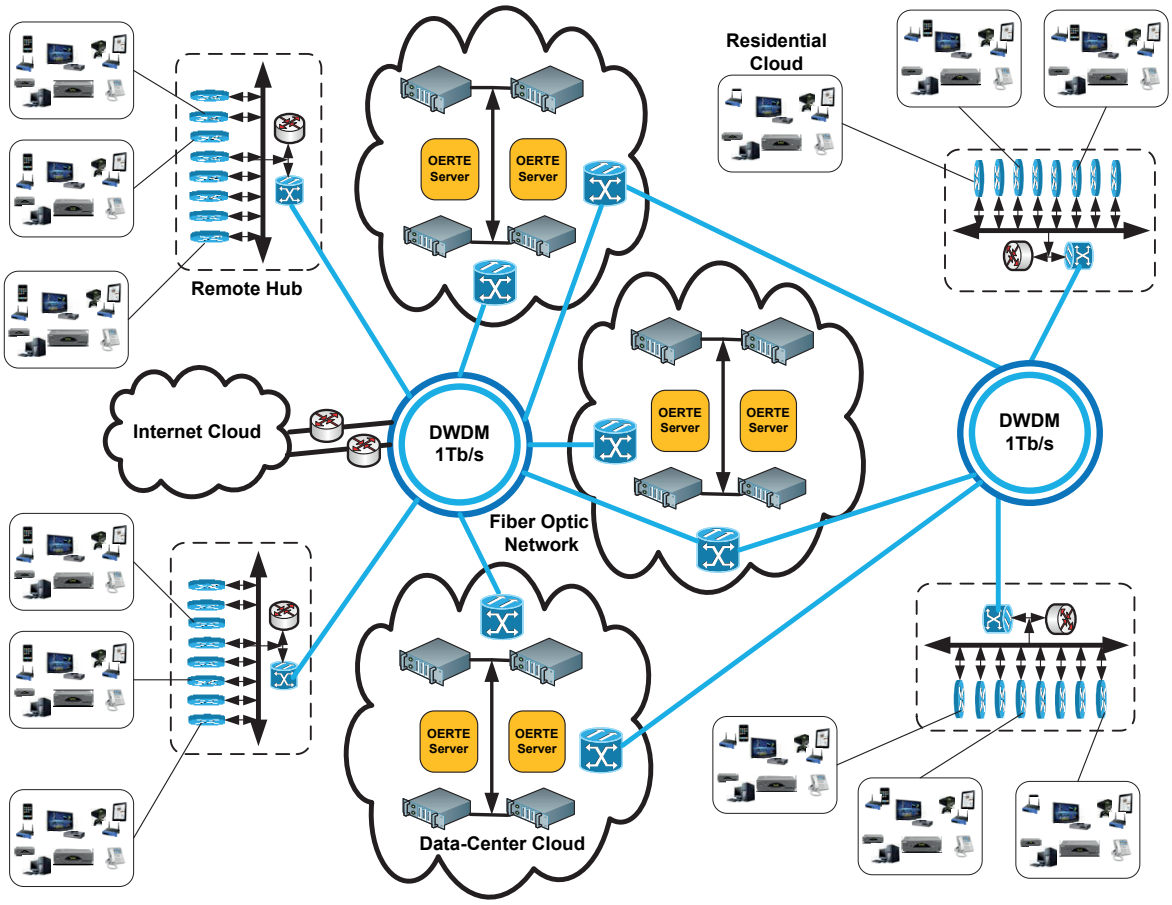


Figure 8.4: Service Provider heterogeneous Cloud system.

2. Scalability supporting millions of nodes.

3. Distributed and geodiverse.

4. Support low-latency communication network among all data-centers and broadband connected embedded devices.

5. Support for scalable distributed runtime environments and models of computation such as Message Passing and MapReduce.

The proposed broadband service provider system is illustrated in Fig. 8.4. The system architecture exhibits each of the key properties listed above. The system is heterogeneous since it contains

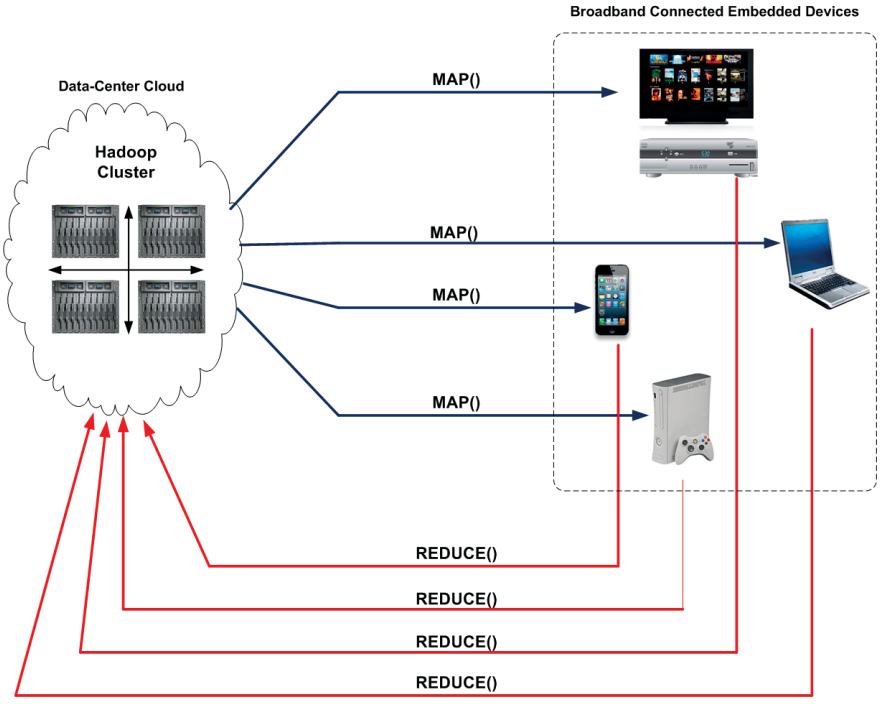
both data-center Linux servers and embedded devices with the capability to integrate wireless devices through distributed network connected access points. It is highly distributed supporting millions of embedded devices and access-point connections across a managed network backbone spanning long distances. The system network is based on 1Tb/s DWDM fiber technology offering both high-performance and low-latency communications. To accomplish geo-diversity and fault-tolerance, the communications infrastructure is fully connected with multiple DWDM fiber backbone connections to each regional data-center. Finally, software frameworks for distributed programming and runtime management of processes enable the execution of Cloud applications. In the proposed system, standard distributed models of computation such as MPI and MapReduce have been implemented to evaluate the feasibility of distributed Cloud computing applications.

8.4 Heterogeneous Cloud Application Scenarios Utilizing Broadband Embedded Computing

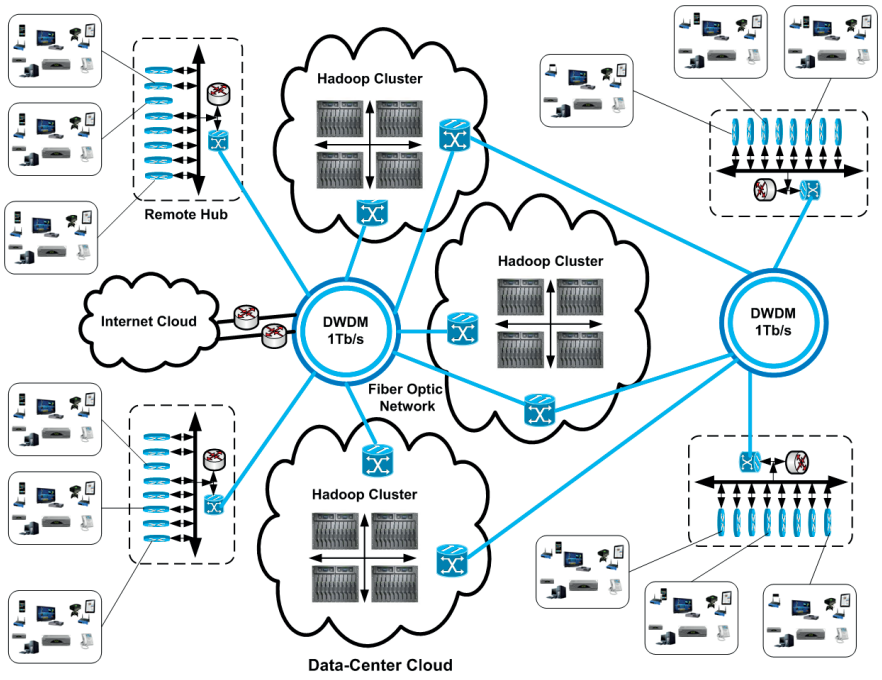
In this section I propose a number of novel heterogeneous Cloud system application scenarios that can leverage broadband embedded computation. Each scenario includes a proposed architecture that illustrates heterogeneous computation across centralized data-center servers and broadband embedded devices utilizing the methods described in previous chapters.

