7-1-2011

Parallel network protocol stacks using replication

Charles Donour Sizemore

Charles Donour Sizemore

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

[illegible], Chairperson

[illegible]

[illegible]

[illegible]

Parallel Network Protocol Stacks Using Replication

by

Charles Donour Sizemore

B.S., Mathematics, University of Chicago, 2003

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 13, 2011

Dedication

To those always looking to go a little faster

Delayed Acknowledgments

Only years of collaboration make this work possible. Although one name has to go at the top, this dissertation would not be complete without acknowledging a host of characters.

First and foremost, I thank my dissertation committee: Prof. Patrick Bridges, Prof. Dorian Arnold, Prof. Jed Crandall, and Prof. Nasir Ghani. Their guidance and feedback, throughout the process, has been essential. When I go off the deep end, they reel me back.

Thanks go to the entire Scalable Systems Lab at the University of New Mexico. It is no stretch to call them my second committee. Thank you Barney Maccabe, Patrick Widener, Wenbin Zhu, Eric Nelson, James Horey, Edgar Leon, Manju Venkata, Kurt Ferreira, Philip Soltero, Taylor Groves, Zheng Cui, Ricardo Villalon, and Scott Levy.

I thank Intel and Sun Microsystems for their support: financial, professional, and personal. The Solaris Networking team at Sun, now Oracle, has been extremely supportive of the work and of my career in general. Thank you Nicolas Droux, Markus Flierl, Sunay Tripathi, Greg Lavender and Eric Cheng.

Finally, I thank my family for their unending love and support. Although we are spread around the world, we are connected by our crazy ideas. Thanks Mom, Dad, Meg, and Owen.

Parallel Network Protocol Stacks Using Replication

by

Charles Donour Sizemore

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 13, 2011

Parallel Network Protocol Stacks Using Replication

by

Charles Donour Sizemore

B.S., Mathematics, University of Chicago, 2003

Ph.D., Computer Science, University of New Mexico, 2011

Abstract

Computing applications demand good performance from networking systems. This includes high-bandwidth communication using protocols with sophisticated features such as ordering, reliability, and congestion control. Much of this protocol processing occurs in software, both on desktop systems and servers. Multi-processing is a requirement on today's computer architectures because their design does not allow for increased processor frequencies. At the same time, network bandwidths continue to increase. In order to meet application demand for throughput, protocol processing must be parallel to leverage the full capabilities of multi-processor or multi-core systems. Existing parallelization strategies have performance difficulties that limit their scalability and their application to single, high-speed data streams.

This dissertation introduces a new approach to parallelizing network protocol processing without the need for locks or for global state. Rather than maintain global states, each processor maintains its own copy of protocol state. Therefore, updates are local and don't require fine-grained locks or explicit synchronization. State management work is replicated, but logically independent work is parallelized. Along with the approach, this dissertation describes Dominoes, a new framework for implementing replicated processing systems. Dominoes organizes the state information into Domains and the communication into Channels. These two abstractions provide a powerful, but flexible model for testing the replication approach.

This dissertation uses Dominoes to build a replicated network protocol system. The performance of common protocols, such as TCP/IP, is increased by multiprocessing single connections. On commodity hardware, throughput increases between 15-300% depending on the type of communication. Most gains are possible when communicating with unmodified peer implementations, such as Linux. In addition to quantitative results, protocol behavior is studied as it relates to the replication approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Networking is one of the most important services that any computing system provides. Consumers now have access to streaming media on televisions, desktops, laptops, mobile devices, and in their vehicles. Cloud storage systems allow users to retrieve data from any Internet-connected device. Enterprise systems have large databases that often need to be transferred for backup or to handle shifting workloads. All of these applications must move data quickly.

Recent hardware developments require parallelized software to achieve good networking performance. Good performance is crucial to enable current and future generations of applications, as well as to reduce the costs associated with deploying these applications. Fortunately, networking hardware performance continues to increase with Moore's Law. However, it is a major challenge to provide networking software that can keep pace with hardware capabilities. Without the software support, application performance stagnates.

This dissertation presents a replication approach for parallelizing network protocols. My thesis is that parallelizing the network protocol stack using replication increases throughput on fast links and on multi-core systems. Replication allows for the full use of computational resources to achieve high throughput, even with single network connections. To demonstrate the thesis, this work describes Dominoes, a replication-based framework for operating system services. Dominoes allows the user to build traditional system services with a novel, lock-free queuing system and explicit consistency management. Dominoes is used to implement high-speed network protocol processing. This illustrates the strengths and weaknesses of the replication approach as well advancing the state-of-the-art of networking.

1.1 Applications

There are many computing applications which demand high-performance networking, yet are not able to fully exploit the hardware resources available. The applications vary in that they span the breadth of computational domains, from consumer electronics to supercomputing. A common theme for these applications is that they require maximum performance from the available hardware. This will continue as hardware gets faster. In almost all cases, this requires significant computational capacity.

On example application is Network Attached Storage (NAS). NAS devices have begun to displace local disks for storing large files. Sometimes these devices are accessed directly from network clients. More often, they are accessed through file servers which provide security, reliability, encryption, etc. Each of these features introduce heavy computational demands on the network data stream. Like conversing through a translator, this processing must keep up with the network or the applications will spend most of their time waiting.

Complicated data descriptions necessitate high-speed processing. On enterprise systems, large data warehouse applications such as Netezza, LexisNexis, and Greenplum require high bandwidth, reliable network connections to synchronize and replicate their backends. The storage systems are often specialized, but the communication system is much the same as a local area network (LAN) environment. The systems use common protocols like TCP/IP to ensure that the backend data arrives reliably and in order. On high-performance computing (HPC) systems, systems like ADIOS [45] provide mechanisms for users to attach programmatic functions to I/O operations. This can be used to transform the data, perform diagnostics, or increase/decrease level of detail.

Perhaps the most compelling case for the increasing computational demand on networks is Quality of Service (QoS) processing. This includes bandwidth throttling, deep packet inspection, network observability, and traffic monitoring. Such systems require complex processing on very high bandwidth network links. Combining this with encryption or data compression results in a high per-packet processing cost. The performance of these applications is critical to providing a user experience that is either useful or enjoyable.

1.2 Network Protocol Processing

Network protocols are the standardized rules for exchanging data (messages) between computing systems. Protocols provide the features needed for distinct systems to communicate effectively. This may include addressing, routing, security, etc. A variety of protocols exist because applications have varying requirements. A file server may require that data arrive in strict ordering while a video game server may prioritize low latency. Applications such as Skype [4], may require that all data is encrypted in order to protect the privacy of its users. Without the protocol there is no method for marshalling the data to transmit and decoding it when it arrives.

Network protocol processing is an important but often overlooked component of digital communications. Many times, it is a computationally expensive component of a given system. Application developers often consider it to be part of the networking infrastructure while network designers consider it part of the application. Living in the operating system, it sits somewhere in between. Consequently, a great deal of the processing occurs in software.

Protocols can be divided into two broad categories: stateless and stateful. These categories have far reaching consequences for processing and performance. Both types are abundant on local networks, on the Internet, and in research environments.

Stateless protocol traffic usually has no dependence between packets. This allows multiple packets to be processed independently without coordination between processors. These are often characterized as connection-less. Such data streams may contain messages from multiple senders or for multiple recipients. Like a mailbox, the contents of connectionless transmissions may be from a variety of sources. These messages are frequently referred to as datagrams. Examples of stateless protocols include the User Datagram Protocol (UDP), Internet Control Message Protocol (ICMP), and the Hypertext Transfer Protocol (HTTP).

Stateful protocols, in contrast, contain session information that is updated when data is sent or received, the network conditions change, or the application intervenes. These are often referred to as connection-oriented protocols because the session refers to a connection between multiple, distinct parties. An everyday example of a connection is a telephone call. When two parties are connected over a telephone, only those two parties participate. The connection is a powerful concept for efficient delivery or reliable and ordered data. However, the state introduces significant complexity – especially in regards to parallel processing.

1.3 Multicore Implications

1.3.1 Moore s Law

Like the majority of the microelectronics industry, processor (CPU) performance has long been propelled by Moore's Law. For several decades, the growth in transistor counts translated to increasingly complex features, increased caches sizes, and above all, increased clock rates. This allowed host processors to perform network protocol processing ranging from cryptography to forward error correction to sophisticated acknowledgement and window processing at network line-rate speeds.

This trend, unfortunately, has ended. Recent CPU designs from Intel, AMD, IBM, and Sun, for example, have essentially the same clock speed and hardware architectures as previous generations but place more cores on each die. Ethernet and Infiniband NICs, however, continue to increase in speed, with 10 Gigabit NICs now common and 40 Gigabit NICs (e.g., 4x QDR Infiniband) being deployed. Similarly, future processor roadmaps call for more and more specialized but not faster cores, while network architects are already planning for and designing 100 Gigabit Ethernet NICs.

Figure 1.1 presents the per-core clock-rates and theoretical throughputs of PC CPUs and network interface cards (NICs). It illustrates the tremendous growth of both CPU performance and networking throughput from the early development of the transistor up to modern day. It is particularly striking to observe the clock-rate elbow that occurs circa 2006. The per-core performance shows very little gain, but the addition of cores guarantees a steady increase in aggregate chip performance. Likewise, the networking performance growth continues and is expected to continue into the foreseeable future.

Furthermore, Figure 1.1 presents a stark picture for software designers. It makes clear the ever widening gap between network capacity and processor core speed. Applications that are limited to a single core will not obtain meaningful speedups on future CPUs archi-

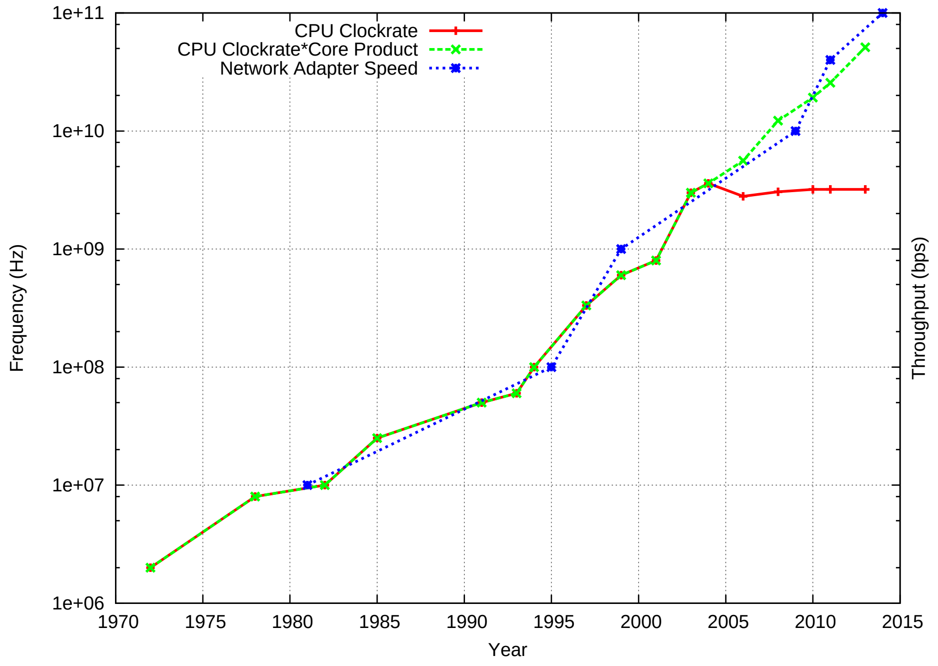


Figure 1.1: Historical Data of CPU and Network Speeds

tectures. Instead, they must utilize the full chip capacity, in aggregate, to process massive data streams that such systems are capable of delivering. That requires a solution that is not just concurrent, but scalable.

1.3.2 Parallelization

Applications need *strong scaling*, where a fixed workload is executed more quickly with increased core counts [16]. Multiple cores must closely coordinate activities and multi-core system designs which rely on only intermittent synchronization will bottleneck on inter-core synchronization. As a consequence, the resulting system performance can be disappointing. Some work proposes a distributed systems-oriented approach, where data structures are replicated or segregated between processors [5, 13]. These designs use a multiple-instruction/multiple-data (MIMD) parallel execution model, and generally focus on services for which coarse-grained synchronization is sufficient. Because of this, such systems generally provide only *weak scaling*, where more cores can be used to handle a larger workload (e.g., more TCP/IP connections) but cannot make an existing workload (e.g., a single TCP/IP connection) run faster.

Stateful protocols naturally have inter-packet dependencies. In the most simple form, a connection consists of initialization, data transmission, and termination. The ordering of those events is important and they cannot overlap. Any system that attempts to process the connection in parallel must keep track of the connection state and ensure that its operations are valid and consistent. Complex features increase these dependencies. For example, reliable data delivery requires senders to retransmit data if any is lost. In turn, this requires keeping track of what has been received because the state changes with time. This makes it difficult to parallelize the execution of such protocols.

Concurrently processing multiple packets that update the same state leads to contention on that state. In order to avoid corruption, synchronization is needed. The most common synchronization mechanism for shared data structures is the lock. Mutual exclusion lock (mutexes), reader-writer locks, and the like are used in virtually every network protocol system that supports multiprocessing. They have been very successful in handling unconnected network traffic as well as supporting many simultaneous connections. However performance for single connections has been a great challenge. Single connection performance is vital because so many applications depend on the features that connection-oriented protocols provide. As described in Chapter 2, this problem has been studied for many years, but no solution has been developed. Like the classical memory wall problem, this is a case of keeping the processor, indeed the entire system, fed with data.

1.4 Replication

This dissertation proposes a *replication-based* approach to multi-core network protocol stack design and implementation. In this approach, connection and protocol state is replicated across multiple cores, avoiding unnecessary locking, caching, and scheduling costs. Replication provides an alternative to the lock-based mechanisms that limit the performance of many system services, including networking. In this case, replication refers to both data replication (state) and computational replication (request processing). The replication approach allows for lock-free access to system resources and to the associated control state. It also eliminates the caching issues associated with shared state.

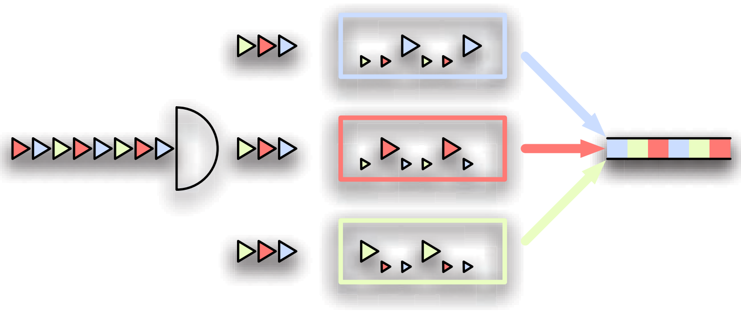


Figure 1.2: Replication Dividing a Stream Into Three Parts for Processing

Rather than have a single copy of network connection state, the replication approach creates many copies, one for each processor. The processors then process every incoming and outgoing network request. This allows them to keep their states up-to-date. However, there is an important distinction. Networking packets do not require full processing to update the state. For most protocols, only the header needs to be used. The rest of the processing, such as data delivery, does not have globally visible effects. Furthermore, analysis has shown that for many protocols, header processing is a small percentage of the overall computational load.

Replication systems are considerably more complex than single threaded (or single processor) alternatives. Replicated states require careful management in order to ensure that the separate copies remain consistent. Replication introduces new challenges such as efficient and scalable data delivery, load balancing, reordering, and consistency management.

To study and address these problems, this dissertation describes Dominoes, a framework for constructing replicated system services. Dominoes provides the infrastructure to study and solve many of these consistency problems. Network protocols are particularly well suited to these issues because network are inherently unreliable. Consequently the protocols must be robust and capable of recovering from errant behavior.