8.4.1 Big Data Mining and Processing

When applied to the processing of large-scale, peta-byte data sets, in data-mining and analytics applications Cloud computing is commonly referred to solving the problem of *Big Data*. Chapter 1 presented the explosion of Big Data applications primarily as a result of the growth in the number of users of mobile and other embedded devices at the network edge. Embedded devices at the network edge connect to centralized Cloud infrastructures across wireless and broadband networks. Broadband service provider systems support millions of embedded devices at the network edge in the form of set-top boxes, tables, home-gateways and other network attached intelligent devices. Utilizing methods for broadband embedded computing, centralized Cloud data-center processing for Big Data applications may be augmented using distributed computation at the network edge to increase Big Data computational performance and throughput. Chapter 6 described a broadband



(a)



(b)

Figure 8.5: Broadband System for Big Data: (a) Basic Block; (b) Cloud System Architecture

embedded computing system for MapReduce computation based around the Hadoop system that may be applied to the implementation of a heterogeneous Cloud systems platform for Big Data computing.

As an example of a typical data-mining application, the left side of Fig. 8.5 illustrates a basic heterogeneous Cloud system block composed of a single Hadoop cluster and various embedded devices executing Map and Reduce processes. The right side of Fig. 8.5 expands this basic block to a large-scale heterogeneous Cloud system architecture described in Section 8.2. In this scenario, multiple Cloud data-centers are executing Hadoop tasks across millions of broadband connected embedded devices. The system is efficient since data generated at the network edge may be directly processed by the embedded devices, prior to further computation in the centralized Cloud system. The Cloud system is fully heterogeneous as computation is shared across both the centralized Cloud data-center servers and the broadband embedded devices. Cloud services are delivered as either PaaS or IaaS and accessed by client applications from the centralized data-center clusters. The augmentation enabled by the broadband embedded computing system is fully transparent and hidden from the client application devices.

8.4.2 Cloud Recommendation System

A second scenario is shown in Fig. 8.6 illustrating an architecture for anonymously generating consumer recommendations. The system is a heterogeneous Cloud system based on broadband embedded computing and MPI. Delivery of consumer recommendations is provided in a *software as-a service (SaaS)* model to third-party content delivery systems that consume the service for the personalization of content as illustrated at the top of the figure. Consumer recommendations are computed individually and locally on each embedded device in an inherently parallel manner. A centralized server manages both the broadband embedded computation and data operations using distributed MPI collective operations. Distributed recommendation process management is provided using a customized Open MPI runtime framework such as the OERTE described in Section 5.4.2. Individual consumer preference profiles are processed independently on the broadband embedded devices as shown on the right-end side of Fig. 8.6. Embedded device software performs both sensory and analytical processing that is transmitted using collective MPI operations to the centralized Cloud server for final processing and service delivery. Note that since computation is

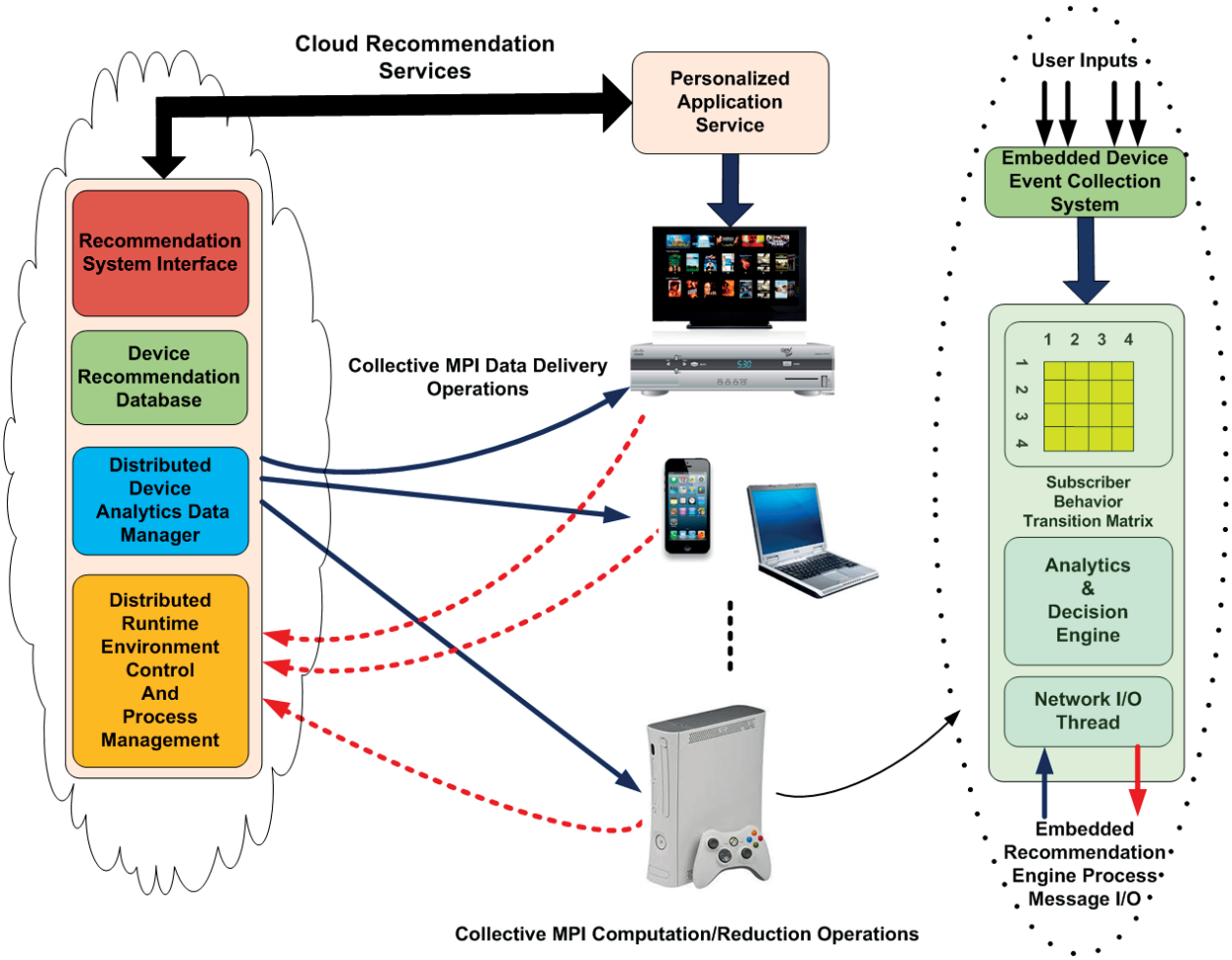


Figure 8.6: Recommendation System Architecture.

local to each consumer device, personal data can be transformed into non-personal data that may be then processed in the centralized Cloud infrastructure anonymously. Therefore content providers can personalize their services to consumers based on heterogeneous Cloud system computation without any knowledge of the consumers. This example highlights an advantage of heterogeneous Cloud systems architecture where distributed computation may be leveraged to support applications that require protection of sensitive information.

8.4.3 Network Intrusion Detection System

Standard security deployments such as firewalls, patched operating systems and password protection are limited in their effectiveness because of the evolving sophistication of intrusion methods. For instance, distributed network attacks are increasingly capable of breaking through infrastructure entry points [87]. Intrusion Detection Systems (IDS) are designed to combat these attacks. A

network-based IDS monitors data-traffic data from computers and other devices such as routers or gateways that are normally subject to attacks. However, millions of computer devices that utilize broadband networks for Internet access are potential sources of attacks, supporting growing interest in IDS systems that include broadband network systems.

A network based IDS works by matching data-traffic contents against a known attack profile, also known as a *signature*. A signature-based implementation approach utilizes various comparison methods that share implementation similarities with the MSA application discussed in Section 4.5. Both methods apply pattern matching to align one or more data-sets or signatures optimally. Coull *et al.* [23], modified the pair-wise sequence alignment phase to align sequences of system commands instead of sequences of DNA. The resulting score indicates how similar two command sequences are to one another as part of a strategy for detecting anomalous behavior. This same approach may be used to compare a known network signature with an anomalous one representing any number of network based attacks. For example, a TCP SYN flood attack represents a common denial-of-service (DOS) network attack where a target system is exhausted of resources leading to a disruption or reduction of service or even a crash. Typically one or more attacking hosts fake their source Internet addresses and generate numerous TCP SYN requests (the TCP SYN request is the first request of the standard TCP 3-way handshake required during a connection request between a client and server) to a target system. The attacking clients then ignore any responses from the target server, resulting in target server resource depletion and eventual loss of service. An IDS system based on broadband embedded computing is shown in Fig. 8.7. The figure illustrates a possible architecture for implementing a heterogeneous Cloud system for IDS PaaS service delivery to one or more service providers [97]. The system operates as follows. A centralized IDS master controller maintains a signature repository of *attack scenarios* which are distributed to multiple IDS embedded control systems located in each remote-hub facility. Each IDS embedded control system manages a portion of the broadband network and associated set-top embedded devices that are utilized for network monitoring. Multiple IDS embedded control systems partition the network into smaller, more manageable units. Fig. 8.7 illustrates three independent broadband networks, 10.4.x, 10.3.x and 10.2.x, that are monitored for sources of network attacks to a centralized server complex connected to all three networks. The set-top devices continuously monitor the broadband network and execute a signature MSA comparison algorithm. This algorithm matches an attack