1.5 Contribution of This Dissertation

The contributions described in this dissertation are:

- A replication-based parallelization approach that improves the performance of network protocol processing, even for single connections;
- A new framework for building scalable operating system services, Dominoes, that provides a high-performance, replicated queue and the necessary infrastructure for implementing replicated request processing
- A replicated network protocol stack, using Dominoes and the Scout research operating system:

    * The behavior of the system, including consistency, is presented and analyzed.

    * The performance and bottlenecks of the system are analyzed.

- An analysis of network protocols features based on their applicability to the replication approach.

1.6 Dissertation Outline

The remaining chapters of the dissertation are organized as follows. Chapter 2 discusses related work. This includes network protocol processing in general, network performance, other replication systems, and distributed systems. Chapter 3 introduces the replication approach to parallel protocol processing, discusses the advantages, possible speedup, and challenges. This chapter also introduces Dominoes, a framework for implementing the approach. Chapter 4 describes a network implementation using the Dominoes architecture as well as describing the construction of a replicated network protocol stack built with the Scout research operating system. Chapter 5 gives a detailed analysis of performance. Chapter 6 analyzes the limitations and drawbacks of the system as well as protocol features that proved difficult to adapt. Chapter 7 concludes this document and discusses future work.

Chapter 2

Related Work

This chapter discusses previous work on network protocol processing, replication systems, and distributed systems in general. Section 2.1 describes earlier work on network protocols and their performance. Section 2.2 surveys existing network parallelization techniques. Section 2.3 discusses previous use of replication in computing systems. Lastly, section 2.4 summarizes how these approaches have addressed the problems from Chapter 1 and how they compare to the approach discussed in this dissertation.

2.1 Network Protocol Designs

2.1.1 OSI Stack Models

Protocol processing involves a large range of operations, including data integrity checks, re-ordering, routing, and flow control. At the lowest level, the networking wire, data is almost always organized into fixed size parcels: packets. The protocols above give the data the appearance of a stream or connection. The commonly accepted model for protocol design is given by the International Organization for Standardization (ISO) in the

Open Systems Interconnection (OSI) model. The OSI model breaks down protocols into 7 categories, depending on their function. Table 2.1.1 enumerates the each layer and briefly describes the function of that layer.

	Data Unit	**Layer**	**Function**
Host Layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Inter-host communication
	Segments	4. Transport	End-to-end connections and reliability, flow control
Media Layers	Packet	3. Network	Path determination and logical addressing
	Frame	2. Data Link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

Table 2.1: OSI 7 Layer Protocol Model

The 7 layer model has largely been displaced by a simpler, 4 layer model (Table 2.1.1); an example of which is the Transmission Control Protocol (TCP) / Internet Protocol (IP) suite. This simpler model better encapsulates the needs of most networked applications and was championed by the the researchers who developed the Internet protocols.

Layer	**Function**	**Example Protocol**
Application	Application-specific functions	SMTP, HTTP
Transport	Application rendezvous, flow control, reliability	TCP
Network	Addressing, routing	IP
Network Access	physical addressing and transmission	Ethernet

Table 2.2: DoD 4 Layer Protocol Model

2.1.2 Network Stack Implementations

The advent of the local area network (LAN) brought an explosion of communication research, including protocol design and performance analysis. The introduction of Ethernet in the late 1970s brought high-speed communication capabilities to low-cost minicomputers. Accordingly, sophisticated network protocols started in software with mainstream systems such as Unix [70].

AT&T released Unix SystemV R3 in 1983 with support for STREAMS [66, 69]. STREAMS introduced a modular approach to I/O systems and popularized the concept of a processing "stack" where several functions were linked together and executed as a sequence. In later versions of SystemV, STREAMS was used to implement complex protocols such as the Internet suite. Berkeley Software Distribution (BSD) 4.2 version of Unix was also released in 1983. It included the first modern incarnation of the Internet protocols as well as the Berkeley Sockets interface, which became the standard user interface for network communication [72].

Jacobson [41] cites BSD4.2 with introducing the standard model of network processing as shown in Figure 2.1. This model was used by most early network implementations and is still followed by mainstream networking systems such as Linux. Data arrives from the network in packets and is transferred to kernel buffers in interrupt context or by direct memory access (DMA). These buffers are processed then converted to task level buffers where they are marshalled into byte streams. The operation happens in reverse went data is transmitted to the network.

About ten years later, the x-kernel [36, 37] aimed at creating an operating system around network protocol processing. From the beginning, communication primitives were available such as `protocols`, `sessions`, and `messages`. The x-kernel was created as a vehicle to explore how protocol features can be separated and composed. Processing occurred on a per-message basis, rather than per protocol. Messages passed through the

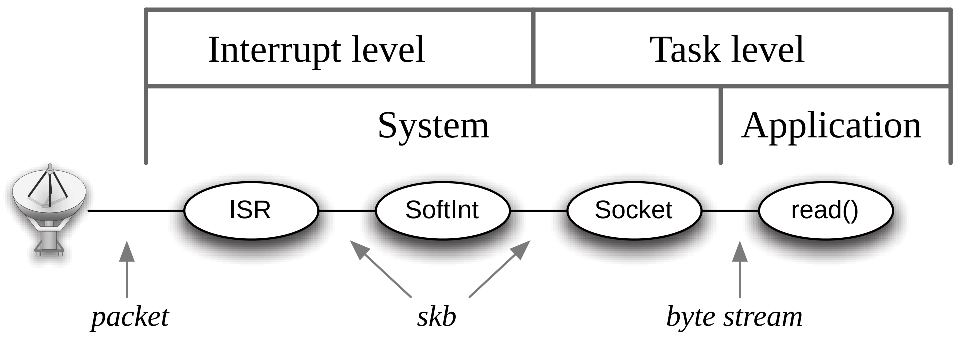


Figure 2.1: Jacobsen's "Standard Model" for Networking from BSD4.2 Implementation (figure taken from talk)

protocol layers using the standard stack operations *push* and *pop*.

The Scout operating system used parts of the x-kernel's networking infrastructure to build a system that was scalable and provided predictable networking performance, even under load [55, 57]. Scout organized all communications around the concept of a `path`. `Paths` contained both the protocol-specific data necessary to process network requests as well as scheduling primitives. This created a single target for resource management and explicitly identified the control path and data path for network connections. It also avoided situations where scheduling policy might interfere with I/O, such as at the user-kernel boundary or between tasks. Building on the x-kernel, Scout provided a comprehensive set of network protocol implementations, including the Internet protocols.

Scout's performance predictability and single processor implementation make it ideal for testing replication-based scalability. Chapter 4 describes how Scout is used to test the behavior of replication on network protocol processing.

2.1.3 Protocol Performance

The performance of the network protocol stack has long been a concern for operating system developers and application programmers. While there are vast number of application specific, custom protocols, a few are general purpose and have received intense study. Chief among these are the suite of Internet protocols, UDP/IP and TCP/IP [60, 63, 64]. The popularity of the Internet has driven a tremendous amount of research on TCP, which is complex. This includes sophisticated congestion control [39, 15, 40, 48, 67], flow control and acknowledgements [49], and performance characterization [47, 20, 30].

Other work has focused on the memory requirements for high-performance network protocol processing. The Berkeley Sockets interface incurs expensive copies and work was done to bypass it [19]. The FBufs project reduces memory bandwidth requirements by remapping pages [24].

Due to the tremendous growth in per-core, or per-thread, performance, computational capacity has kept pace with increasing network speed. This has allowed for increased workloads by either distributing across processors in the stateless model or increasing the number of connections in the stateful model. As discussed in Chapter 1 network protocol implementations must be parallel in order to get the best performance on current and projected multicore architectures. Researchers have tried a variety of approaches to increase throughput and decrease latency. These include software parallelization and specialized networking hardware. The success of both have been limited.

Recent work on scaling TCP/IP network stacks has focused primarily on scaling the number of connections or flows that can be supported rather than on the data rate of individual connections. Corey, for example, uses a private TCP network stack on each core [5, 13, 14], and RouteBricks segregates each IP flow to its own core [23]. These approaches allow systems to scale to handle a large number of connections or flows, but limit the data rate achievable on each flow to well below the throughput of modern network cards.

In some cases, the computational load of the protocol stack is greatly increased by Quality of Service (QoS) processing or packet filtering [54]. Operations such as decryption or string matching greatly increase the per-byte costs of data delivery. This processing is often performed as early as possible (e.g., in the kernel) to avoid memory copies for data that is ultimately discarded.

2.1.4 Protocol Of oading

Another proposed solution for increasing protocol processing speed is to offload the work to dedicated hardware. The most common among these are the commercial TCP Offload Engines [74, 26]. These devices combine a high-speed network device with an application-specific integrated circuit (ASIC). The ASIC performs computationally expensive protocol operations in hardware at high speed. The ASIC capacity is matched to the link speed in order to achieve maximum line rate transmissions with minimal additional memory and power requirements. Similar to a dedicated graphics processing unit (GPU), offloading NICs relieve the computational burden from the CPUs.

A major obstacle for offloading engines in consumer hardware has been the cost. Offloading NICs are roughy 10x more expensive than comparable “dumb” NICs. Unlike GPUs, which have a clear and apparent affect on the user experience, the network improvements have traditionally been marginal for consumer devices and certainly not enough to justify the cost. Consequently, offload NICs are not able to take advantage of the same economies of scale as GPUs are deployed less frequently in datacenters and high-performance computing environments. More importantly, there are major limitations to the offloading features that render them undesirable in these situations.

Because offloading NICs completely bypass the operating system network protocol stack, they are less flexible than software solutions. They do not support complex firewall rules, QoS filtering, and the traffic inspection features on which network servers have come to rely. They do not support link aggregation features, such as 802.11ak. The hardware is dedicated to a single protocol, usually TCP/IP, which limits their use in applications that require custom networking features. Unless there is a substantial decrease in costs, these devices are unlikely to see widespread adoption.

Some high-performance network fabrics also fall into this category [12, 17, 61]. Rather than advertising themselves as offloading engines, they include complex protocol features in the fabric itself. Infiniband [3] includes reliability in two of the four basic transfer modes. In fact, the reliable-connected mode is the most commonly used and most optimized in many vendors NIC firmwares. The end-to-end addressing, routing, reliability, and ordering of Infiniband usually obviate the need for common layer 3 and 4 protocols such as TCP/IP. The biggest tradeoffs are expensive networking hardware, non-standard user APIs, and the need to re-encode traffic to and from other layer 2 networks. This impedes such fabric from use in long-haul networks or for last-mile connectivity. More generally, work has indicated that cost of providing features at a low level is higher the value of implementing them in hardware [71].

A simpler strategy is used by some high-performance NICs. Instead of offloading the full protocol, they perform data segmentation in hardware [31]. Often termed TCP Large Segment Offload (LSO), the operating system generates large read and write requests (up to 64 KB) and provides a template for the packet headers. The hardware then splits the data into packet-sized segments and appends an appropriate header. Protocol processing still happens in software, but is significantly reduced because there are more bytes of payload per packet.

2.2 Protocol Parallelization

Very early in the development of network protocol stacks, work tried to improve performance by introducing parallelization. The difficulties were identified as early as 1989 [33]. Stateless protocols naturally have no inter-packet dependencies and it has been straightforward to adapt them to parallel execution. The same has not been true for stateful protocols, however. Several studies were conducted on the parallelized performance of UDP/IP and TCP/IP.

Bjorkman [11] shows that locking is a significant cost of parallel protocol processing. Stateless protocols, such as UDP/IP have low locking costs and scaled well and were mostly unaffected by these costs. On the other hand, stateful protocols such as TCP/IP incurred significant penalties because of lock contention. The cause of these delays was the complex connection state.

In 1994, Nahum, Yates, Kurose, and Towsley studied TCP and UDP performance on the x-kernel [58]. While much slower, their Silicon Graphics server looked much like today's multi-core architecture. It had eight cores, a shared memory bus, and roughly the same ratio of network bandwidth to CPU cycles. For UDP, scaling was essentially linear and they achieved 1.8x speedup when doubling the number of CPUs. For TCP, they showed that packet-level parallelism offered no performance benefit because the locking costs are prohibitive. Send side TCP speedup was modest 2x at four CPUs and did not increase at larger scales. The receive side attained no speedup. On the other hand, good performance was attained for connection-level parallelism with TCP. With multiple connections, a speedup of 1.8x was again attained when doubling the number of CPUs.

Willmann, Rixner, and Cox completed a similar study in 2006, this time with FreeBSD and faster hardware [81]. They were able to find modest speedup at four CPUs, but the locking and scheduling costs were still prohibitive in the case of packet-level parallelism with TCP. Multiple connections were required to approach the maximum link speed.

Another project, Ensemble TCP, studied the performance benefits of caching and sharing state between TCP connections [27]. By using historical data, the protocol implementation was able to make better congestion control choices and increase network link utilization. Repeated network access were accelerated by up to 28% and access times were, in some cases, reduced by 75%. Unfortunately the scalability implications were limited. The tests involved short-lived connections with relatively expensive set-up and tear-down costs. Large, high bandwidth network transfers tend to enter a steady state and do not benefit from the Ensemble TCP approach.

Rather than speed up a single network connection, many mainstream operating system implementations have focused instead on optimizing their protocol stack for a large number of simultaneous connections. Linux and Solaris [75, 82] have well-studied implementations and considerable effort went into ensuring high-performance for many connections. This has been important for certain applications, such as web servers, which have relatively short-lived connections and needed to support tens of thousands of concurrent clients. However, the limit of per-connection speed to a single CPU throughput is particularly apparent in the architecture of chip multi-threaded (CMT) SPARC servers.

Applications such as GridFTP [2] increased throughput over TCP by opening multiple, simultaneous connections. Each connection was then processed independently by separate processors. Data was split before being sent to the network protocol stack. The data is then reassembled on the receiving end by GridFTP. The resulting transfers are close to the sum of the speed of each individual connection. This technique also has the advantage of allowing transfers across multiple network or multiple links, similar to multi-homing or Ethernet link aggregation [38]. In some cases this approach worked well, but it had significant limitations.

Multiplexing a single network flow over multiple connections undermines the congestion control mechanisms that were developed to provide fair access to network resources. Routing and switching systems rely on these mechanisms to enforce network traffic policies and to avoid starvation [29, 44, 28]. More generally, this approach is not applicable to all conditions. Using multiple connections shifts the reordering and reliability burdens to the application, out of the network protocol stack. As a result, the application needs to maintain state for these functions and the locking, caching, and scheduling issues from parallel protocol processing return. Moreover, this solution is not scalable. Without coordination, multiple TCP connections will compete which each other for network resources. Again, coordination will lead to high locking, caching, and scheduling costs.

The BitTorrent protocol [21] provides increased throughput of a single data stream by communicating with multiple peers at once. If a single peer is made to appear as a group, then it can use multiple connections to transfer data and then do the reassembly at the client. This leads to the same limitations as GridFTP. Furthermore, this breaks existing network resource allocation systems [65]. Multiple connections allow BitTorrent to consume a greater share of network bandwidth than application using a single connection.

The results of this large body of work indicate that another approach is needed to successfully parallelize stateful protocols such as TCP. As described in Chapter 3, replication eliminates the locking costs that prohibit scalability while allowing high throughput on single connections.

2.3 Other Replication Systems

2.3.1 Replication in Operating Systems

This dissertation is not the first use of a replication approach in operating systems. It is similar to the fault tolerance mechanisms used in group RPC system [34, 35, 18]. However those systems do not use this approach to increase throughput. Replication has been used in the Hurricane and K42 operating systems to reducing locking and caching cost of shared data. This technique has allowed key OS objects to be visible across systems with many processors.

In developing the Hurricane OS, Unrau, Krieger, Gamsa, and Stumm used replication to alleviate much of the locking cost on a large SMP system [77]. Hurricane arranged kernel data structures using hierarchical clustering. Data that was read often was replicated across different clusters and data that was written often was shared. The shared data could migrate to a different cluster in order to exploit performance benefits from non-uniform memory architectures (NUMA). The hierarchical structure was used to efficiently propagate changes to replicated data.

Replication allowed Hurricane to bound lock contention and increase total lock bandwidth but it did incur additional checks in the case where a lock or data structure was not on the local cluster. In this case lock acquisition required fetching data from a remote cluster, an RPC operation [43, 42].

Work at IBM and the University of Toronto took replication further with clustered objects throughput the operating system [32, 22]. These clustered objects allowed data to be replicated and partitioned for quicker memory access on NUMA machines. The objects communicated through the Protected Procedure Call mechanism which concisely encapsulated the required locking semantics. The work showed that the clustered objects reduced lock contention which in turn greatly reduced the locking overhead. Pagefault and stat() operations were shown to scale well on eight or sixteen-way SMP machines with nearly flat service times in multiple tests.

2.3.2 Replication in Distributed Systems

The problems associated with shared state or copies of shared state have along been of interest to distributed system research. Distributed OSes, distributed filesystems, databases, and batch scheduling systems have encountered similar issues with locking performance and consistency management. Work such as Ramen [68] advocates using a distributed resource management strategy to achieve high throughput. In this work, a classified advertisements mechanism is designed to take advantage of weak consistency requirements.

Baumann [6] argued that modern architectures look much like a distributed system and should be treated as such. Their project, Barrelfish, advocated the use a multi-kernel design, in which multiple copies of the operating system executed, each on a single processor. These copies then coordinated using an asynchronous messaging system. Replication consistency issues were resolved with single-phase commits over the lightweight URPC protocol. The authors readily admitted that this was an extreme approach and intended to raise questions about heterogeneous architectures. Barrelfish performance was ultimately limited because it did not leverage sharing in architectures that are fundamentally still *shared multi-processors*.

Distributed systems such as ISIS [8, 9] use replication to provide fault tolerance. Some components of the system are replicated and failover gracefully. This allows for a more robust system that can withstand both software and hardware failure while still exploiting parallelism. Ensemble [10] provides a set of communication tools to build distributed applications using these techniques.

2.4 Summary

Exploiting the full capabilities of future high-speed NICs on commodity multi-core processors requires an approach to network protocol stack design that avoids the locking and caching costs that limit current systems. Such an architecture must support both a large number of connections and individual connections with bandwidth and protocol processing requirements beyond what can be handled by a single commodity processor core. Without this support, applications that demand extremely high bandwidths will not be able to fully utilize emerging network interfaces. Replication provides a viable alternative to exists parallelization techniques, but has not yet been explored as strategy for processing network protocols.

Chapter 3

Replication Approach

To address the scalability of network protocol stacks on multi-core systems, this dissertation introduces a replication-based approach to multi-core network protocol stack design. This approach uses replication to avoid the synchronization, scheduling, and caching costs that have limited scaling in network stack parallelization, as described in Chapter 2. Replication allows for an entirely new method of network protocol parallelization by making all common operations lock-free.

This chapter is divided into two parts. The first section describes the approach, benefits, and possible drawbacks. An analysis of possible speedup is included. The second section describes Dominoes, a framework for implementing replicated request processing. Dominoes is a generic replication system and is independent of networking in general. In subsequent chapters, it is used to construct a replicated network protocol implementation.

3.1 Approach Strategy and Bene ts

3.1.1 Overview

The goal of the replication approach is to increase performance by parallelizing a given computation that requires accessing and updating state. This approach gives every processing thread its own copy of the state which can be updated without locks. For this to operate properly, every replica (i.e., thread) must process every update. In the networking context, this requires processing every packet. The cost of this stream of updates is decreased because the global effects, such as data copying, are split between replicas. The total cost of the computation increases because many redundant updates are performed, but the per-replica cost decreases.

Given a stateful connection and a set of processors, every incoming packet is sent to every processor. Each processor maintains a local, exclusive copy of the state. For each packet, some processor is marked the *primary*. The primary processor is tasked with fully processing the packet and delivering the data to the application. Other processors only partially process the packet and update their connection state. Speedup is attained by reducing the cost of most of the incoming data stream. Notice that replication refers to both the state itself and to the operations on the state.

For every outgoing packet, a primary processor is designated. This processor is tasked with delivering data to the network. Other processors update their state and set local timeouts, if necessary. When acknowledgements are received, the incoming packet mechanism ensures that each processor resets these timers correctly or generates additional networking traffic as needed.

Because each processor accesses only its local state, no locking or synchronization is needed on a per-packet basis. For each incoming message, the processor has complete information without communicating with the other replicas. This eliminates many of the parallelization costs that hindered earlier work. The state replicas have the added effect of improving cache performance because there is no write contention for shared memory. If desired, each replica can be on its own cache line. Processors do not write to state information on the other, external replicas. The notion of global vs. local updates provides a powerful model for describing the replicated processing system.

Figure 3.1 illustrates how this approach might distribute a load of four packets across four processors. One packet gets marked for full processing by each processor, marked in red. The other packets, marked in blue, are used to update state, then discarded. In the case where state updates are free, speedup is 4x. In the case of more typical network processing, state updates comprise approximately 10% of the work. Accordingly, the speedup would be 3.08x, as discussed in section 3.1.3.

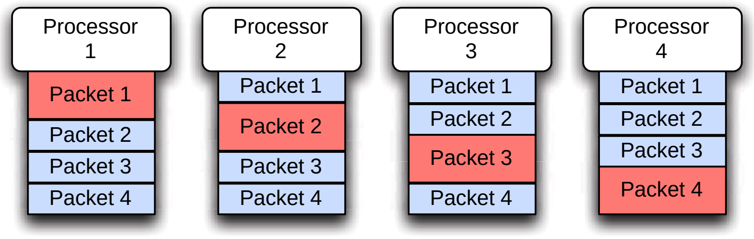


Figure 3.1: Replication Approach to Work Distribution

As an example, consider the reception of four packets by a replicated TCP stack. Each TCP replica would process every request to update its sliding window and other acknowledgement state. However, each replica would copy only the data for its primary requests into the application's receive buffer, and only half of the replicas would enqueue acknowledgements to be sent by the network device using the standard TCP delayed acknowledgement protocol. One replica may also have to start a delayed acknowledgement timer, but that timer can be local to the replica as its firing does not result in a change in connection state; other timers that do impact connection state such as retransmission timers, however, would have to be handled by every replica. Note that this example assumes that the NIC can offload packet data checksumming, a common feature on all modern NICs, as otherwise this expensive processing might have to be replicated.

The primary goal of the replication approach is to increase the throughput of a stream of CPU-bound requests. There are other metrics possible for network performance such as latency or per-byte computation costs. For some applications, those are the characteristic measurements. However, this work focuses on high throughput and the associated computational difficulties.

3.1.2 Parallelizable vs. Sequential Work

The central concept of replicated processing is that there are two types of operations: sequential and parallelizable. Sequential work updates the replicated state and this approach allows it to be processed concurrently on all processors. Parallelizable work is any operation that has side effects, such as delivering memory buffers to the application or expiring timers. For network protocol processing, parallelizable operations tend to be more expensive than sequential ones. Therefore, they are only performed on the primary processor.

For network protocol processing, the following operations are parallelizable:

- Reading/writing data from the application; for most systems this is encapsulated in the Sockets programming interface [72].
- Sending/receiving data from the network interface; examples include dequeueing incoming packets and generating acknowledgements.
- Handling catastrophic events that result in connection failure; examples include reset-generating data and networking link failure.
- Synchronizing replicas
- Timer expiration

These are the principal parallelizable operations that every non-trivial network protocol implementation must perform. Transferring data between both the application and the network is possible using novel queuing techniques that are discussed later. This includes assigning the primaries, which can be done independently, therefore concurrently. Specific protocols may include features that have global effects not in this list, but most fall into one of these categories. For example, a TCP implementation may require that acknowledgements be coalesced before transmitting them to the network. This can be performed after queuing functions, thus concurrently or out-of-band with protocol processing.

A few of these operations are both global in effect and inherently serial. Connection setup and teardown as well as connection failure require state creation and destruction. This requires writing to demultiplexing tables as well as general system configuration access which in turn leads to contention on shared resources. Fortunately, these operations have fixed costs and lie outside the bounds of normal processing. While connection setup costs are expensive, they happen only once and for a high-bandwidth connection they incur a very small fraction of the total computational costs. In this work, parallelization techniques are a performance optimization and the setup/teardown costs are not significant. In fact, the costs are so small that no attempt is made to parallelize them, although many production systems focus on reducing such costs.

3.1.3 Speedup

The replication approach requires every incoming request (i.e packet) be processed by every replica. Speedup is attained by reducing the cost of a portion of these requests – preferably the majority. Partially processing requests is less expensive than fully processing them because the system does just enough work to update state. The minimum cost is attained when full processing is split most evenly between processors. This results in the least amount of parallelizable work for a given processor.

For a given stream of N packets and number of processors P, the fraction of packets marked primary for each processor is $\frac{N}{P}$. Without loss of generality, assume the full processing cost of a packet is one (1). The remaining $\frac{NP-N}{P}$ packets are partially processed. This partial processing is some fraction of the full processing cost. Representing the sequential cost as a fraction of the parallelizable cost, L, the cost of processing the data stream is given by the expression:

$$\frac{N}{P} + L\left(\frac{NP-N}{P}\right) = \frac{N + LNP + LN}{P}$$

From this expression we can calculate the speed with P processors:

$$\texttt{speedup} = \frac{N}{\left(\frac{N+LNP+LN}{P}\right)}$$

and the maximum theoretical speedup:

$$\texttt{speedup}_{max} = \lim_{P\to\infty} \frac{N}{\frac{N+LNP+LN}{P}} = \frac{N}{0 + LN + 0} = \frac{N}{LN} = \frac{1}{L}$$

Therefore the speedup will be dictated by the ratio of parallelizable to sequential work. For typical network protocol processing, sequential costs are near 10%. This gives a value of $L = 0.1$ and a theoretical speedup of 10x. This analysis shows an important trend. Increasingly complex features that do not require additional state updates provide an excellent opportunity for parallelization. Demanding tasks such as encryption can be parallelized effectively. Indeed, any feature that requires more parallelizable work than sequential presents additional possibility for speedup. This result is equivalent with Amdahl's Law where the parallel portion of the program is $\frac{9}{10}$ of the total.

One crucial assumption is implicit in this analysis: that all packets have equal processing costs. This is certainly not true in practice as packets can arrive in various sizes and from various sources. Packets that arrive in bundles tend to be more efficient than ones that arrive separately because of architecture issues such as interrupt coalescing.

3.1.4 Specialized Primaries

The core idea of the replication approach is that the primary replica performs all operations that are globally visible. This centralizes all side-effects and operations that might have to wait, such as delivering data to a busy network device. Conceptually, this is the simplest way of assigning work to the replicas, but more advanced techniques are possible.

Speedup is attained by parallelizing the sequential portion of processing. However, some of the parallelizable operations may be processed concurrently as well. Further performance improvements may be possible by selecting individual operations and assigning them to different primaries. This depends heavily on the specific protocol and types of operations available.

Two common operations are data copying and acknowledgement generation. In-kernel protocol implementations must perform at least one copy because data is processed in kernel memory but is accessed by the application in user memory. Acknowledgements are required for reliable data delivery and use by common protocols such as TCP. These two operations can be performed concurrently. Using the simplest method of work assignment, both of these operations are performed serially on the same replica. Alternatively, the system could assign these operations to different replicas. This allows for parallel processing of different packets and of single packets.

3.1.5 Consistency Management

Connection State Consistency

The replication approach introduces new challenges to network protocol implementation. Because there are many copies of the connection state, concurrency issues can cause these states to diverge. This divergence cannot be allowed to cause erroneous protocol behavior, so it must be managed. If the same operation is performed multiple times but with inconsistent state, the system performance may degrade or worse the connection may fail altogether.

Network protocol designs can be resilient to inconsistency. Networks are inherently unreliable. Data gets lost, routes change, and packets can arrive out of order. Such behavior is not just recoverable, but common [7]. Protocols like TCP are designed to behave well in bad conditions. This allows for a relaxed consistency model, something not typically seen in traditional replication work. It is important to identify what consistency must be maintained before studying how such a relaxed model affects the protocol behavior. Anomalous inconsistency can come from two sources: ordering and timing effects.

Ordering

The order of the requests into the protocol processing system affects how the state changes over time. When first considered, packet ordering seems irrelevant to consistency management. Reliable protocols have mechanisms for reordering data after it arrives at the destination. They must be able to cope with reordering performed by the networking fabric. Some work [27] targets exactly such behavior in the context of multi-path routing. However, interaction with other protocol features causes problems.

Taking TCP as an example, packet ordering affects the flow control and congestion control mechanisms. Consider an implementation with two state replicas that process incoming packets (segments in TCP parlance). If one replica processes segments out of order then it will generate duplicate acknowledgements. The sender interprets this as network congestion and reduces its window size in half [64]. On the sending side, replicas responsible for sending different packets and acknowledgements can enqueue packets to the device out of order. Receiving packets or acknowledgements out of order unnecessarily can have significant impact on protocol performance.

If separate queues are used for incoming and outgoing requests, processing requests in different orders can result in catastrophic protocol failure. Figure 3.2 shows a case where processing separate queues in different orders can cause inconsistency, in this case one replica processing an acknowledgement for an outgoing packet that it has not yet processed. Three packets are queued for transmission and two replicas are available to service them. `Replica 2` is marked primary for `Packet 1` and `Replica 1` is primary for `Packet 2` and `Packet 3`. `Packet 2` and `Packet 3` are processed normally, but `Replica 2` stalls and does not run for a few milliseconds. Meanwhile, `Packet 2` and `Packet 3` arrive at their destination and an acknowledgement for them arrives from the network. This acknowledgement is delivered to both replicas. `Replica 1` processes the acknowledgement normally. When `Replica 2` is rescheduled, it must process `Packet 1` first. If it processes the incoming data in a timely fashion, then it will register an acknowledgement for data that has not been marked as sent. Standard behavior in such a case is to send a reset and close the connection.

Timing Effects

Figure 3.2 raises another issue of consistency, timing. The ordering dilemma is magnified because one processor (or thread) stalls and is de-scheduled. This can also delay the expiration of timers or the servicing of packets *before* timers expire. Expiring a timer in one replica without expiring it in another may result in unnecessary retransmissions, spurious duplicate acknowledgements, or connection failure.

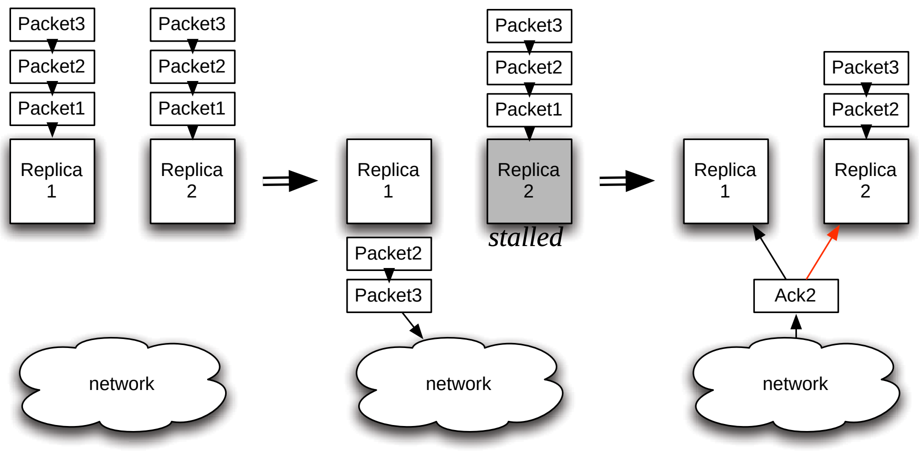


Figure 3.2: Scheduler-generated Inconsistency; replica 2 could end up processing an acknowledgement for a packet that it does not know has been sent.