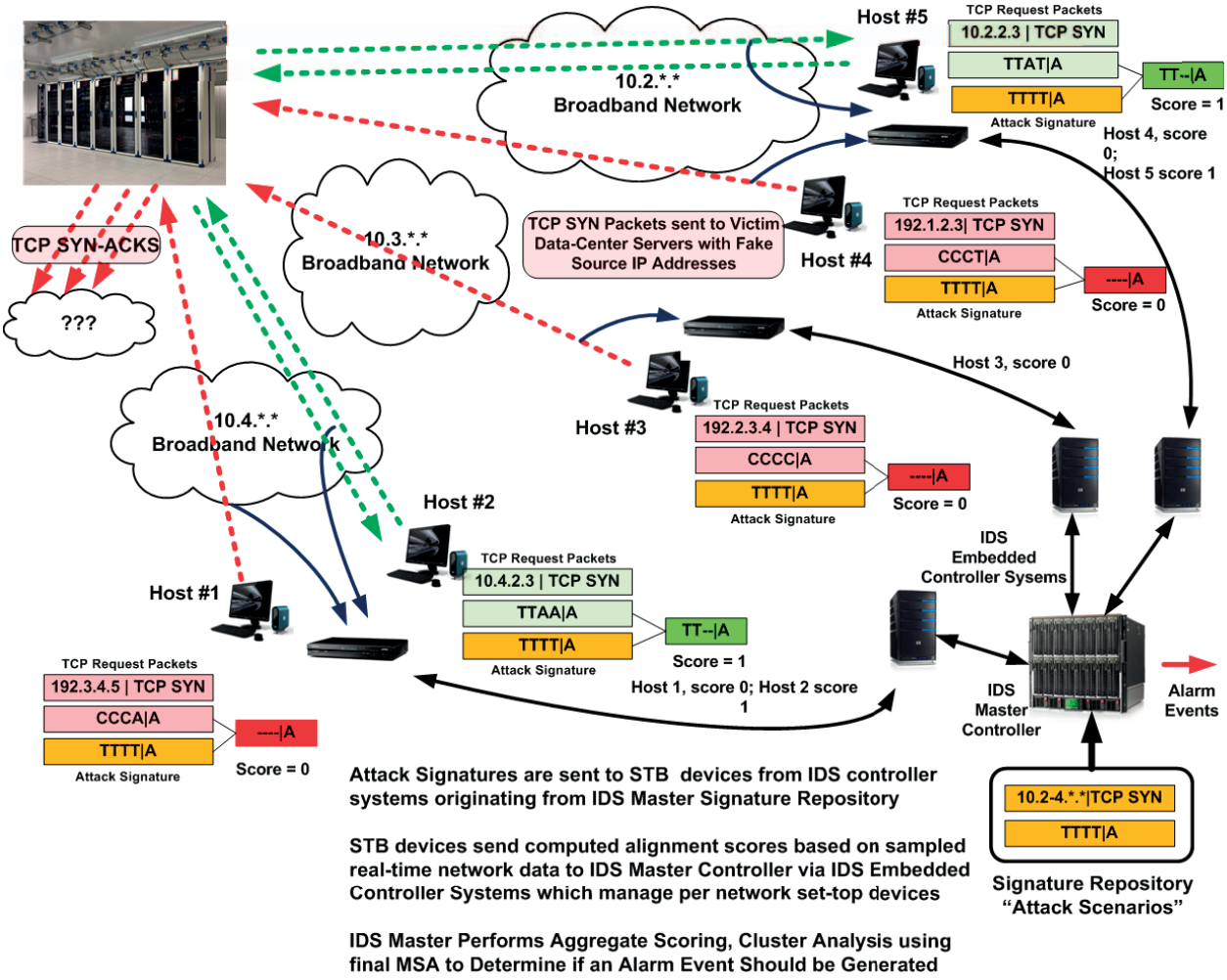
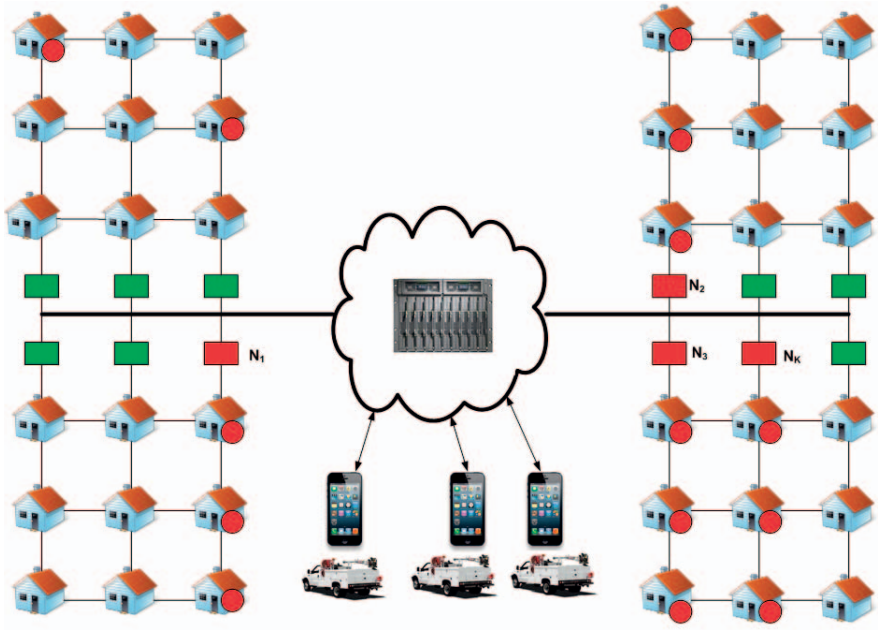
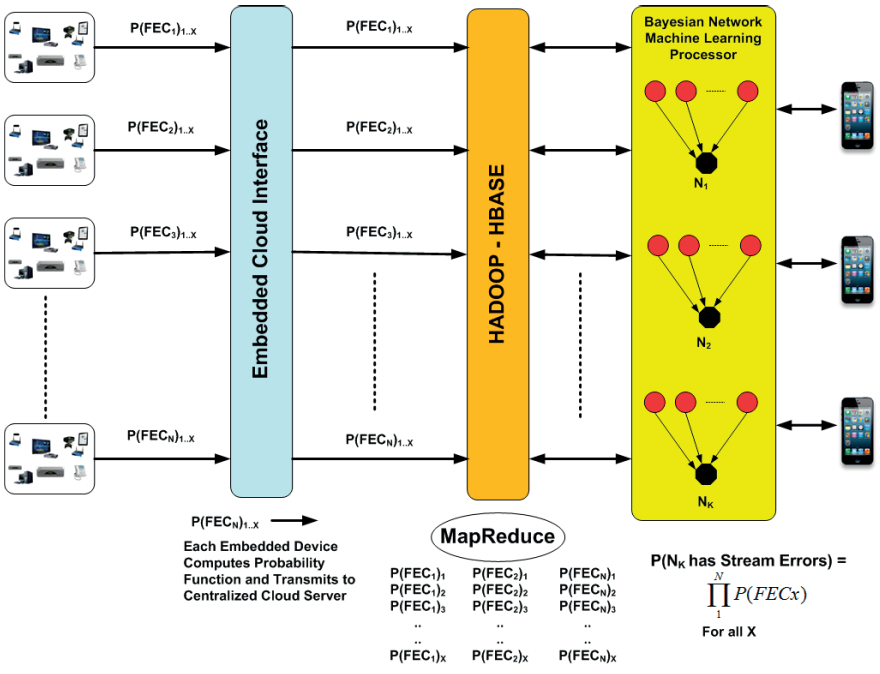


Figure 8.7: System architecture for a broadband network Intrusion Detection System.

scenario (in this case TCP SYN flood attack represented as TTTTA) with outgoing network traffic from potential attack computers coexisting on the set-top network. The figure shows resulting positive alignment scores corresponding to two attack hosts #2 and #5. These scores are sent to the master IDS controller which generates an alarm delivered as a Cloud web service to subscribing clients. A powerful aspect of this architecture lies in the use of broadband embedded computing to monitor and analyze multiple networks simultaneously by leveraging millions of embedded devices which act collectively as a single distributed network analyzer. The ability to execute parallel MSA across a large-scale broadband network of embedded devices as described in Chapter 5 enables the detection of distributed denial-of-service attacks (DDOS) and coordinated-attack scenarios across multiple networks concurrently. As a heterogeneous Cloud service, multiple service provider Cloud systems may coordinate among one another, providing a fully distributed IDS system that is capable of detecting network attacks originating over vast geographical areas.



(a)



(b)

Figure 8.8: Predictive Broadband Plant Monitoring: (a) Physical Network; (b) Heterogeneous Cloud Monitoring Architecture

8.4.4 Broadband Plant Monitoring

As a final example, I propose a system to predict physical broadband network degradation through the use of broadband embedded computing in conjunction with a centralized Cloud data-center infrastructure.