3.2 Dominoes Framework

3.2.1 Overview

This section describes the architecture of Dominoes, a software framework for building replicated system services. In the simplest terms, it is a high-performance request processing system. Included are the philosophy behind the design, the several parts of the system, and how these parts interact. It frees the programmer from the details of lock-free queuing/dequeuing as well as the threading and scheduling intricacies to support them. The design is intended to allow the construction of virtually any system service, not just the networking examples given in this dissertation. Many traditionally tedious features are already built so that the user may quickly get to the task of defining how replication will function for a given service and proceed to implement it.

Dominoes consists of three key mechanisms :

1. A high-performance, multi-producer, multi-consumer, lock-free queue,

2. **channels**, which abstract queues and control access, and
3. **domains**, which abstract threads and process channels.

Domains encapsulate both concepts of replication: the thread which performs replicated work and state that is replicated across processors. Channels provide the mechanisms for feeding data in and out of domains quickly and in a lock-free manner. Domains process channel data through a subscription mechanism. The combination of channels and domains provides flexibility in the framework. Because domains are very general processing primitives, they can be adapted easily. Likewise, channels allow domains to be connected in various ways to provide not only high throughput, but processing control and synchronization.

Figure 3.3 demonstrates how these components can be used to build a hypothetical processing system. In this example there are four domains, each mapped to one of the available four processors. Two channels are created, one for incoming requests and one for outgoing messages. Each domain subscribes to the incoming channel and can publish data to the outgoing channel. The domain takes care of processing requests and performing sequential and parallelizable operations. The channels ensure that the correct data is delivered to the replicas and maintains reference counts on the data. When the data becomes unused, the channel frees it.

A third channel is presented in Figure 3.3 to manage consistency between the replicas (e.g., domains). The Consistency Manager controls the interface to the channels as well as resolving inconsistency, when it occurs. These operations happen out-of-band with normal request processing and the computational overhead is low. This component is the most specialized because it depends entirely on the behavior of the specific protocol implemented.

The following sections discuss how each part of Dominoes is constructed and how they can be used together to build a replication system.

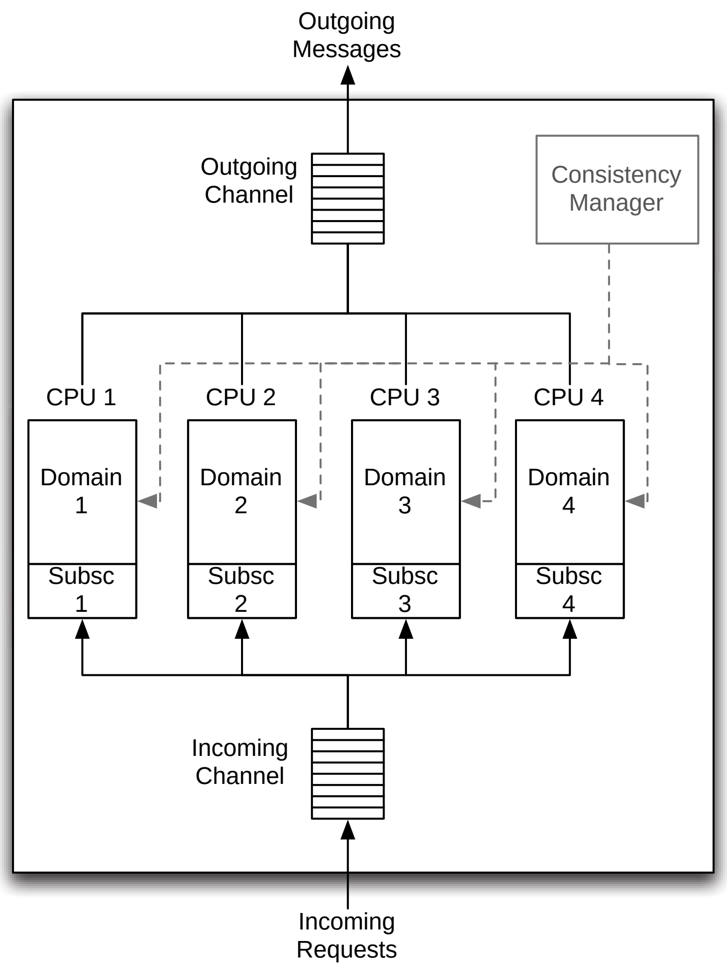


Figure 3.3: Example of Dominoes Architecture with Four CPUs

3.2.2 Lock-Free Request Queue

At the heart of any request processing system is a robust, high-performance queue. Dominoes requires that these queues support multiple producers and multiple consumers. In addition, the queues must be lock-free [51] to provide good performance with replication.

The default Dominoes queue is a fixed size FIFO, implemented as a ringbuffer. The ringbuffer is split into cells corresponding to the size of the requests that will pass through. In turn each cell contains a reference count to indicate when it becomes active and the

```
int RingbufferRemove(ringbuffer_t *rb, void **item)
{
        ringbuffer_cell_t   **thr_head, *curr;

        thr_head = pthread_getspecific (rb->key);

        if ((*thr_head) == rb->tail) return -1;

        /* Dequeue the nesxt item we want and advance the per-thread head. */
        *item = (*thr_head)->value;
        curr = *thr_head;
        *thr_head = *thr_head + 1;
        if (*thr_head >= rb->end) *thr_head = rb->buf;

        /* Update the cell refcount and maybe global head */
        if (atomic_dec_atomic((atomic_t *)(&curr->refc)) == 0)
                rb->head = *thr_head;

        return 0;
}
```

Figure 3.4: Source Code for Dequeuing from a Replicated Ring Buffer

remaining number of consumers that need to process it. The queue heads and tail are indicated as pointers to cells in the ringbuffer. A single tail is used for insertion, but there is a distinct head for each consumer. Consequently, the number of consumers must remain fixed after the ringbuffer is created. Figure 3.4 illustrates the behavior of the dequeue operation.

Concurrent access to the head and tail pointers is implemented using the compare-and-swap operation (CAS), which is guaranteed to be atomic by the hardware. The CAS operation updates memory locations by taking a crucial third argument, the old value. The update is completed only if the current value matches the old value. This obviates the need for costly software locks. Failure of this operation results in neither a corrupt queue state nor context switch. Aggressively retrying[1] failed CASs results in high throughput with zero chance of queue corruption. Channel throughput performance is presented in Chapter 5.

The novelty of this lock-free queue is the key component to enabling the replication approach because it allows for simultaneous, high-speed access to the incoming data stream. Traditional synchronization primitives, such as mutexes and semaphores, would severely limit the queue throughput. Such an approach would effectively move the locking bottleneck from the connection state, as seen in earlier work, to the queue interface. This system is able to do away with locks altogether.

3.2.3 Channels

To separate the logistics of controlling request streams from the queuing implementation, a further abstraction is introduced, the channel. Dominoes channels allow the user to create and manipulate streams of requests without needing to know the details of the underlying queuing mechanism. Depending on the how they are constructed, a channel can be ordered, unordered, FIFO, LIFO, or a user-defined stream type.

Most importantly, channels control who can *publish* (enqueue) or *subscribe* (dequeue) from a given queue. This allows concurrency without having to manually control the number of heads for a multi-consumer queue or manage reference counts. Each channel has a `Publish()` function to add new requests to the queue. Removing these requests is handled through Subscriptions, which are described in section 3.2.4.

The channel data-structure is abstract and must be subclassed to be used. Combined with the lock-free queue, a RingChannel object is provided as a ready-made channel. RingChannels are the fundamental data structure of the Dominoes framework and are controlled by an API of six principal functions.

[1]In this case the application only waits a few processor cycles.

- `int RingChannelInit()`
- `int RingChannelDestroy()`
- `int RingChannelEnqueue()`
- `int RingChannelDequeue()`

  Functions to insert/remove items from Channels. RingChannel queue elements are are a generic `Request` type.

- `int ChannelSubscriptionActivate()`

  Enable a single path of delivery for a channel.

- `int ChannelSubscsriptionDeactivate()`

  Disable a single path of delivery for a channel.

The data in channels is carried by a special Dominoes `Request` type. In addition to a payload pointer, reference count, and deallocator, each request designates one domain as a *primary*. The primary domain for a request is tasked with completing parallelizable work in addition to sequential work. It must *fully* process the request as described in Chapter 3.

3.2.4 Domains

To process channels, scheduling and context information is collected into domains. Domains also contain local state, which is replicated in each instance. Thus domains are the primary tool for implementing replicated services. Users can specify what data is local, how it gets updated, and how the processing is scheduled. In other words, request processing occurs *in* domains. Everything else exists to feed data in/out.

Typically, a domain is analogous to a thread and is mapped one-to-one with system threads. This makes the domain implementation somewhat uninteresting because threading models are well developed and understood [59, 1, 36]. However performance requirements often lead to several domains within a single OS thread. Consider a Dominoes implementation of the `select()` system call which searches through a list of Subscriptions for one with live data. A context switch for each domains is expensive, but with them all in a single thread the cost is simply a function call.

Domains consist of the following fields and methods:

- **Thread context**

  The thread context allows the domain to interact with the system scheduler. For a kernel space service, this will correspond to a kernel thread context. In the case of a user space service, a wrapper is provided for the common Pthreads library. Contexts are sometimes shared between domains. The context object must provide a scheduling routine when multiple domains share the context. Context sharing is appropriate for low priority domains or a set of domains that are mutually exclusive, such as threads competing for access to a hardware device.

- **Replicated state**

  The state object is a generic pointer to a protocol (or implementation) specific object that will be replicated across all domains. For common network protocols, as TCP, this is the connection state. Practical coding necessity makes the domain the simplest and fastest location for storing this information because it allows domains to access their state quickly, without list manipulation or hash table lookups.

- **Subscription list**

  A list of active subscriptions is needed to process incoming requests.

- **Domain ID**

The domain Identifier is a user provided field. It is not used within Dominoes itself, however request processing code may use it to identify which domain(s) is running. This can be used to implement domain specific state updates or special consistency management policies.

- **Scheduler callback**

  The scheduler callback is a user supplied function that determines how active subscriptions are serviced. A simple scheduler may only call `subscription->next` to implement round-robin access to the subscribed channels. More complicated polices are possible such as a priority scheduling.

  This callback does not affect scheduling of the thread contexts, nor does it affect how domains are scheduled within a single context. Thread context scheduling depends heavily on the service type and implementation. For this reason system scheduling is isolated in the context object and does not interact directly with domains.

- **Subscribe/Unsubscribe callbacks**

  Special callbacks are provided to add and remove items from the subscription list.

- **Run callback**

  The `Run()` function is called whenever the domain's context is scheduled. It processes a single request and returns. Servicing all the requests on a channel, or servicing multiple channels, requires multiple calls to `Run()`. The pseudocode in Algorithm 1 describes the behavior.

  An important distinction of the `Run()` implementation is that it greedily assigns a primary if one is not already set before the request is processed. The first domain to attempt processing will automatically become the primary. This behavior is easily overridden by manually specifying the primary before publishing the request to the channel.

Algorithm 1 Behavior of domain $Run()$

```
$subscription \leftarrow$ Scheduler()
$request \leftarrow subscription.Dequeue()$
**if** $request \neq \emptyset$ **then**
    CAS($req.primary$, $\emptyset$, domain)
    $subscription.Callback(request)$
**end if**
ReleaseReference($request$)
```

Subscriptions

A system built with Dominoes will often have many domains and many channels simultaneously. The mapping of which domains read from which channels requires careful management to ensure that replicas all receive the same stream of data. This many-to-many mapping is managed by a Subscription structure. Subscriptions provide a simple way of querying which channels a domain is watching as well as which domains consume a given channel's data.

Subscription and un-subscription is synchronous. No data is delivered while updates occur. This is one of very few system-wide, serial operations. It guarantees that sequences of subscribe $\rightarrow$ publish $\rightarrow$ unsubscribe operations occur in the desired order. For network protocol processing, it turns out that the serial processing penalty of this operation is negligible. The subscribe/unsubscribe cost is subsumed into the connection setup/teardown operations, which are already serial. Once a connection has commenced processing, Subscriptions do not need to be altered. The costs for other types of services, which may require this, are not studied here.

3.2.5 Timer Channels

Network timeouts are beyond the scope of normal data-driven events. While it is not always necessary for timeouts to preempt an ongoing process, they should be handled out of band with regular traffic. Certainly, a long series of send requests should not be completed if the first send times out.

Timer management is abstracted by a TimerChannel. TimerChannels use timer wheels [78] to provide timer start and cancel in $O(1)$ time. TimerChannels allow timers to be used in one of two modes: each domain can have its own channel to which it subscribes and services timeouts. Otherwise a third party can subscribe to the TimerChannel and publish timeout requests to the domain's main channel when they occur. Instead of the timeouts being processed as normal requests with the push/pop interface, the request handler calls a special timeout handler that operates similarly to the signal handler of a traditional timeout.

3.2.6 Dif culties

The combination of channels and domains provides flexibility in the framework. Domains can be chained together with channels as the glue and complex interactions can be captured. This complexity can quickly become a determent if it is not carefully managed. Choosing when to use channels and when to delivery data directly has important performance implications. The publish/subscribe model includes an implicit handoff of memory buffers. Research on STREAMS has shown these handoffs to be expensive if done often or needlessly.

Because channels are built with finite hardware limitations, they are bounded in length. When they provide data for domains, it introduces an explicit load balancing problem. In the case where there are many domains processing a network stream, all requests processed by the most advanced domain, but not by the trailing domain, must be buffered. Therefore the memory requirements are directly proportional to this progressing gap. For a real implementation there must be both bounds for the gap and a way to enforce it. In other words, the implementation must ensure that one replica does not get "too far behind". If one replica falls behind then it cannot process its share of the requests in a timely manner. This requires either extra work to help the lagging replica catch up or the other replicas must wait. Either way, careful planning is required to ensure that the replication approach actually increases throughput.

3.3 Summary

This chapter introduced the replication approach to parallel network protocol processing. The approach allows for concurrent processing of stateful protocols by replicating the state across many processors. Thorough study of the literature has revealed no work on applying such an approach to networking or to high-throughput system services in general, beyond that described in Chapter 2.3.

The replicas allow for local access to state information without expensive synchronization mechanisms such as locks or transactional updates on a per-packet basis. Speedup is attained by distributing the parallelizable work between processors and reducing the per processor cost for the data stream. This introduces more work by processing the same stream many times, but accelerates throughput because each replica executes more quickly. The maximum theoretical speedup is given by the ratio of parallelizable to sequential work. For example, a ratio of 10:1 can expect a speedup of 10x under the best circumstances.

Consistency management is a significant challenge in adapting this approach because of the difficulties in maintaining a globally consistent model of time and providing reasonable ordering semantics for data delivery. A key insight is that networking protocols are naturally tolerant of unreliable systems. This is advantageous when implementing the approach. Chapter 4 describes the architecture and implementation that was developed to explore these problems and possible solutions.