To proactively manage network degradation is an important service provider goal, as any reduction in physical network system robustness impacts negatively the customer experience. For example, faulty broadband network components, such as the RF nodes discussed in Section 2.4.2, impair streaming video quality or computer network access performance. In the worst case, complete interruption of service is possible. The top of Fig. 8.8 illustrates the relationship between broadband network impairments and cable system nodes (shown as green or red rectangles) relative to a hypothetical centralized Cloud data-center. Homes connected to green nodes are operational. A red dot within a given home indicates some impairment. In the case of a home with a red dot connected to a green node, the impairment is local to the home. e.g. the issue is within the home. However, faulty nodes (indicated in red) represent an impairment effecting homes that reside over a larger contiguous geographical area. In the example of Fig. 8.8, all nodes shown in red are faulty, along with the homes connected to the corresponding node.

A system architecture that predicts which geographical regions will likely suffer an impairment from one or more faulty broadband components is illustrated at the bottom of Fig. 8.8. The system operates as follows. Broadband embedded devices continuously monitor physical networks simultaneously capturing STB chip-level video and network error counters. Counter values are utilized in the computation of probability statistics, such as forward error correction rates that provide evidence of an emerging network impairment. The system utilizes broadband embedded computation to generate the continuous stream of probabilities for Cloud processing. These are indicated in the figure as $\mathrm{P(FEC}_n)_{1..x}$, where the n^{th} embedded device computes and transmits 1..x probability values to the centralized Cloud data-center systems. Referring to the bottom of Fig. 8.8, the centralized Cloud utilizes Hadoop and MapReduce processing to organize the embedded device data into a format that can be processed by a machine learning system. The system architecture presented in Chapter 6 describes methods to implement the Hadoop and MapReduce system illustrated in Fig. 8.8. The actual impairment prediction is computed within the Cloud server by a machine learning subsystem that accesses the HBASE database of sorted probability

values computed during the MapReduce phase. In this example, a Bayesian Network machine learning processor, illustrated as a *Simple Bayesian Network*, is used to predict the probability that a given broadband network node is likely to suffer an impairment.The resulting prediction statistics are then used by field operational teams to perform any necessary plant repairs.

8.5 Related Works

The evolution towards heterogeneous Cloud computing emerged from the use of GPU coprocessors alongside traditional blade server systems [24] to augment overall Cloud system processing capabilities. However, the diversity of heterogeneous Cloud systems and applications is expanding. This trend is acknowledged by Bonomi *et al.* with the work on Fog computing [16] and further illustrated through numerous application examples such as those from vehicle, robotic, WSN, and mobile Cloud computing [83; 21; 84; 29].

Similar to Fog computing, this chapter presented the use of embedded computing resources at the network edge. However, unlike Fog computing and the other related works, this chapter presents an architecture based on the implementation of real heterogeneous computing systems [98; 79]. Examples are then proposed that demonstrate a new class of heterogeneous Cloud applications that are unique to large-scale broadband service providers.

8.6 Summary

I presented a system and network architecture that supports the implementation of a heterogeneous Cloud system based on broadband embedded. The overall architecture achieves massive scale, based on the replication of multiple facilities called remote-hubs and data-centers, effectively partitioning the massive system into many smaller subsystems. Remote-hubs contain edge networking elements that provide physical network connectivity between embedded devices and regional data-center server systems. The regional data-centers are all interconnected over very high-performance fiber-optic networks. The integration of data-center system clusters with broadband embedded devices results in the creation of a new form of heterogeneous cloud systems architecture. The OERTE software is developed enabling heterogeneous Cloud software execution across data-center and embedded devices. OERTE virtualizes the embedded platform runtime environment into the

equivalent data-center software platform. The resulting distributed computing framework enables the execution of heterogeneous Cloud applications across both data-center servers and broadband embedded devices using standard distributed programming models such as MPI and MapReduce. The heterogeneous Cloud system platform supports the development of a new class of distributed applications reflecting the emergence of distributed broadband embedded devices sharing network connectivity with data-center systems and other connected devices. Finally, four examples of proposed heterogeneous Cloud applications are described. The application scenarios range from personalized consumer experiences to operational systems for intrusion detection and network monitoring, demonstrating the potential capabilities of heterogeneous Cloud systems when combined with broadband embedded computation.

Chapter 9

Conclusions and Future Work

The goal of my research has been to investigate and develop methods that extend the emerging class of heterogeneous Cloud systems by integrating the concept of broadband embedded computing. To support this thesis, my dissertation research entailed the investigation and implementation of fundamental contributions required to implement a heterogeneous Cloud system based on the utilization of a typical broadband embedded system. Two heterogeneous computer system platforms based on MPI and MapReduce were built that leverage broadband embedded computation. Broadband embedded computing experiments utilized the service provider STB, an embedded device that is deployed in very large numbers, and, therefore, is representative of the envisioned system scale of future heterogeneous Cloud systems. Numerous experiments were completed including the execution of real-world bioinformatics, image rendering, standard communication benchmarks, and extensive network simulation that validate the thesis and in some cases expose its limitations. To demonstrate the utility of my thesis goals, a fully scaled, service provider heterogeneous Cloud system was presented and a number of real-world applications of my vision were described. Throughout the dissertation research, a number of challenges that indicate performance limitations or scalability concerns were identified. Contributions that overcome these challenges were discussed and implemented, whereas others are left as opportunities for future work. Based on the summary of results, the evidence is conclusive that heterogeneous Cloud systems based on broadband embedded computing are not only feasible, but can be implemented across a compelling set of use-cases. In concluding this dissertation, the following sections summarize key contributions, results and future work.

9.1 Contributions

In support of my thesis goals, this dissertation includes the following contributions:

- The definition of **broadband embedded computing** as the useful and practical execution of application processes on service provider embedded devices across managed broadband networks to computationally augment and extend the taxonomy of heterogeneous Cloud systems platforms.

- **The quantitative demonstration of the feasibility of broadband embedded computing** through the development of an experimental system to measure available computational, memory, network and device uptime characteristics over an extensive period of time. Results show that typical service provider broadband embedded devices operate continuously up to 30 days at a time, and utilize a small fraction of application processor and network resources. Processor resources consume less than 50% utilization, while network bandwidth utilization is less than 1%.

- The implementation of a first-generation **heterogeneous system utilizing broadband embedded computing and based on MPI** to experimentally validate the composition of broadband embedded devices and centralized Linux cluster servers. Experiments validated both the feasibility of a heterogeneous Cloud systems platform based on broadband embedded computing and the scaling potential through the execution of the ClustalW MSA algorithm. Experiments exposed scaling challenges inherent to a data-center class runtime environment that is not optimized for executing applications across a large-scale distributed embedded system. The suitability of the standard MPI software platform associated with resource-constrained embedded devices was investigated.

- The implementation of a second-generation system for broadband embedded computing based on Open MPI that introduces a novel **virtualization model for embedded devices enabling scalable launch and execution of MPI applications**. The new system enables the implementation of large-scale runtime systems, and process lifecycle management across millions of devices. The virtualization system leads to the development of the **Open Embedded Runtime Environment** or OERTE. To overcome limitations discovered in the

first-generation system, additional contributions include the **development of an optimized Open MPI library for resource constrained embedded devices** that is fully compatible with the standard Open MPI library for interoperability with centralized Cloud system data-center servers. Extensive experiments including multiple workloads such as MSA, ray-tracing, and IMB communication benchmarks are carried out to validate the contributed system architecture and software frameworks.

- A **broadband embedded computing system for MapReduce utilizing Hadoop** extends the contributed software framework to enable the integration of broadband embedded devices that support Java into the ubiquitous MapReduce Cloud service delivery model. A key contribution includes **a method for porting the Hadoop Java software system to the embedded device to overcome the incompatibility between embedded and enterprise JVM environments**. MapReduce execution on broadband embedded STB devices produce consistent experimental results across both Linux and embedded devices. However, network and I/O experimental results show performance issues with the Hadoop MapReduce execution and replication environment under broadband network systems, suggesting an area of optimization.

- A **quantitative analysis of broadband network characteristics** to determine: **1) the design of a highly-scalable application process launch and execution runtime system, and 2) optimal broadband communication patterns during application execution, given message size, and communication flow characteristics**. Both lab and simulation experiments confirm the optimal properties of multicasting versus unicast message delivery.

- The **architecture for a highly-scalable heterogeneous Cloud system based on broadband embedded computing** for service providers is described and illustrated. To demonstrate the potential of the system, four application scenarios are described to motivate future work.

9.2 Future Work

While the dissertation introduced and covered a range of key topics enabling the implementation of a variety of heterogeneous Cloud system architectures, a number of challenges and emerging applications open up several research opportunities.

Optimized communication libraries. Chapter 7 confirmed ideal scenarios for leveraging multicast versus unicast message delivery. Communication libraries that underly both MPI and MapReduce runtime management systems as well as collective message operations can be optimized for additional performance. How to best utilize adaptive techniques that take advantage of multicasting and unicast message delivery characteristics within a broadband network environment remains an open problem.

Replication and fault-tolerance. In systems like Hadoop MapReduce, data block fault-tolerance is achieved through a replication protocol that results in a chained communication pattern. In a broadband environment this results in performance degradation proportional to the number of replications as all communications must traverse through the centralized broadband router. An open question is how to improve performance if a unicast graph model of the replication pattern can be transformed into a set of multicast operations. The MapReduce framework inherently provides fault-tolerance. However, MPI is prone to failure in the presence of communication or system errors. This is especially true when executing MPI applications within distributed embedded system. Consequently, the addition of fault-tolerant MPI frameworks for distributed embedded systems is an open area of research.

Broadband network quality-of-service. Broadband networks support multiple layers of quality-of-service (QoS) in order to control traffic utilization. Runtime systems that operate across broadband networks as well as high-priority collective operations can achieve higher performance through the use of QoS facilities. Opportunities exist in refactoring the underlying MPI and MapReduce communication system to support QoS mechanisms to further improve the performance of broadband embedded computing systems.

Cloud middleware and service delivery models. Distributed computing frameworks such as MPI and MapReduce represent the lowest levels of infrastructure within a given Cloud computing platform. Generally, Cloud services are offered at higher levels of abstraction in order to offer SaaS and PaaS services at the complexity of Web based applications. High-level Cloud platform

extensions to the frameworks described in Chapters 5 and 6 are logical next steps. How to develop a fully transparent and abstract service interface that is consistent with a larger population of clients is one goal. Longer term, future work that includes integrating frameworks such as Openstack [71] would provide better interoperability between heterogeneous and non-heterogeneous Cloud systems.

Heterogeneous Cloud Systems and Embedded IoT Computing. Perhaps the most interesting and important area of future work includes the study of new heterogeneous Cloud applications and systems in the context of the emerging Internet-of-Things. A challenge exists in balancing the delegation of computation between the centralized Cloud data-center and devices at the network edge. Solving this open research area will enable service providers to enhance their managed networks to support, not millions, but billions of devices through a variety of broadband, wireless access points and emerging network technologies.

Appendices

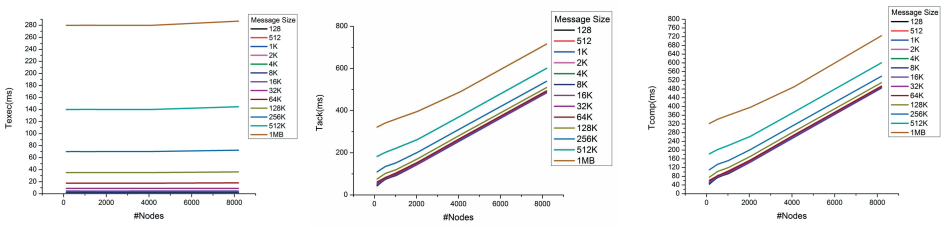
Appendix A

Broadband Network Experimental Results

This appendix includes results of all broadband network process launch-execution and parallel MSA experiments presented in Chapter 7.

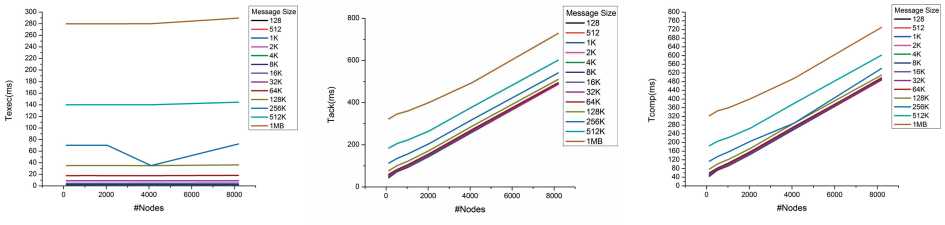
Process launch-execution experimental results for simulation are illustrated in Figures A.1 through A.8. Corresponding lab results are illustrated in Figures A.9 through A.16. Results are reported for the following scenarios: multicast-unicast and unicast-unicast scenarios, with/without traffic, and for both prewired/not-prewired connections.

Parallel MSA test results are also shown for both lab and simulation. Parallel MSA lab results are illustrated in Figures A.17 through A.20. Simulation results are illustrated in Figures A.21 through A.24. Test scenarios for parallel MSA include: multicast-unicast and unicast-unicast scenarios, with and without traffic.



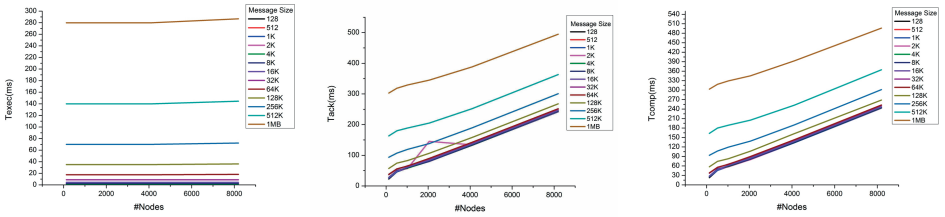
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.1: Multicast-unicast, not-prewired, no-traffic simulation scenarios.



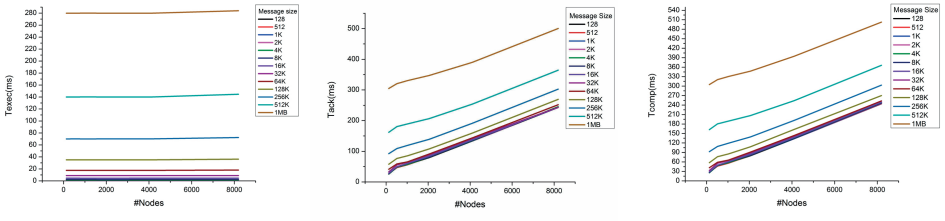
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.2: Multicast-unicast, not-prewired, traffic simulation scenarios.



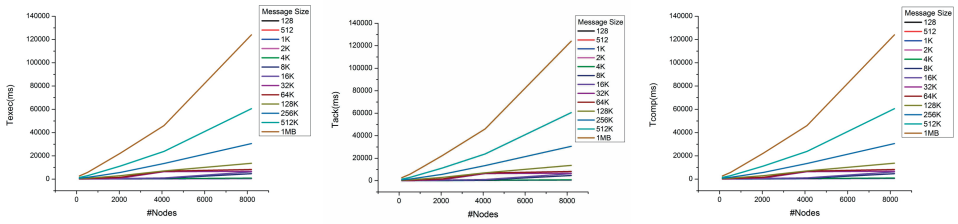
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.3: Multicast-unicast, prewired, no-traffic simulation scenarios.



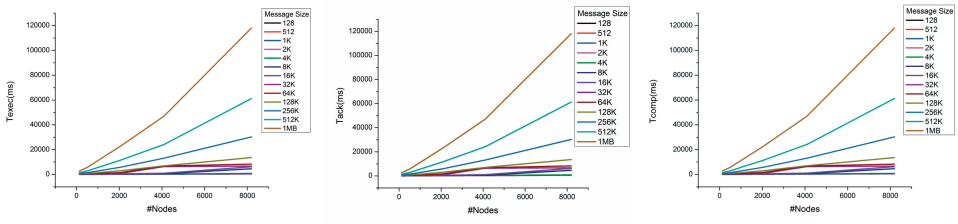
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.4: Multicast-unicast, prewired, traffic simulation scenarios.



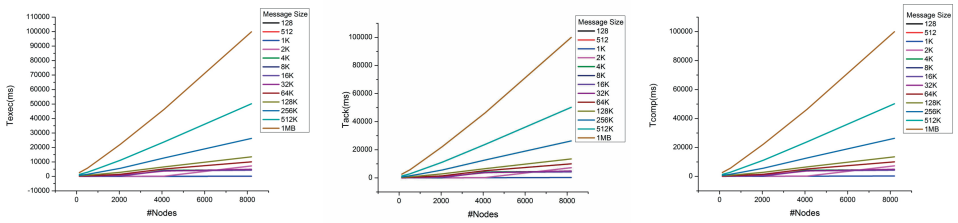
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.5: Unicast-unicast, not-prewired, no-traffic simulation scenarios.



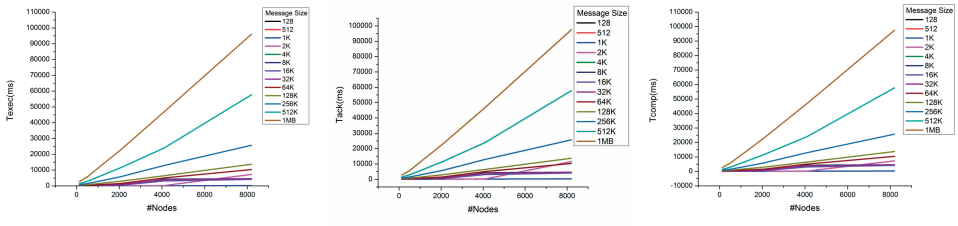
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.6: Unicast-unicast, not-prewired, traffic simulation scenarios.



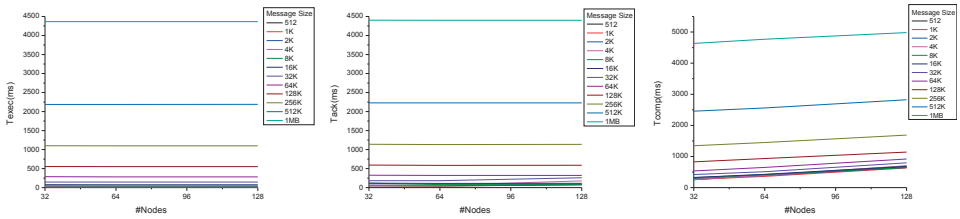
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.7: Unicast-unicast, prewired, no-traffic simulation scenarios.