This chapter also describes Dominoes. Dominoes provides a framework for implementing replicated request processing. It focuses primarily on system services, but the framework is general enough to allow for many types of systems. It significantly reduces the cost of adapting network protocols to the replication approach by centralizing the queue management and scheduling routines. The channel and domain abstractions are lock-free and provide intuitive interfaces for building replication systems.

Chapter 4

Dominoes Network Stack Implementation

The approach described in Chapter 3 has never been addressed by any existing network protocol implementations. Former parallel protocol stacks have locks as a core tenet and synchronization primitives permeate their design. Additionally, general-purpose replication systems are not available for building operating system services, such as networking. In order to test this approach, a new architecture must be developed.

This chapter discusses adapting the Scout networking system to the Dominoes framework. The combination of these parts provides a comprehensive system for testing the replication approach and for evaluating the suitability of various networking features to the approach. As much as possible, an effort is made to separate the replication mechanisms from the protocol details. Some concessions are necessary and are described when they appear.

4.1 Overview

The Dominoes framework does not contain any network specific functionality. This section focuses on using Dominoes to construct a replicated network protocol processing system. Rather than reinvent the wheel, this work takes existing protocol implementations and adapts them to Dominoes.

Choosing the protocols to implement is straightforward. The Internet protocol suite (IP, TCP, UDP) is well established as the de facto standard for providing addressing, routing, reliability, and ordering on high-speed networks. IP runs on virtually every type of device and every type of network: wide area or local, wired or wireless, terrestrial or satellite. Chapter 2 illustrates the massive amount of previous work in understanding the behavior of IP as well as the performance limitations. Thirdly, widespread deployment of IP means that performance improvements will have the greatest possible impact on real applications. At the end of the day, application performance is driving this work.

There are many IP implementations extant and choosing one is difficult. Performance is critical, so the work must start with an implementation that already provides good single core performance. Because Dominoes is a lock-free system, any locking primitives need to be removed. Many IP implementations, such as Linux [80], Solaris [50], and BSD, have complex synchronization mechanisms because they are built to run many simultaneous connections and on multiprocessors. This is compounded by the fact that such systems support a wide variety of hardware and have complex interfaces to their device drivers.

This work uses the Scout [55] networking system to perform protocol processing. The Scout networking stack provides a number of advantages for this purpose, including:

- A simple, modular, open-source design and implementation that eases integration with Dominoes
- A focus on uniprocessors, eschewing the complicated locking mechanisms of high-

performance network stacks

- Performance tuning and optimizations not available in other simple open-source stacks such as lwIP [25]
- Early demultiplexing, allowing Dominoes to easily determine the set of replicas to which each packet should be assigned

Rather than implement the replicated networking system as a stand-alone operating system, Scout is adapted to run as a library. This allows for easy testing by linking with userspace applications. It also leverages existing OSes, such as Linux, for scheduling, I/O, and device drivers. For performance reasons, it is sometimes desirable to have protocol processing occur inside the kernel in order to avoid a needless data copy. This is also possible because Scout can run as separate system, independent of third-party libraries. This flexibility allows the system to be run in a variety of environments. An example testing environment is described in Chapter 5.

4.2 Scout

4.2.1 Overview

The principal abstractions in Scout are routers, stages, and paths. Routers are protocol classes that are arranged in a graph, with edges between routers indicating connections between protocols. Stages are instances of routers generally associated with a single network connection, while paths are sequences of stages along with queues containing requests to be processed by those stages. Paths constitute the primary entity for scheduling and state encapsulation, as processing of a request along a path is essentially atomic - the Scout scheduler removes a request for one path queue, runs that request to completion through the path, and then chooses a new path to schedule.

Each protocol, such as Ethernet, IP, or ARP, is implemented as a router with a standard interface. Scout draws much of its protocol code from the x-kernel and data is transferred between these routers by stack-like `push()` and `pop()` functions. To add a new protocol, a programmer simply builds a new router and attaches it to some point in the router graph. Figure 4.1 shows the router graph for a simple suite of protocols, including IP. An application that wants to use UDP/IP creates a path (shown in bold) which traverses the graph. Each node in the path then gets a specific instance for that application – a stage.

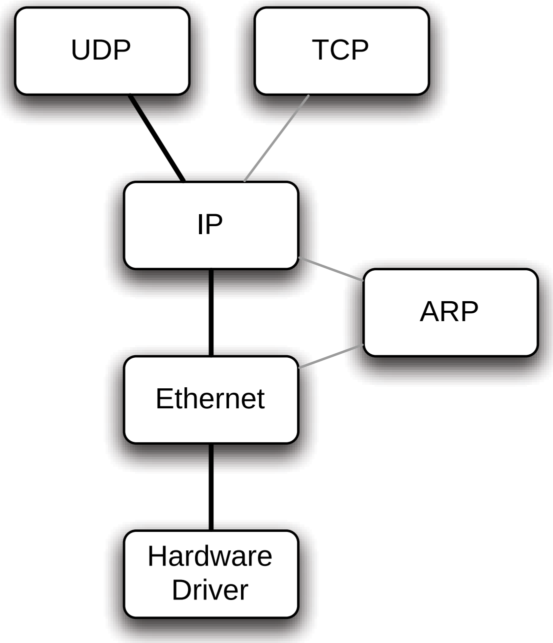


Figure 4.1: Scout Router Graph of IP protocols with UDP/IP Path in Bold

When the application sends and receives data on this path, it goes only through those stages. The path itself is better illustrated with Figure 4.2. Once the path is created, it behaves very much like a queue. Data is enqueued, processing happens internally, then data arrives at the end to be dequeued. Some protocols are bi-directional, which requires paths have two entry and two exit points, one at either end. When paths are created, they require a participant list to know what kind of stages to create. Items in the participant list include IP addresses, UDP ports, etc. The participant helps determine if data is appropriate for a given channel or whether is should be discarded (i.e., dropped).

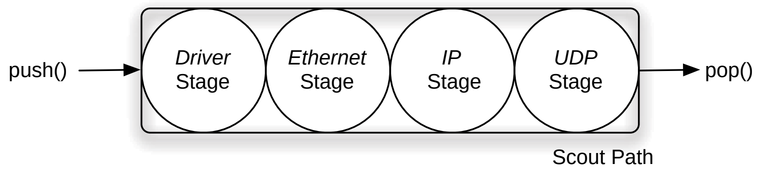


Figure 4.2: Example Scout Path with a Common Protocol Stack

4.2.2 Network Routers

Scout provides routers for a variety of common network protocol such as the Address Resolution Protocol (ARP), Ethernet, Internet Protocol (IP), etc. This work chooses a subset of these to implement with Dominoes. Implementing the entire set of routers would be both time consuming and wasteful because not all the services would benefit from replication. The following protocols are supported in the Dominoes implementation.

- Device driver, Ethernet, IP, ARP, UDP, and TCP

The device driver router is needed to communicate with the network. Without the ability to send and receive traffic, the protocol system has no data to process. Scout supports a limited range of networking hardware, dating from its origins in the mid 1990s. None of these include modern, high-speed NICs. The driver router has been replaced by a generic system that can hook into the native operating system's I/O infrastructure. In userspace, it supports standard APIs such as *libpcap* [46] or Infiniband Verbs. Inside the kernel, it can be attached to a native device driver. The driver router operates independently of any on-the-wire data format. A separate router is provided to parse and generate Ethernet headers. The Ethernet router carries over from Scout, unmodified.

The IP router is far more complicated than the previous two. IP provides addressing, routing, and fragmentation. The replicated version of IP requires several major changes from the original scout code, as described in the next section. Only IP version 4 is ported over. The IPv6 Scout functionality is unsupported. Because the IP implementation will be carried over Ethernet, the ARP [62] protocol is also required.

ARP translates IP addresses, which are global, to Ethernet MAC addresses, which are only routed to the local network. Internally, ARP consists of a hash table that maps IP address to Ethernet addresses. Implementing ARP in Dominoes is more complicated than Ethernet, because it is stateful, but still very simple because the ARP table is updated very infrequently. By its nature, ARP is sensitive to latency, but not throughput. Replication is not performed for ARP processing.

IP provides host-level addressing. To deliver data to application, two higher level protocols are used: TCP and UDP. These protocols provide application-level (port) addressing. UDP is stateless protocol. It is a good study of maximum throughput for the replication approach because the sequential work is nearly zero. As shown in Chapter 2, UDP achieves good speedup with traditional packet-level parallelization techniques. To be successful, replication will need to duplicate these results.

TCP provides much more sophisticated protocol features such as data ordering, reliability, flow control, and congestion control. The TCP connection state provided the fundamental bottleneck for traditional lock-based parallelization methods. With the replication approach, the TCP router is adapted to use a per-domain state rather than per-router one. This state is stored in the domain's user field. The TCP code is adapted to use replicated processing by performing domain lookups rather than simple state access.

4.2.3 Support Libraries

To support the network routers, a large portion of the Scout infrastructure is required. Most functions are available inside Scout as libraries. The significant libraries are:

- Type library

  Scout contains a number of specialized types such as the AnyType object. Rather than trying to duplicate these types using system libraries, the Scout type objects are included.

- Attributes

  Attributes are name/value pairs that communicate data, usually control data, along a path. The name is an integer index and the value is an AnyType object. Attributes are used during path creation and destruction.

- Heap

  The heap library is Scout's built-in dynamic memory allocator. The interface is preserved, but its implementation has been replaced with the system's allocator. In this case the underlying functions are replaced with Dominoes' specific `ContextMalloc()` and `ContextFree()`.

- Checksum

  Several checksumming routines are provide to perform the protocol specific checksum operations. This code is sufficiently generic to be used unmodified. It is thread safe and recurrent.

- Msg / buffer

  The Scout message library provides an efficient system for manipulating and delivering messages. Messages are constructed as a list of buffers that allows for efficient fragmentation/reassembly and for quickly altering headers. Here, Scout draws heavily from the x-kernel to provide a flexible message object. Similar objects are used extensively for high-speed networking protocol implementations such as Linux, Solaris, and BSD. The message library provides one crucial feature that enables replication. Duplicate messages can be created through the use of a `msgInitWithMsg()` function that creates another set of control structures, but uses a shared buffer. This allows domains to quickly create a copy of the message to process locally.

- Buffer, lifo

  The message objects is augmented with a Dominoes-specific control field. This allows the message to be embedded inside a Dominoes `request`. In turn, this allows Scout code to access domain information such as the primary. Along with the message object, there is a special buffer library that allows fast message allocation from a pool of pre-constructed buffers. The buffer management code requires a lifo queuing mechanism, which is also provided. The internal reference counting code is not thread safe and has been reengineered to use CAS operations.

- Map

  The map library is a highly-tuned hash table system. Maps are used for demultiplexing incoming packets, so they are performance critical. Analysis of the Scout map performance and behavior is available [56]. The map code functions essentially unchanged.

- Semaphore

  The Scout semaphore object is needed for protocols that have to “wait”, such as TCP. Scout semaphores operate slightly different than POSIX semaphores, but the native Scout objects do not function without Scout scheduling. Internally Scout semaphores are replaced with system native version, although the API remains intact.

- Tracing, debugging, misc

  A few other, small library functions are needed in order to use Scout router objects. This includes the Scout network interface API, the option handling mechanism, and the debugging macros. The tracing code is also provided because most of the Scout code is already instrumented. A single trace source file gives access to a wealth of data, easily.

All of these functions are collected together to form a library called `scoutbase`. The `scoutbase` library encompasses a large range of functionality that is stand-alone and can operate inside a Dominoes system with little or no modification. On its own, `scoutbase` is a large framework, but the development effort was proportionally low because the code had already been written and tested in Scout.

4.3 Embedding Scout

Building a replicated network stack with Scout is best described as embedding Dominoes into a Scout path. Adapting Scout to use Dominoes is a significant engineering challenge, requiring some major changes. Unmodified Scout paths contain protocol data as well as the thread information needed to schedule the path. In this case, Dominoes can do the scheduling itself. In Scout, the path is scheduled explicitly. Dominoes moves the scheduling functionality to the domain. Dominoes channels replace the queues on both ends of the path. Additionally, the `pathCreate()` function is modified to create Stage objects for each domain.

Figure 4.3 shows the result of this embedding. Scout's uniprocessor ringbuffer path queues are replaced with Dominoes' replicated channels and the sequence of stages associated with Scout paths are replicated into multiple domains. This structuring preserves the same protocol implementation API on Scout paths while supporting replication internally through the use of Dominoes channels and domains.

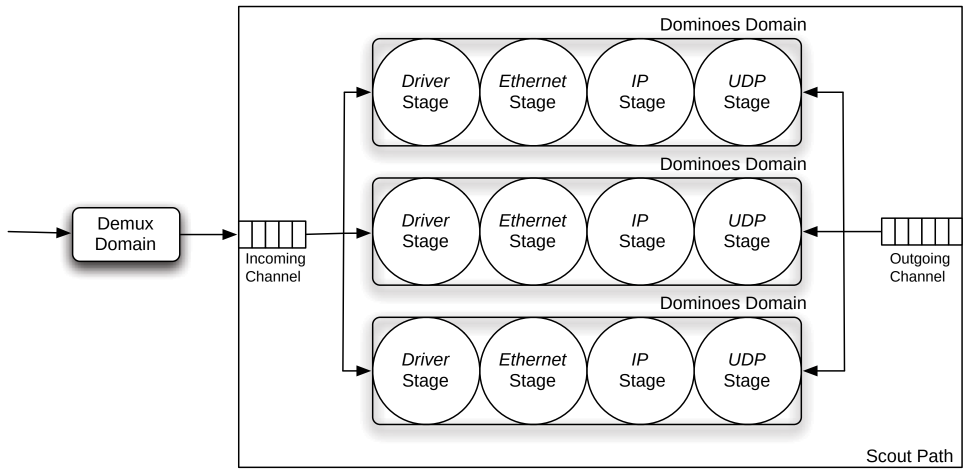


Figure 4.3: Scout Path Similar to Figure 4.2, with Channels and Multiple Domains

Each replica on a Scout path processes the same set of requests (from the shared channel) while preserving scheduling and state independence (within a Dominoes domain). Demultiplexing, which in Scout is performed by the low-level device driver in interrupt context, is done in a separate Dominoes domain that queues demultiplexed packets to the appropriate path's channel.

A single domain handles all incoming packets on a given interface. The performance is sufficient because the domain only performs a demux operation, then publishes the packet. For outgoing packets, two methods can be used. A second domain, mirroring the demux domain, can be used to write data on the outgoing channel. If the data being sent is small, such as acknowledgements, the path domains can deliver the data themselves via the driver stage.

An additional channel is present, but not shown in Figure 4.3. The context `Run()` function, which schedules domains, is supplied by the user and is configurable. Therefore, external functions are not allowed to directly alter that scheduling queue. In order to add or remove domains from a given context, a special control channel is provided. The contexts periodically check this channel for changes to their scheduling policy. For example, at startup an application can create a set of contexts to use for processing. When domains are created they can automatically schedule themselves by publishing a request to the desired context's control channel.

4.3.1 Semaphores

Semaphores present an interesting problem for this architecture. The current Dominoes implementation does not allow for pre-emption of threads. More importantly, the semaphore objects are often local within a stage. When a given domain waits on a semaphore, the same domain needs to process the request that wakes it up. However a sleeping thread cannot process any requests. An example of this behavior occurs during the TCP three-way handshake.

When a TCP server receives a connection request (SYN), it generates a response (SYN+ACK), then waits for a response (ACK). While waiting for the response, it monitors a semaphore, indicating the connection setup is complete, before it transitions to the *established* state. In Scout, the incoming response generates an interrupt which will force processing of the new data and wake up the sleeping thread. In the Dominoes implementation, the interrupt only publishes a request to the appropriate channel. The domain that needs to process the packet is sleeping and the system has no way of pre-empting it.

Several possible solutions to this problem exist. Dominoes could explicitly pre-empt the running domain with a request that it knows to be of higher priority, although this requires that all requests be prioritized. Alternatively, domains could designate a peer to process their requests while sleeping. This would require inter-domain state visibility. Both solutions require storing the call stack while a request is in flight. A third solution is used here.

A special semaphore wait function is provided for domains. This function trampolines back into the domain scheduler to check if new packets or new channel management requests have arrived. This function leverages the fact that the domain scheduler processes a single request per call. It is vital to process requests only up to releasing the semaphore, then let the call stack return to normal. The original scheduler may have work to do before processing subsequent requests. If no requests are present in the wait function, it checks the semaphore and loops. Figure 4.4 describes the behavior of semaphore waiting.

```
void semaphoreWait_dominoes(Semaphore s, context_t *c)
{
  int rc = sem_trywait(&s->sem);
  while(rc != 0){
    if (c){
      ContextRunCtrl(c);
      ContextRunSingleDomain(c);
    }
    rc = sem_trywait(&s->sem);
  }
}
```

Figure 4.4: Dominoes Semaphore Wait Implementation

This code imitates the pre-emptive behavior of Scout, but requires no changes in Dominoes. The disadvantage of this approach is that resources must be released in the same order that they are requested. There is no method to resume the original function, immediately, when the first semaphore is signaled. In the case where a sending domain must wait for buffer data to be available, Dominoes provides the ability to temporarily deactivate a Subscription. This allows the domain to process other requests, such as acknowledgements, while waiting for the semaphore. The distinction is important. The semaphore wait allows other contexts to be scheduled, not just domains. Thus different paths can be executed on the same processor. For example, IP may have to wait while an ARP request resolves.

4.3.2 Timers

Two modes are possible using Dominoes TimerChannels. Domains can have their own TimerChannels, which they setup and handle as timeouts occur. Alternatively, a single TimerChannel can be configured to publish timer expiration to specified channel. This allows multiple domains to see the same order of events, such as send requests and timeouts. Two of the protocols in Scout need timeouts to function properly, ARP and TCP.

The ARP protocol is not sensitive to a performance because the number of requests it must service is very low. The typical TCP connection needs to make only a single ARP lookup because the resulting mapping (IP address to Ethernet address) is valid for several minutes. Either method of TimerChannel handling is sufficient for ARP, but this implementation restricts ARP to a single processing domain. Therefore the simplest solution is to have that domain manage its own channel and handle the timeouts directly.

TCP is more complicated. The receive side is event driven and not dependent on timeouts for flow control or to maintain consistency between replicas. However, the send dynamics to handle lost data. A send timer indicates fires when an appropriate acknowledgement has not arrived for sent data. This happens fairly infrequently on high throughput connections because subsequent packets will trigger a duplicate acknowledgement which also notifies the sender of lost data. A persist timer is set when a receiver advertises an empty receive window. This avoids deadlock in the case that an updated, open receive window notification is lost.

4.4 Parallelizable Work Assignment

Using the replication approach, only the primary replica executes parallelizable operations. The system must differentiate between parallelizable and sequential operations, a distinction that does not exist in the original Scout implementation. This requires evaluation of each protocol.

IP uses local state only when performing fragmentation/de-fragmentation. In Scout, fragmentation is considered the slow path through IP. It occurs as a result of errant network behavior such as failure to detect the path MTU. The Dominoes implementation does not support replicated processing of fragmented IP packets. Without fragmentation, IP is stateless. All work is parallelizable and can be executed concurrently. Likewise UDP is stateless and has no sequential work.

TCP has two types of operations with global effects: delivering data to the application and delivering data to the network. Delivery to the application requires marshalling data from receive buffers and copying the data to application buffers. Delivery to network includes sending data, re-sending data, and sending acknowledgements. Both types of operations benefit from concurrent execution and are designated parallelizable. All other operations (explicit state updates, timer expiration events, etc.) affect only the local replica's state.

4.5 External Interface

A replicated Scout path has two important routers that serve as end interfaces. An application wrapper (Appwrap) router provides hooks for application to receive and transmits data to the path. A driver router behaves like a generic network interface for communicating with the network.

The Appwrap router provides a `push()` and a `pop()` function, like all Scout routers. Pushed data is delivered to path. The `pop()` function is empty. It is a hook for applications to attach their own receive functions. When linking with the Dominoes Scout library, a programmer must provide a function named `appwrap_pop()`. This function is automatically configured as the delivery function when a replicated path is created. Using this interface, the path behaves like a queue with replication happening invisibly.

The driver router operates in a similar fashion, but at the other end of the path. It provides `read()` and `write()` functions to deliver data to the network. This is a departure from other Scout device drivers, which are interrupt driven. The `read()` function performs a demultiplex to select the correct path for delivery. It then publishes the incoming data on the path's receive channel. Because domains are explicitly scheduled and cooperatively multithreaded, the driver system is event driven. These events may be interrupts if the driver is connected directly to a device driver's TX and RX functions, but they are not required to be. The driver can be connected to a communication library such as `libpcap` to perform data transfer.

Some protocols, such as TCP, guarantee ordered data delivery. This can be difficult when many domains are pushing messages to the application, simultaneously. They maintain this ordering a thin layer is included in the data delivery functions. This Ordering Manager controls how data is copied into application buffers. The Ordering manager works by allocating a fixed size buffer for the application, then dividing that buffer into cells. Each of these cells is marked by a sentinel array which indicates how much data is available in each cell. This is shown in Figure 4.5. It allows the application to read the incoming data, in order, while several domains are writing it. As long as the data buffer is larger than the available data, such as the TCP window, separate domains cannot overwrite each others data. When data is read, the sentinel value is set to zero and writer domains will not update new cells until they see a clear sentinel.

Alternative to the Ordering Manager implementation exist. Lock-free priority queues [73] provide this function along with good performance. Some protocols may require their use, but the simpler mechanism has proven sufficient for this work.

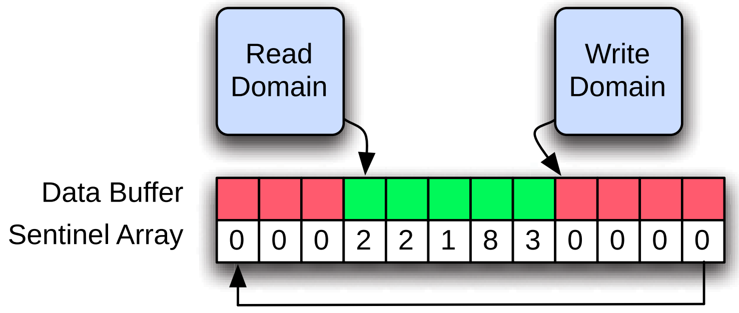


Figure 4.5: The Ordering Manager controls access to a shared ringbuffer with a sentinel array.

4.6 Userspace Test Rigging

The Dominoes Scout implementation provides a flexible tool for testing because the entire system is structured as a programming library. Using the driver router, the system can be plugged into userspace libraries for direct NIC access. The standard tool for this on Linux is `libpcap`.

`libpcap` provides an API for capturing and injecting raw network traffic from a physical interface. Because Ethernet is a broadcast medium, `libpcap` allows software to communicate using a different address space than the host. The native network implementation, in this case Linux, simply drops those packets. This eliminates any interference from the host. `libpcap` also introduces some difficulty. Data arrives from the network and is placed directly into a kernel buffer, but protocol processing occurs in userspace. This necessitates a copy in the Scout driver. This copy additional overhead and is not incurred by in-kernel implementations. Additionally, high-performance network device drivers coalesce interrupts to improve system response. An effect is that packets are coalesced into bundles and the number of overall network requests that the protocol system has to process is lower. `libpcap` does not capture this behavior. Instead, data is delivered and sent one packet at a time. A system call is needed for each incoming packet, which has a similar performance impact as one interrupt per packet. The performance improvements given later are achieved despite these disadvantages.

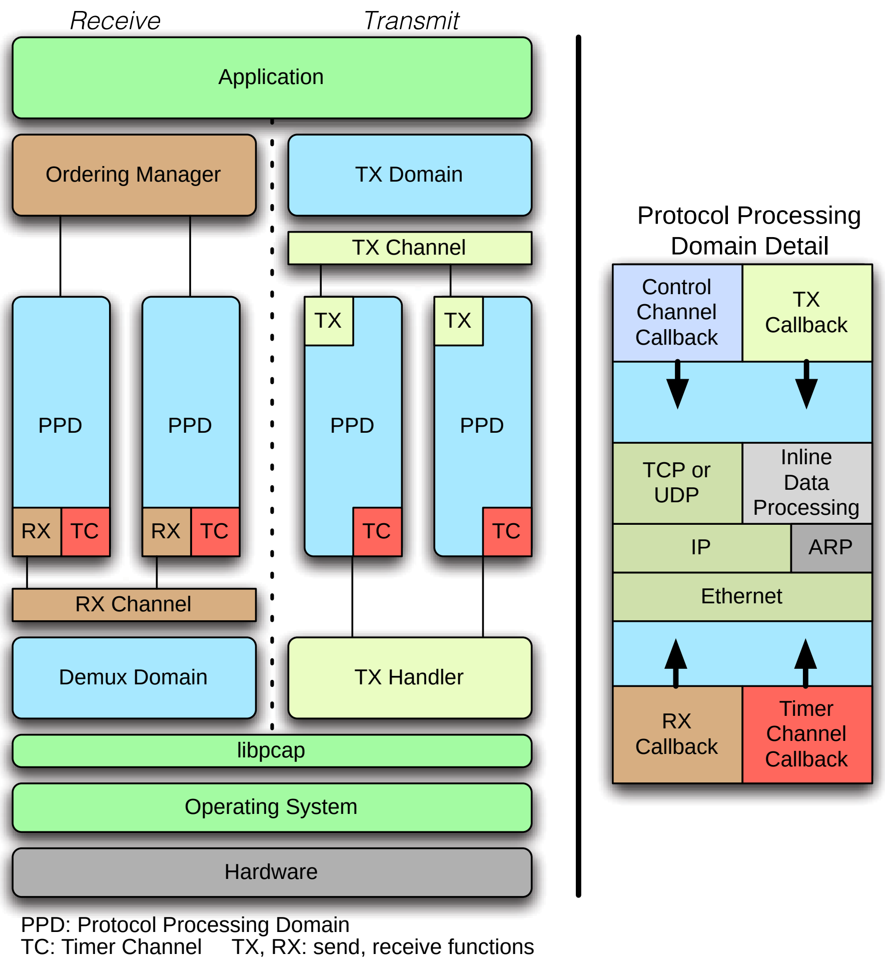


Figure 4.6: Dominoes Test Program Architecture

Figure 4.6 illustrates the complete test system's processing model. In this case, the application is simply an aggressive network use thread (i.e., a tight loop around either `read()` or `write()`). This ensures maximum possible usage of the network resources. The system can operate in either transmit or receive mode. The architecture is slightly different between the two code paths.

Transmits require the application to publish data to a TX channel. All processing domains are subscribed to this channel. When data has been processed and is ready for transmission, the domains then call a special TX handler function which interfaces with `libpcap`. This allows many domains to send simultaneously. Receive operates similarly, but in reverse. Incoming packets are published to a special RX channel by the demux domain that monitors `libpcap`. The processing domains then complete their work before delivering the data to the application by a call to the Ordering Manager. The Ordering Manager is needed to ensure that concurrent deliveries do not introduce data reordering. Some data, such as acknowledgements are consumed by the domains and are not passed to subsequent levels.

With this design, each processing domain services one TX channel, one RX channel, one timer channel, and several low traffic service channels that are used to setup Scout paths and perform Subscription management. This requires significant computational resources. Most domains require a dedicated context for best performance, although the TX domain and the application can often share a thread or processor. For an eight processor system, this leaves six processors for protocol work.

4.7 Summary

The replicated implementation of Scout networking is built to be as simple as possible, while preserving the semantics of the IP, UDP, and TCP protocols. Using Dominoes, this implementation provides a standard compliant protocol stack that uses replication for processing, where appropriate. Following the goal of the replication approach, the design targets a throughput increase for CPU bound protocol processing. Chapters 5 and Chapter 6 discuss the performance of the implementation along with the drawbacks of using replication for this type of processing.

Chapter 5

Performance

This chapter presents performance results of the replication approach, the Dominoes framework, and the replicated Scout networking stack. The performance results of software systems are dependent on the platform in which they are examined. Therefore, this chapter describes both the testing hardware and software environments. This chapter combines performance figures with description of testing methodology as well as analysis of the performance results.

5.1 Testing Platform

Empirical results are especially sensitive to variation in the testing environment. All the tests in this chapter maintain a consistent testing platform to provide fair performance comparisons. The tests in this chapter use two types of machines. Both contain 64-bit x86 processors with configurations as shown in Table 5.1. One configuration uses processors from Intel and the other AMD. The performance characteristics of the two platforms are remarkably similar, with both generating nearly identical throughput numbers for network send and receive. The differences in configuration produce almost no variation in the performance of network protocol processing. Both configurations run Linux and a standard set of system software.

	Intel	AMD
Processor	Xeon E5410 @ 2.333 GHz	Opteron 2376 @ 2.311 GHz
Core count	4 cores $\times$ 2 sockets	4 cores $\times$ 2 sockets
Cache size	12 MB L2	512KB $\times$ 4 L2, 6MB L3
Bus speed	1333 MHz Parity FSB	1000 MHz Hypertransport
Memory	8 GB DDR2-667 ECC	32 GB DDR2-667
Network Interface	Sun Dual XFB 10GbE	Intel NetEffect NE020 10GbE
Transceivers	Intel SR XFP	Finisar FTLX1471D3BCL
Bus speed	PCIe 8$\times$	PCIe 8$\times$
Switching Hardware	Fujitsu XG2000 10GbE	
Disk	SAS 500GB 7200 RPM, 16MB cache	network booted
Operating System	Ubuntu Server 10.04	Fedora 10
Kernel	Linux 2.6.32	Linux 2.6.27
Compiler	gcc 4.4.3	gcc 4.4.3
Libraries	libpcap 1.1.1 glibc 2.11.1 openssl 0.9.8	libpcap 1.1.1 glibc 2.9 openssl 0.9.8

Table 5.1: x86 Test Hardware Configuration

Throughout this chapter, experiments are run without debugging symbols present and with a GCC optimization level of `O2`. This provides parity when testing different parallelization strategies or when comparing against off-the-shelf implementations such as Linux. Hand tuning the generated code is not likely to produce significant performance increases because production networking stacks must provide good throughput in a variety of environments. The networking implementations discussed here already optimize load/store behavior when allocating memory. Unless otherwise noted, all the tests in this chapter use the Intel machine to generate the performance results and the AMD hardware as the network peer.

5.2 Synthetic Test

This section tests the feasibility of the replication approach with a synthetic workload. Synthetic tests evaluate the general approach. They also indicate which types of workloads are best suited to replication. They remove all networking and architectural issues from the computation and allow direct measurement of the scalability of replication.

The tests use a simple computation kernel that performs memory accesses based on a specified parallelization strategy. Time is split between doing sequential work concurrently and performing parallelizable operations. There are three strategies, each differentiated by the type of parallelizable operation.

1. Coarse-grained locking

   Parallelizable operations require access to shared state. A single mutex is used to serialize access to the state. The computation kernels acquire the mutex, complete their work, and then release it.