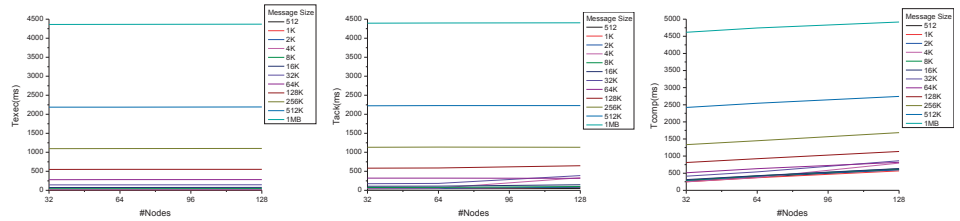
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.8: Unicast-unicast, prewired, traffic simulation scenarios.



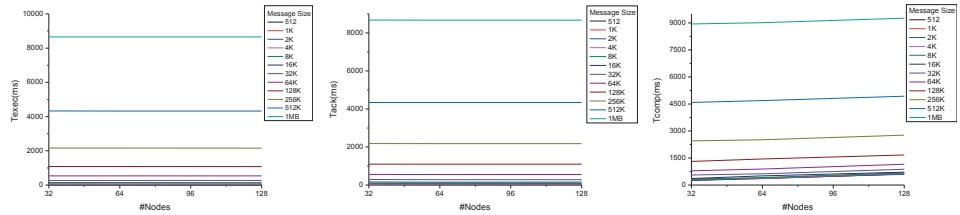
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.9: Multicast-unicast, not-prewired, no-traffic lab scenarios.



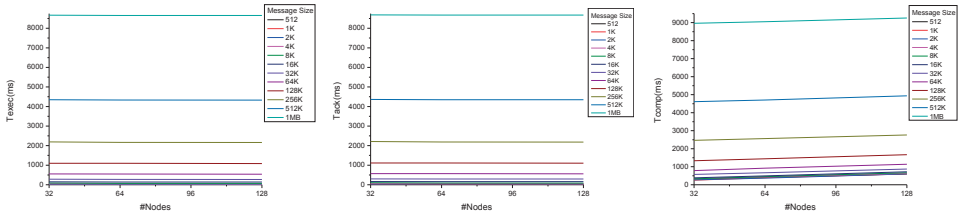
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.10: Multicast-unicast, not-prewired, traffic lab scenarios.



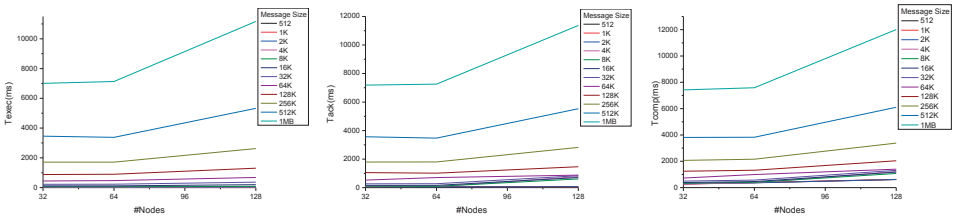
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.11: Multicast-unicast, prewired, no-traffic lab scenarios.



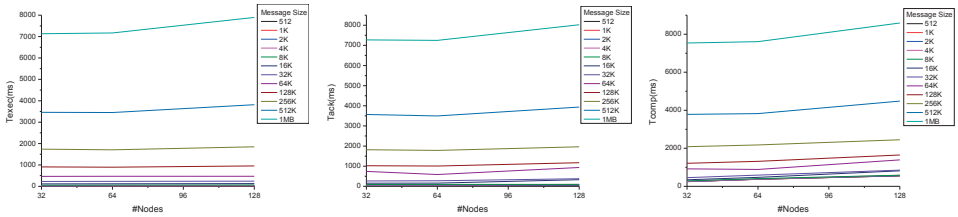
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.12: Multicast-unicast, prewired, traffic lab scenarios.



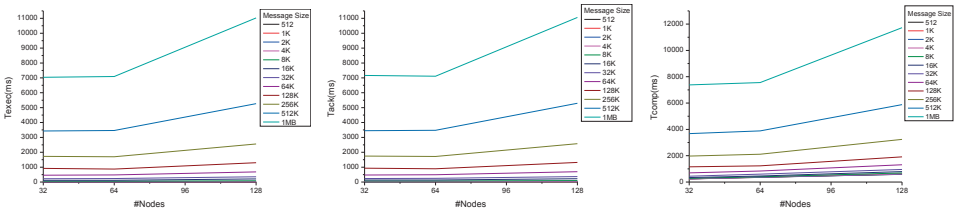
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.13: Unicast-unicast, not-prewired, no-traffic lab scenarios.



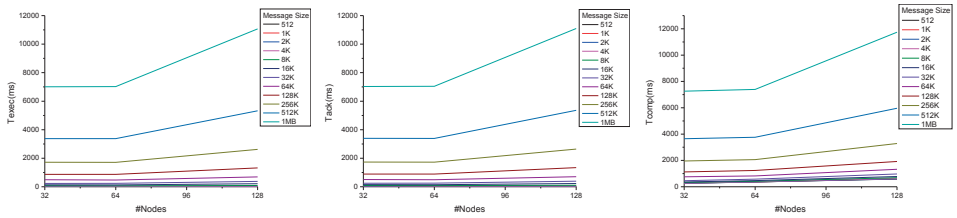
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.14: Unicast-unicast, not-prewired, traffic lab scenarios.



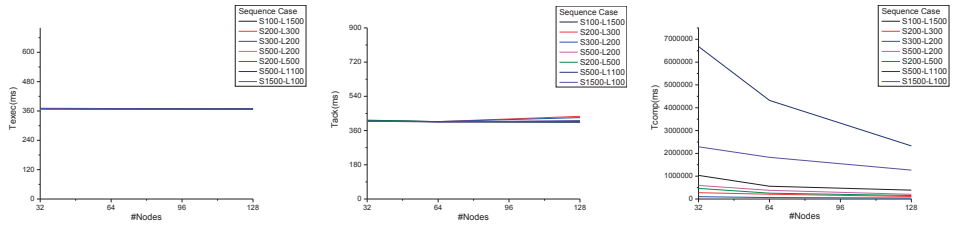
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.15: Unicast-unicast, prewired, no-traffic lab scenarios.



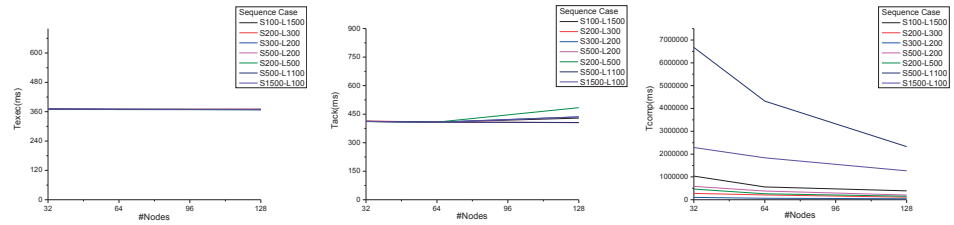
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.16: Unicast-unicast, prewired, traffic lab scenarios.



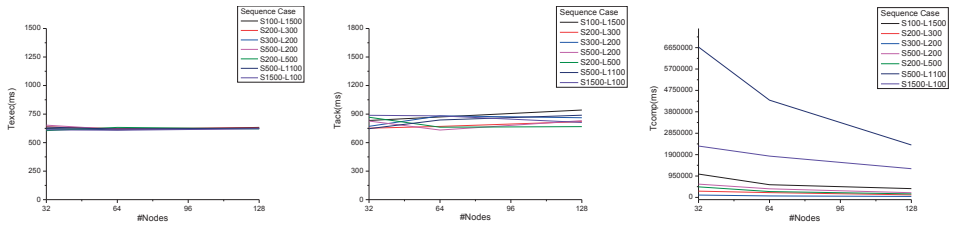
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.17: Parallel MSA multicast-unicast, no-traffic lab scenarios.



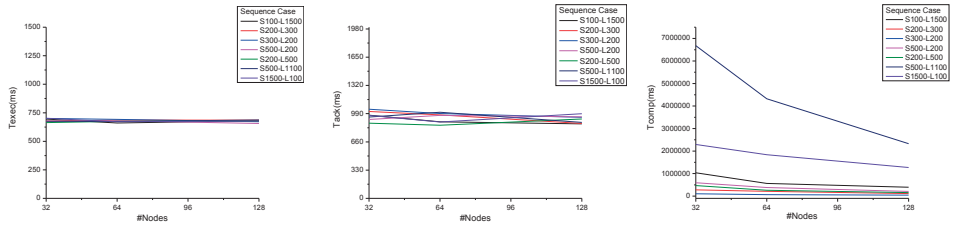
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.18: Parallel MSA multicast-unicast, traffic lab scenarios.