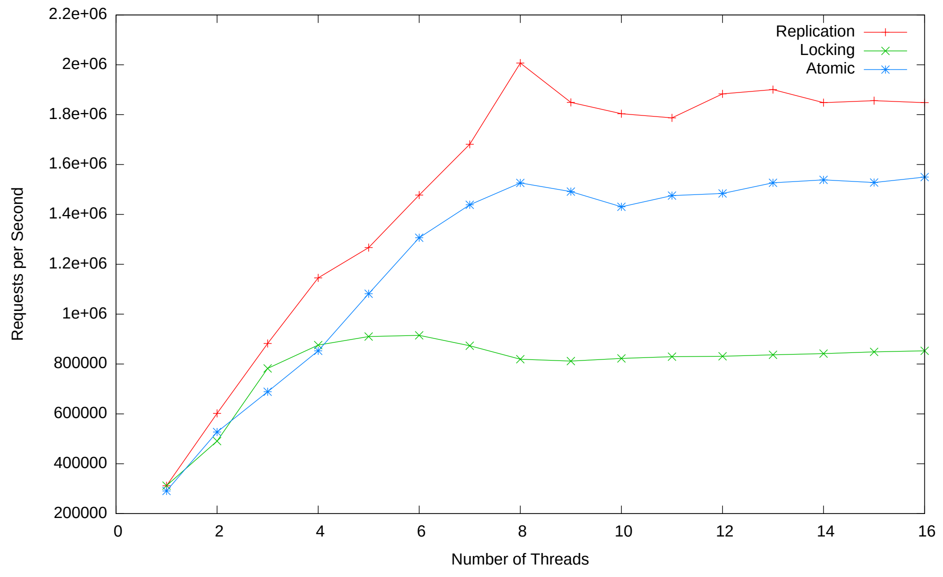
2. Atomic operations

   This strategy is similar to coarse-grained, except that updates are performed with hardware atomic operations. All threads update a single shared state.

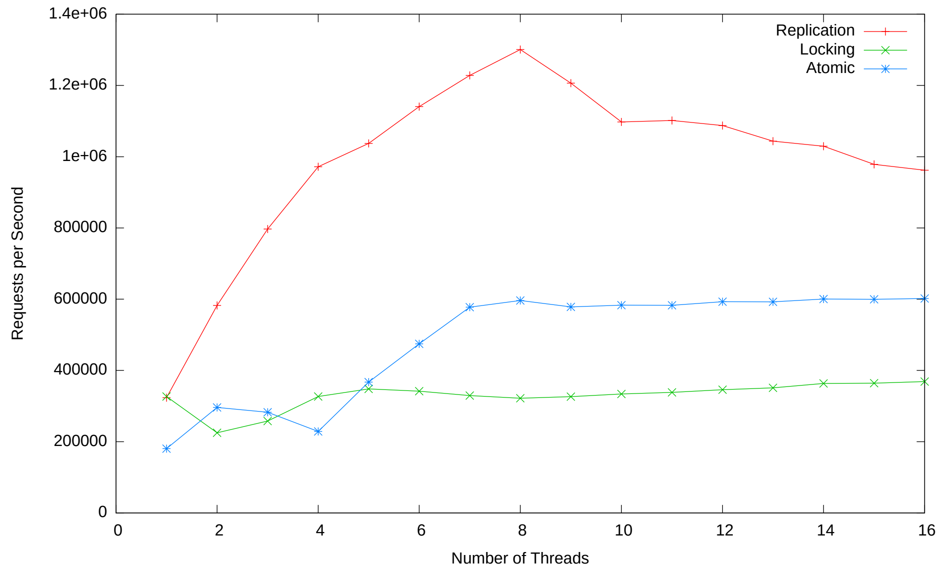
3. Replication

   Strategy 3 uses a replicated global state. No locking is needed because each replica has complete information. The state is assumed to remain consistent based on per-thread computation only. No coordination occurs between replicas.

Each of these strategies is tested with different ratios of parallelizable to sequential work. Figures 5.1 and 5.2 illustrate the results of a simple, synthetic scaleability test. In this case, the AMD test machine is used. It has eight processing cores, but the tests are run with up to sixteen threads. The additional cores test the performance of the strategy as the system becomes over-committed.

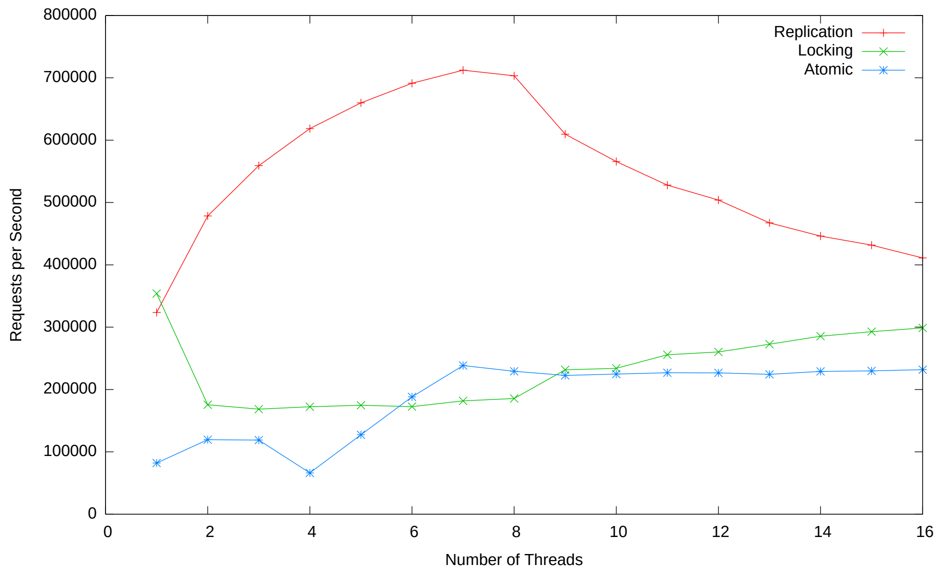


(a) 100:1 Sequential to Parallelizable Work

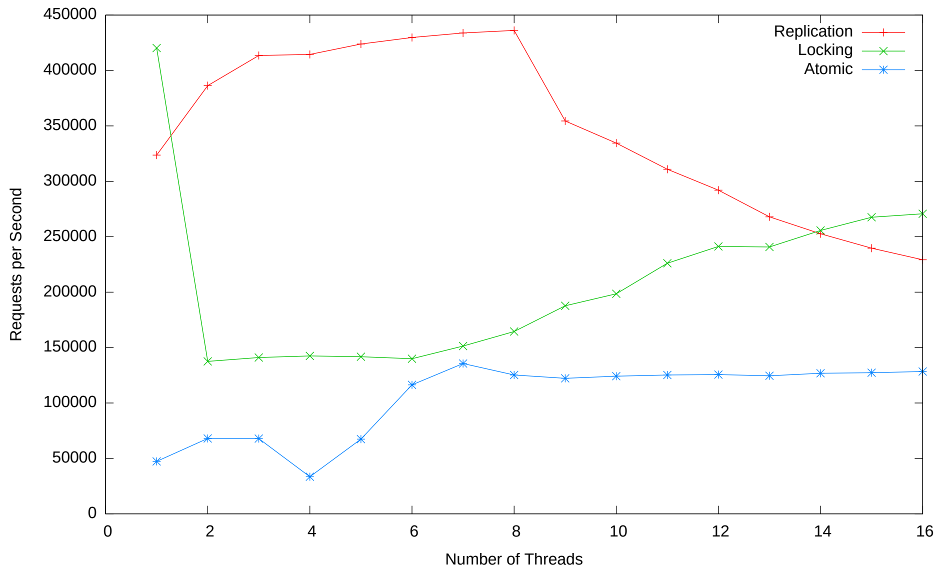


(b) 10:1 Sequential to Parallelizable Work

Figure 5.1: Simulation of Request Processing with Different Methods of Consistency Management



(a) 2:1 Sequential to Parallelizable Work



(b) 1:2 Sequential to Parallelizable Work

Figure 5.2: Simulation of Request Processing with Different Methods of Consistency Management (continued)

At 100:1, almost all of the processing is parallelizable and scalability is very good for all strategies. 10:1, in figure 5.1(b), is of particular interest, because it matches closely with the expected performance of network protocol processing. Replication allows for a speedup of 4-5x, which far outpaces the locking mechanisms. This is roughly 50% of the theoretical maximum, but still encouraging. Architectural issues cause the performance elbow at four cores. The dual socket test machine has shared cache for CPUs 0-3 and for CPUs 4-7. Stepping over the 4 processor boundary causes a fixed performance hit, but maintains scaling. This closely mimics the scalability problems of common protocols which were described in Chapter 2. In particular, the traditional locking scheme (coarse-grained) shows virtually no speedup at any scale.

At 2:1, the ratio of parallelizable work is quite small. Performance using replication is slightly better than expected from the theoretical results. Additional CPUs allow access to more cache and additional memory bandwidth. Locking techniques provide no performance benefit as the additional cores spend their entire execution time waiting to run. 1:2 is an even more extreme example. In this case the portion of parallel work is only 33%. Meaningful speedup is not possible.

One drawback of replication is the projected poor performance when over-committed. Additional threads in the locking simulation incur additional waits on the locks, but do not generate computational burden when they are sleeping. Replication increases the workload with each replica because of additional, redundant work. In the case where there are more than eight replicas, the added threads are performing more work, but provide zero performance benefit because they do not increase parallelism; their work simply becomes scheduler overhead. Locking or atomic strategies do not increase the amount of work as the number of threads increases. Additional threads are simply scheduler overhead.

While the results from this test strongly support a replication approach, they illustrate the importance of properly mapping the computational resources to the task. Over-committing the system can lead to performance collapse.

5.3 Request Processing Throughput

As described in Chapter 3, the ringbuffer-based channel provides the crucial function of delivering data in and out of Dominoes. This section evaluates the throughput of Dominoes in two configurations: single producer / multiple consumers and multiple producers / multiple consumers. The framework must handle high throughput in order for the system, as a whole, to process high-bandwidth data.

5.3.1 Single Producer, Multiple Consumer

The single producer case tests the framework's suitability for high-performance networking. At a minimum, channels should provide sufficient throughput – with reasonably sized packets – to saturate a modern network link. With standard 1500 byte Ethernet frames, this requires 8.33×10^5 requests/sec to achieve 10 Gb/s. While this is not a hard limit, it provides a baseline to evaluate channel performance.

Figure 5.3 shows the per-core and aggregate dequeue rates from a ringbuffer channel. In this case, one core is dedicated to handling "interrupts" and publishes data to the channel. Up to seven domains subscribe to this channel and dequeue the data. The green line shows the total number of requests than can be enqueued by the publisher. It decreases as the number of subscribers increases, but it does not drop off linearly. The green line shows the total number of successful dequeues from the system. This is also the product of enqueues and subscribing threads. The 8.33×10^5 request/sec threshold is shown in blue. Aggregate throughput increases to the maximum number of CPUs, indicating effective use of hardware resources.

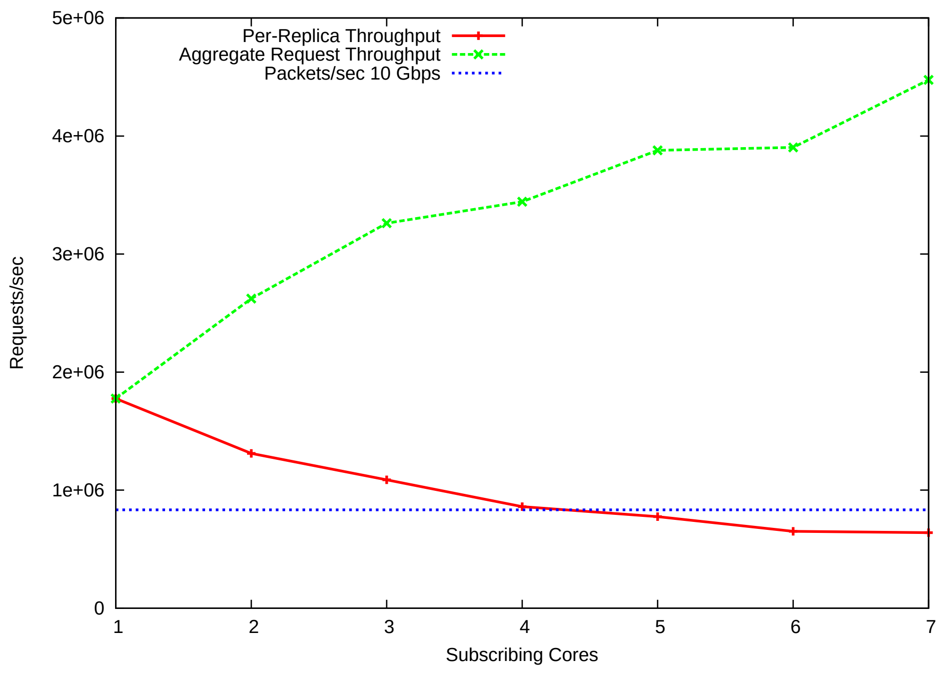


Figure 5.3: Dominoes Channel Single Producer Throughput Performance

5.3.2 Multiple Producer, Multiple Consumer

Dominoes channels also support multi-producer channels, which is important for delivering data from domains to the network. Table 5.2 shows throughput with varying numbers of simultaneous publishers. Only a fully loaded system with seven publishers drops below the target throughput of 8.33×10^5 req/sec with a single subscriber. This is sufficient for a 10 Gb/s link with multiple domains sending data.

As an implementation target, 8.33×10^5 req/sec is artificially pessimistic. This assumes that the maximum message size matches the maximum transmission unit (MTU) of the network link. For high-speed links, the number of requests is far fewer due to interrupt coalescing [53] on data receipt or segmentation on transmission. The data travels through the protocol stack as bundles of dozens to hundreds of packets. This greatly decreases the required number of channel requests. For example, coalescing receive-side (RX) interrupts

	Throughput (req/s)		
# Publishers	**1 subscriber**	**2 subscribers**	**4 subscribers**
1	1693833.9	1294543.1	915600.0
2	1780225.3	1376557.2	894457.0
3	1191606.3	1086043.6	805401.6
4	1037426.9	851383.2	645373.6
5	871464.8	849827.1	-
6	847435.4	639468.0	-
7	660352.2	-	-

Table 5.2: Throughput for Multi-Producer Channels

to a frequency of 100μs reduces the required number of requests to approximately 10^4. This is well within the performance capabilities of any publisher/subscriber combination. Interrupt frequencies in this range are well below round-trip times (RTT) and are realistic for existing hardware.

5.4 IP Suite

While the synthetic test results are important validation of the replication approach, standard protocols are the true measure of its effectiveness. Many performance features of real networks are difficult to capture with simulation. These features include memory architectures, link delays, and operating system noise. The approach must perform well in this environment in order to deliver increased throughput to applications.

This section evaluates two protocols: UDP/IP and TCP/IP. UDP most closely matches the synthetic test because it is stateless. Because it is widely deployed, it accurately measures how replication increases throughput to applications that use stateless protocols. TCP is stateful and considerably more complicated. TCP performance illuminates the strengths and weaknesses of the replication approach when managing connection state. The tests process each protocol with varying number of domains and report the throughput and thus the scalability.

5.4.1 UDP

The UDP/IP protocol provides a real-world implementation that closely matches the model of the synthetic test. Processing Ethernet, IP, and UDP headers requires significant per-packet processing, but UDP is stateless. There is no parallelizable portion of the protocol processing. Unlike the synthetic benchmarks, the protocol allows testing with real applications and networks.

The per-packet costs of UDP processing are quite low. Together with IP, the total header length for both protocols is only twenty bytes and the payload portion of the packet does not need to be inspected for delivery. Because UDP is stateless, virtually every implementation allows multiple packets with the same source-destination pair to be processed concurrently. This gives near linearly scalability for production implementations such as Linux and Solaris.