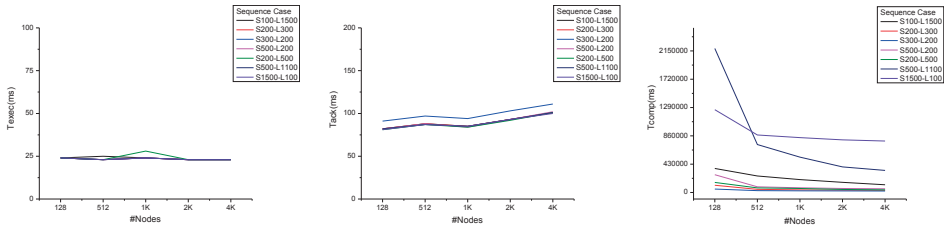
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.19: Parallel MSA unicast-unicast, no-traffic lab scenarios.



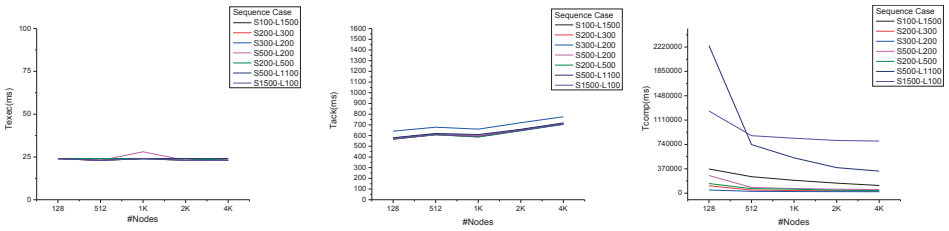
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.20: Parallel MSA unicast-unicast, traffic lab scenarios.



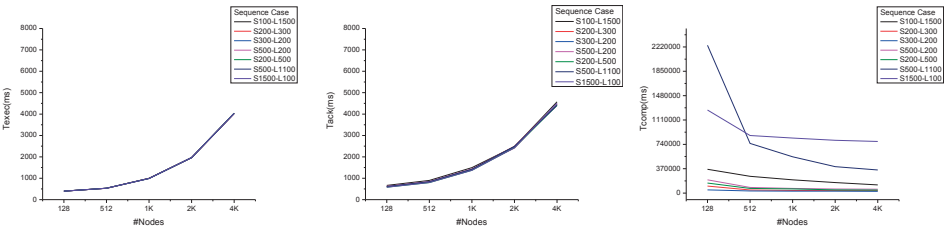
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.21: Parallel MSA multicast-unicast, no-traffic simulation scenarios.



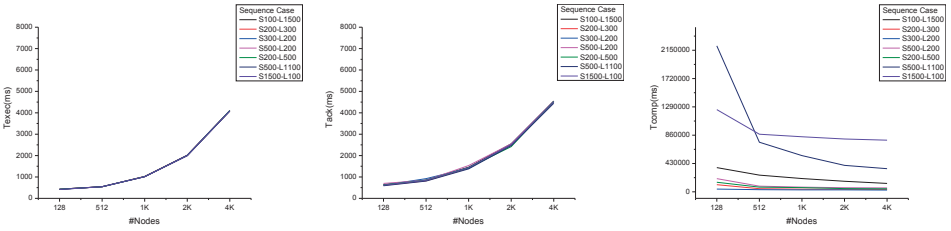
(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.22: Parallel MSA multicast-unicast, traffic simulation scenarios.



(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.23: Parallel MSA unicast-unicast, no-traffic simulation scenarios.



(b) Texec_N (c) Tack_N (d) Tcompletion_N

Figure A.24: Parallel MSA unicast-unicast, traffic simulation scenarios.

Bibliography

[1] "DOCSIS 1.1 part 1: Radio frequency interface specification," Society of Cable and Telecommunication Engineers, Tech. Rep. SCTE 23-1 2010.

[2] "`http://backport-jsr166.sourceforge.net`."

[3] "`http://retrotranslator.sourceforge.net`."

[4] "`http://www.cablelabs.com`."

[5] "`http://www.ietf.org`."

[6] "`http://www.tru2way.com`."

[7] A. Agbaria, D.-I. Kang, and K. Singh, "LMPI: MPI for heterogeneous embedded distributed systems," in *12th Intl. Conf. on Parallel and Dist. Sys.*, pp. 79–86, Jul. 2006.

[8] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *5th IEEE/ACM Intl. Workshop on Grid Computing*, pp. 4–10, Nov. 2004.

[9] D. P. Anderson and K. Reed, "Celebrating diversity in volunteer computing," *Hawaii Intl. Conf. on Sys. Sciences*, pp. 1–8, Jan. 2009.

[10] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," Tech. Rep., Feb. 2009.

[12] J. Arndt and C. Haenel, *Pi-Unleashed*. Springer, 2001.

[13] S. Bangay, "Experiences in porting a virtual reality system to Java," in *Proc. of the 1st Intl. Conf. on Comp. Graphics, Virtual Reality and Visualisation*, pp. 33–37, Nov. 2001.

[14] C. E. C. F. Batista, T. C. dos Anjos, D. Omaia, T. M. U. de Araújo, F. V. Brasileiro, and G. L. de Souza Filho, "Tvgrid: A grid architecture to use the idle resources on a digital tv network," in *Proc. of the 14th Brazilian Symposium on Multimedia and the Web*, pp. 130–137, Oct. 2008.

[15] M. Black and W. Edgar, "Exploring mobile devices as grid resources: Using an x86 virtual machine to run BOINC on an iPhone," in *Proc. of the 10th IEEE/ACM Intl. Conf. on Grid Computing*, pp. 9–16, Oct. 2009.

[16] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile Cloud Computing*, pp. 13–16, Aug. 2012.

[17] D. Borthakur, "The Hadoop distributed file system: Architecture and design," *Hadoop Project Website*, 2007.

[18] G. Bosilca, T. Herault, A. Rezmerita, and J. Dongarra, "On scalability for MPI runtime systems," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, pp. 187–195, Sep. 2011.

[19] A. Bukhamsin, M. Sindi, and J. Al-Jallal, "Using the Intel MPI benchmarks (IMB) to evaluate MPI implementations on an Infiniband Nehalem Linux cluster," in *Proc. of the Spring Simulation Multiconference*, pp. 1–4, Apr. 2010.

[20] G. Chandrasekaran, M. Hawa, and D. Petr, "Preliminary performance evaluation of QoS in DOCSIS 1.1," Information Telecommunication and Technology Center, University of Kansas, Lawrence, KS, Tech. Rep. ITTC-FY2003-TR-22736-01, Apr. 2003.

[21] D. Cook and S. Das, *Smart Environments: Technology, Protocols and Applications (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2004.

[22] I. D. Corporation, "Extracting value from chaos," Jun. 2011.

[23] S. Coull, J. Branch, B. Szymanski, and E. Breimer, "Intrusion detection: A bioinformatics approach," in *19th Annual Computer Security Applications Conferences*, pp. 8–12, Dec. 2003.

[24] S. P. Crago, K. Dunn, P. Eads, L. Hochstein, D.-I. Kang, M. Kang, D. Modium, K. Singh, J. Suh, and J. P. Walters, "Heterogeneous cloud computing." pp. 378–385, 2011.

[25] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[26] A. Datta and J. Ebedes, "Multiple sequence alignment in parallel on a workstation cluster," in *Parallel Computing for Bioinformatics and Computational Biology*, ser. Series on Parallel and Dist. Computing, A. Zomaya, Ed. J. Wiley & Sons, 2006, ch. 8, pp. 193–210.

[27] F. Dawson, *Optimizing the Future with Next-Generation DOCSIS*. SCTE, 2003.

[28] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[29] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, 2011.

[30] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu, "Characterizing residential broadband networks," in *Proc. of the ACM/USENIX Internet Measurement Conf. (IMC'07)*, Oct. 2007.

[31] A. Dou *et al.*, "Misco: a mapreduce framework for mobile systems," in *Proc. of the 3rd Intl. Conf. on Pervasive Tech. Related to Assistive Environments*, pp. 32:1–32:8, Jun. 2010.

[32] J. Ebedes and A. Datta, "Multiple sequence alignment in parallel on a workstation cluster," *Bioinformatics*, vol. 20, no. 7, pp. 1193–1195, May 2004.

[33] P. R. Elespuru, S. Shakya, and S. Mishra, "MapReduce system over heterogeneous mobile devices," in *Proc. of the 7th IFIP WG 10.2 Intl. Workshop on SW Tech. for Embedded and Ubiquitous Sys.*, pp. 168–179, Nov. 2009.

[34] *ST Microelectronics and ARM Team Up to Power Next-Generation Home Entertainment*, "Press release: `www.arm.com/news/26284.html`," Oct. 2009.

[35] R. Esteves, "A taxonomic analysis of cloud computing," 2011.

[36] C. Evans, *Future of Google Earth*. Booksurge Llc., 2008.

[37] D. Feng and R. Doolittle, "Progressive sequence alignment as a prerequisite to correct phylogentic trees," *J. Mol. Evol.*, vol. 25, no. 4, pp. 351–360, Aug. 1987.

[38] K. Fuerlinger, C. Klausecker, and D. Kranzlmüller", ""the AppleTV-Cluster: Towards energy efficient parallel computing on consumer electronic devices"," "2011".

[39] E. Gabriel *et al.*, "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users Group Meeting*, pp. 97–104, Sep. 2004.

[40] F. Gatta, R. Gomez, Y. Shin *et al.*, "An embedded 65 nm CMOS baseband IQ 48 MHz-1 GHz dual tuner for DOCSIS 3.0," *Comm. Mag.*, vol. 48, no. 4, pp. 88–97, Apr. 2010.

[41] O. Gotoh, "An improved algorithm for matching biological sequences," *J. Mol. Biol.*, vol. 162, no. 3, pp. 705–708, Dec. 1982.

[42] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open MPI: A flexible high performance MPI," in *Proc. of 6th Ann. Intl. Conf. on Parallel Processing and Applied Mathematics*, Sep. 2005.

[43] T.-M. Grønli, J. Hansen, and G. Ghinea, "Android vs Windows Mobile vs Java ME: a comparative study of mobile development environments," in *Proc. of the 3rd Intl. Conf. on Pervasive Tech. Related to Assistive Env.*, pp. 45:1–45:8, Jun. 2010.

[44] L. Harte, *Introduction to Data Multicasting, IP Multicast Streaming for Audio and Video Media Distribution*. Althos, 2008.

[45] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, Mar. 1979.

[46] B. He *et al.*, "Mars: a MapReduce framework on graphics processors," in *Proc. of the 17th Intl. Conf. on Parallel Arch. and Compilation Tech.*, pp. 260–269, Oct. 2008.

[47] http://boinc.berkeley.edu/.

[48] http://en.wikipedia.org/wiki/DOCSIS.

[49] http://en.wikipedia.org/wiki/N-body-simulation.

[50] http://folding.stanford.edu.

[51] http://hadoop.apache.org.

[52] http://http://www.mpi-forum.org.

[53] http://setiathome.berkeley.edu.

[54] http://www.arm.com.

[55] http://www.broadcom.com.

[56] http://www.busybox.net.

[57] http://www.cablemodem.com.

[58] http://www.ciena.com.

[59] http://www.cisco.com.

[60] http://www.cisco.com.

[61] http://www.cisco.com.

[62] http://www.davic.org.

[63] http://www.gridrepublic.org.

[64] http://www.istc-cc.cmu.edu, "Foundations for future clouds."

[65] http://www.isuppli.com.

[66] http://www.kernel.org.

[67] http://www.mediabiz.com.

[68] http://www.morganstanley.com.

[69] http://www.ncta.com.

[70] http://www.ncta.org.

[71] http://www.openstack.org.

[72] http://www.point-topic.com.

[73] http://www.st.com.

[74] http://www.ti.com.

[75] http://www.w3.org/TR/ws-arch/wsa.pdf.

[76] G. Huerta-Canepa and D. Lee, "A virtual cloud computing provider for mobile devices," in *Proc. of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, pp. 6:1–6:5, Jun. 2010.

[77] K. Hwang, G. Fox, and J. Dongarra, *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Morgan Kaufmann, Oct. 2011.

[78] J.Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process tolerence for Open MPI," in *Intl. Symp. on Parallel and Dist. Processing*, pp. 415–422, Mar. 2007.

[79] Y. Jung, R. Neill, and L. P. Carloni, "A broadband embedded computing system for MapReduce utilizing Hadoop," in *CloudCom*, pp. 1–9, Dec. 2012.

[80] S. Kågström, H. Grahn, and L. Lundberg, "Cibyl: an environment for language diversity on mobile devices," in *Proc. of the 3rd Intl. Conf. on Virtual execution environments*, pp. 75–82, Jun. 2007.

[81] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, pp. 589–604, Jul. 2005.

[82] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proc. of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 938–948, Jan. 2010.

[83] S. Kumar, S. Gollakota, and D. Katabi, "A cloud-assisted design for autonomous driving," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 41–46, Aug. 2012.

[84] W. Kurschl and W. Beer, "Combining cloud computing and wireless sensor networks," in *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pp. 512–518, Dec. 2009.

[85] K. Li, "ClustalW-MPI: ClustalW analysis using distributed and parallel computing," *Bioinformatics*, vol. 19, no. 12, pp. 1585–1586, Aug. 2003.

[86] M.-L. Li *et al.*, "The ALPBench benchmark suite for complex multimedia applications," *IEEE Workload Characterization Symposium*, vol. 0, pp. 34–45, Oct. 2005.

[87] P. Loshin, "Intrusion detection," Apr. 2001.

[88] G. Maier, A. Feldmann, V. Paxson, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *Proc. of the 9th ACM SIGCOMM Conf. on Internet measurement Conf.*, pp. 90–102, Nov. 2009.

[89] J. Martin and N. Shrivastav, "Modeling the DOCSIS 1.1/2.0 mac protocol," in *Computer Communications and Networks, 2003. ICCCN 2003. Proc. The 12th Intl. Conf. on*, pp. 205 – 210, Oct. 2003.

[90] J. Martin, "Modeling the DOCSIS 1.1/2.0 mac protocol," 2003.

[91] T. McMahon and A. Skjellum, "eMPI: Embedded MPI," in *MPI Developers Conf.*, pp. 180–184, Jul. 1996.

[92] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Tech. Rep. NIST 800-145, 2011.

[93] F. P. Miller, A. F. Vandome, and J. McBrewster, *BD-J: Java Platform, Micro Edition, Connected Device Configuration, Blu-ray Disc, Globally Executable MHP, DVD, Blu-ray Disc Association, PlayStation 3, Interactive television.* Alpha Press, 2009.

[94] C. Min, K. Kim, H. Cho, S. Lee, and Y. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. of the 10th USENIX Conf. on File and Storage Tech.*, Feb. 2012.

[95] C. Moretti, K. Steinhaeuser, D. Thain, and N. Chawla, "Scaling up classifiers to cloud computers," in *8th IEEE Intl. Conf. on Data Mining*, pp. 472 –481, Dec. 2008.

[96] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequences of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, June 1970.

[97] R. Neill and L. Carloni, "A scalable architecture for intrusion-detection systems based on a broadband network of embedded set-top boxes," in *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pp. 1–4, Aug. 2011.

[98] R. Neill, L. P. Carloni, A. Shabarshin, V. Sigaev, and S. Tcherepanov, "Embedded processor virtualization for broadband grid computing," in *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing (Grid)*, pp. 145–156, Sep. 2011.

[99] R. Neill, A. Shabarshin, and L. P. Carloni, "A heterogeneous parallel system running open mpi on a broadband network of embedded set-top devices," in *Proceedings of the 7th ACM international conference on Computing frontiers*, pp. 187–196, May 2010.

[100] Open MPI, "`http://www.open-mpi.org`."

[101] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya, "Reliable multicast transport protocol (rmtp)."

[102] R. Pinilla and M. Gil, "ULT: a Java threads model for platform independent execution," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 4, pp. 48–62, Oct. 2003.

[103] Rhoton, John, (Autor), Haukioja, and Risto, *Cloud computing architected*, ser. Solution design handbook). [UK]: Recursive Press, 2011.

[104] V. Roca, "The MCL multicast library: Concepts, architecture and use," 2001.

[105] J. Rosenberg and A. Mateos, *The Cloud at Your Service*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[106] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogentic trees," *Mol. Biol. Evol.*, vol. 4, no. 4, pp. 406–425, Jul. 1987.

[107] H. E. Schaffer, "X as a service, cloud computing, and the need for good judgment," *IT Professional*, vol. 11, pp. 4–5, 2009.

[108] M. Schoeberl, S. Korsholm, T. Kalibera, and A. P. Ravn, "A hardware abstraction layer in Java," *ACM Trans. Embedded Comp. Sys.*, vol. 10, no. 4, pp. 42:1–42:40, Nov. 2011.

[109] N. Shah, D. Kouvatsos, J. Martin, and S. Moser, "A tutorial on DOCSIS: protocol and performance models," in *In: Proc. of the Intl.*, 2005.

[110] W. R. Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocolls.* Addison-Wesley, 1996.

[111] J. Stone, "An efficient library for parallel ray tracing and animation," In Intel Supercomputer Users Group Proc., Tech. Rep., 1995.

[112] J. Stoye, D. Evers, and F. Meyer, "Rose: generating sequence families," *Bioinformatics*, vol. 14, no. 2, pp. 157–163, Mar. 1998.

[113] N. Thanh-Cuong, S. Wen-Feng, C. Ya-Hui, and X. Wei-Min, "Research and implementation of scalable parallel computing based on map-reduce," *Journal of Shanghai Univ.*, vol. 15, no. 5, pp. 426–429, Aug. 2011.

[114] J. Thompson, D. Higgins, and T. Gibson, "ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Res.*, vol. 22, no. 22, pp. 4673–4680, Nov. 1994.

[115] D. Toth, "Volunteer computing with video game consoles," in *Proc. of the 6th WSEAS Intl. Conf. on SW Eng., Parallel and Dist. Sys.*, pp. 102–106, Feb. 2007.

[116] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 50–55, Dec. 2008.

[117] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.