Figure 5.4 shows receive throughput of the UDP protocol with the Dominoes Scout implementation. Without unmodified UDP processing it quickly scales to the performance ceiling of the `libpcap` driver system: 3 Gbs. This is shown in red. More importantly, it also scales well when an artificial load is applied. In the example, a QoS type load of four memory operations per byte received are introduced into the processing stream. This corresponds to the cost of a typical string search. The throughput is shown in green. At six processors, performance approaches the bandwidth peak of the driver again.

5.4.2 TCP

The following tests shows the throughput of the TCP receive and send operations with multiple processing domains. TCP/IP presents a greater challenge for the replication ap-

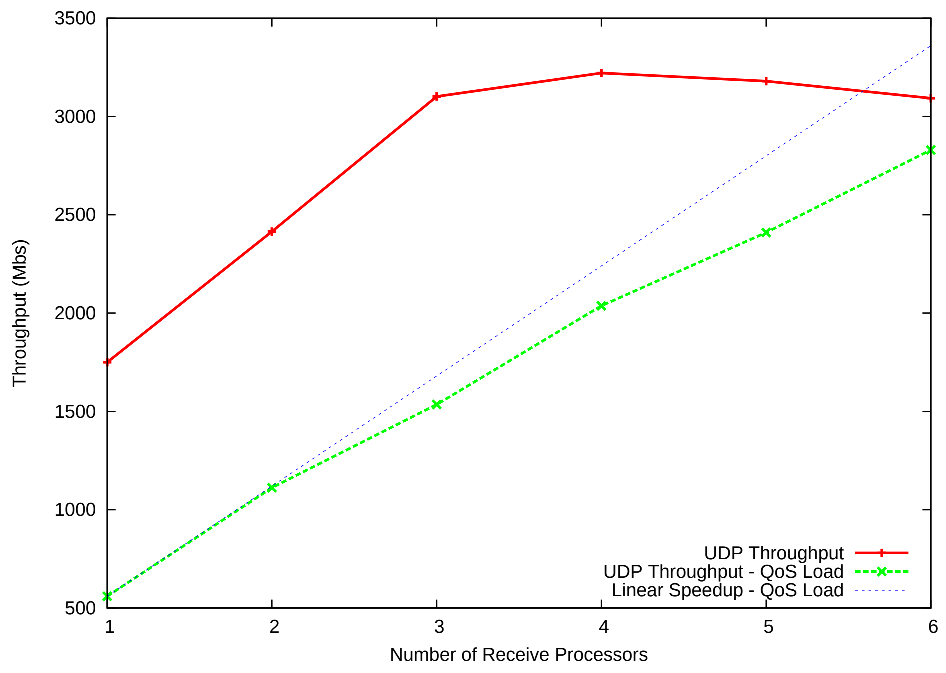


Figure 5.4: UDP Throughput on Eight Processor Server

proach because there is significant local state. Sequential work includes flow control and ordering operations, which can vary in cost, depending on the networking environment. Because the TCP receive window has a fixed maximum size, one replica may be limited by the processing speed of another. All replicas must progress in order for the window to be advanced. Nevertheless, TCP receive throughput – which has traditionally been the computationally expensive side of protocol processing – benefits from the replication approach.

The test systems have eight free cores when an application and driver/publisher are running. This leaves six CPUs for the testing. All tests vary the number of domains from one to six. The NIC supports hardware checksumming, which is enabled.

TCP Receive

Figure 5.5 shows the throughput with varying number of cores and different incoming message sizes. Smaller messages require more header computation per payload byte. The sequential component of the request processing is thus higher for small messages and scalability is more limited. The additional per-byte cost also slows the connection, regardless of scaling issues. Nevertheless, replication provides repeatable performance gains for all message sizes. These improvements fall short of the project goals, but are still substantial.

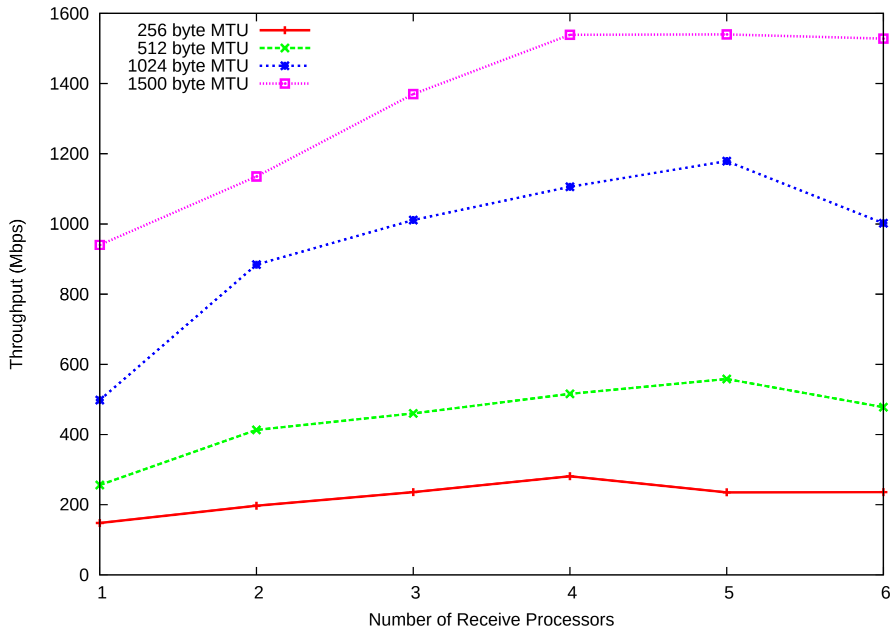


Figure 5.5: TCP Receive Throughput with Varying MTU

In Figure 5.5, the six core case is a statistical outlier because the it represents a slightly overcommitted system. In addition to the six processing domains, there is a domain for handling driver events as well an application thread. That results in near complete utilization of the eight available hardware threads. Other system load such as instrumentation and I/O must compete against the test framework for resources. While this does not result in catastrophic performance loss, as in the synthetic test, it does cause measurable decrease in aggregate performance.

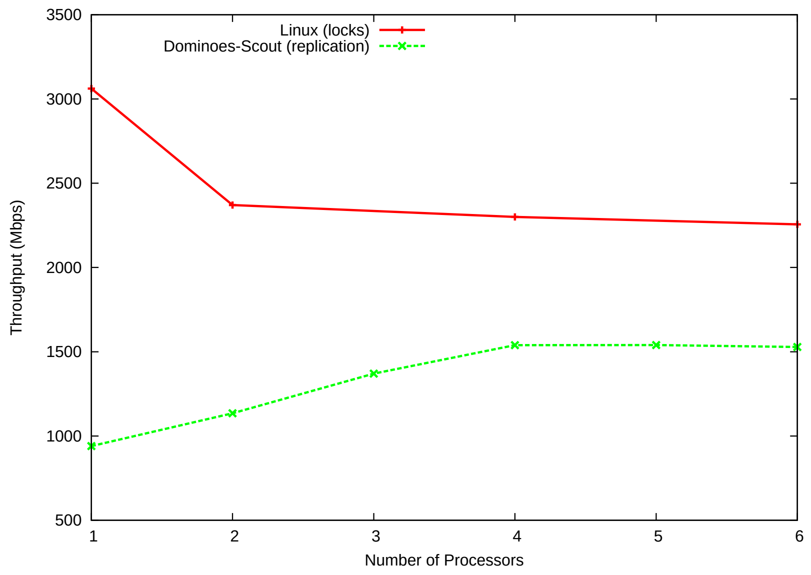
Figure 5.6 compares the scalability of the replication approach to the TCP implementation in an unmodified Linux 2.6.32 kernel. This was performed with 1500 byte packets and no additional processing. The Linux implementation locks the TCP connection state in the event that two or more CPUs attempt to process different packets, from the same connection, concurrently. The Linux implementation is in-kernel. Consequently, its single threaded performance is higher than Dominoes' userspace implementation. With multiple processors, the Linux performance degrades while Dominoes increases. The raw throughput numbers are higher for Linux, but the scalability differences are clear when the results are normalized.

As seen in Chapter 2, the locking approach provides no scalability because processing threads are mutually exclusive. Linux performance degrades as more processors become available because the cost of the lock increases when it is no longer available in L1 cache. Alternatively, replication scales as seen in previous figures. When normalized scalability becomes clear.

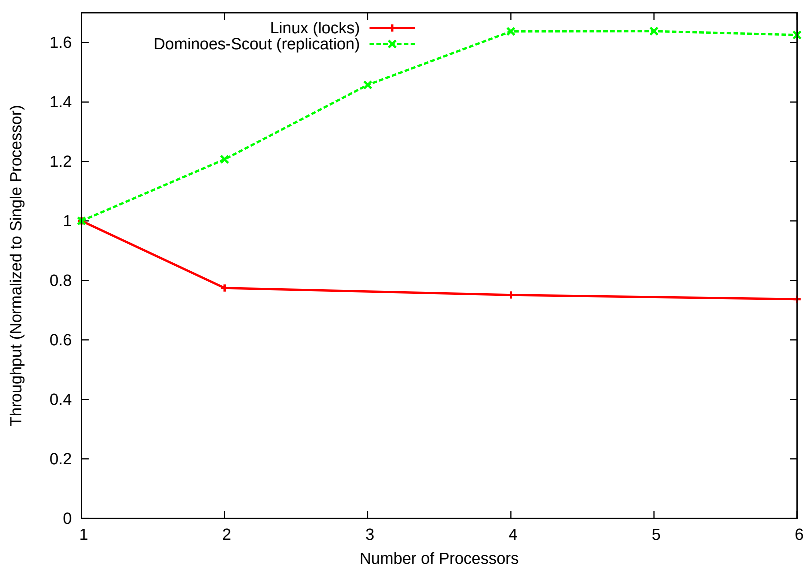
TCP Send

Figure 5.7 shows the result of replication on TCP send performance. Unlike receive, send is not able to benefit from replication and throughput degrades rapidly as processors are added. There are multiple reasons for the performance decrease. Ordering semantics require excessive serialization of transmitted packets and the TCP send buffer code requires synchronization between domains.

When multiple domains process packets simultaneously, they may deliver the data to the network device out of order. This results in the data being transmitted and then delivered out of order. The result is two types of tests: a test where the data is sent asynchronously and a test that synchronizes send packets and the associated buffers. A



(a) Throughput



(b) Normalized Throughput

Figure 5.6: Scalability of Replication vs. Lock-based Linux

more detailed analysis of the problems with send are discussed in Chapter 6.

Performance with a single processor is in line with receive results. Single core through-

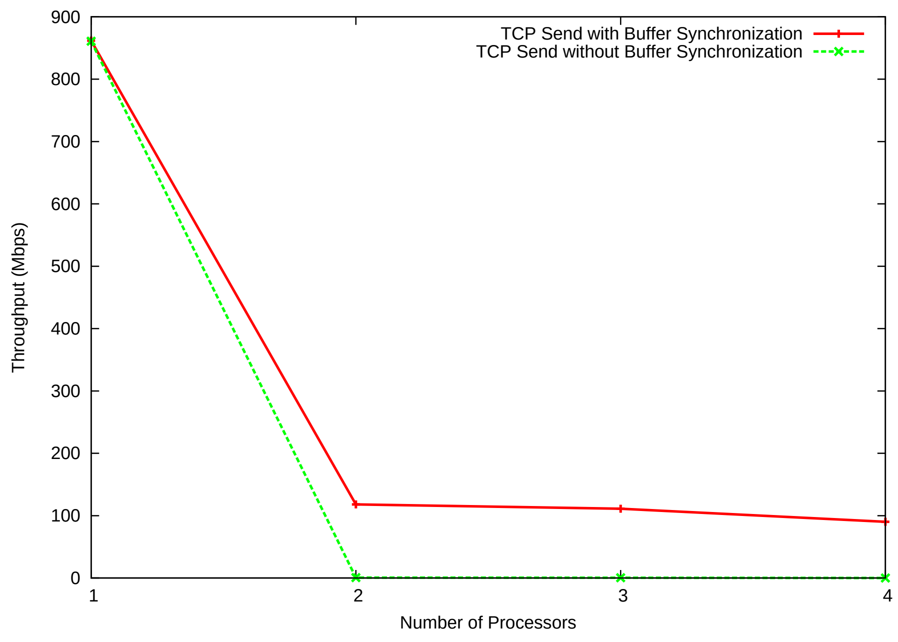


Figure 5.7: TCP Send Throughput

put is approximately 900 Mbps. Adding the second processor incurs so much overhead that performance drops below 200 Mbps and degrades further as more processors are added. In fact the eight processor case is not shown here because throughput is very low and the results are unpredictable due to frequent timeouts. Without synchronizing the sent buffer performance is worse; throughput is negligible with multiple replicas.

5.4.3 Inline Data Processing

The ratio of parallelizable to sequential work affects the scalability of the system. Inline data processing increases the sequential portion and increases the scalability. The performance using replication improves when the system must do additional, inline processing. One common example of inline processing is encryption/decryption. The following tests adds additional load to the TCP receive operations to test this behavior.

Figure 5.8 demonstrates scalability when performing inline data decryption. For this example, each incoming message is decrypted using the Advanced Encryption Standard (AES) cipher with a 256-bit key [52]. This operation is performed using the standardized `EVP_DecryptInit_ex()` and `EVP_DecryptUpdate()` API from the OpenSSL [79] libraries. The additional load reduces the throughput with a single processor by approximately 60%. However, scalability improves because the additional work is only perfomed on the primary. With five or more replicas, 1 Gbps is easily attained for a TCP connection with *software* data decryption.

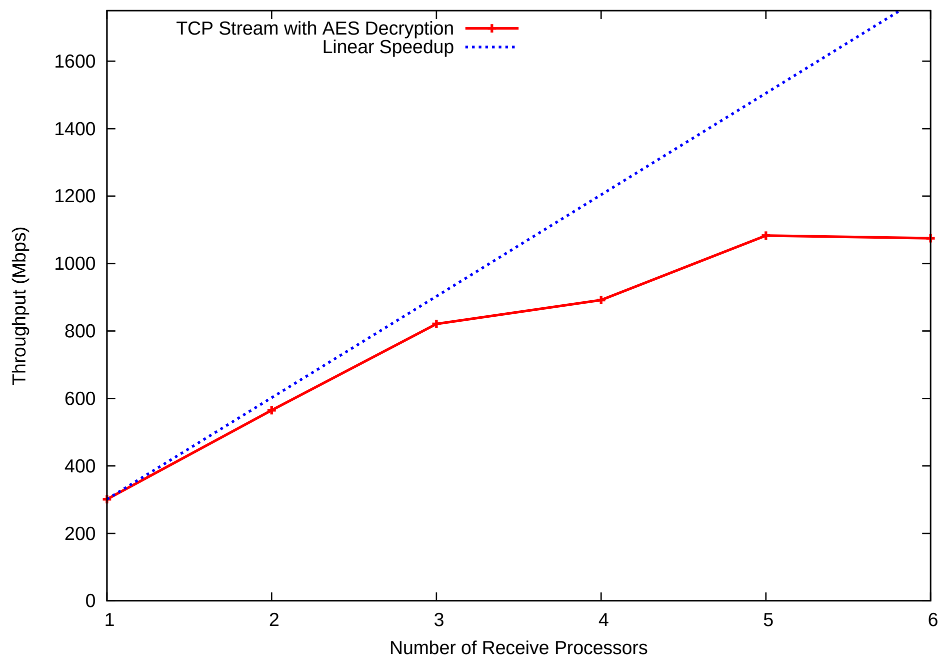


Figure 5.8: TCP Receive Throughput, Performing 256-bit AES Decryption

5.4.4 Specialized Primaries

TCP includes two principal operations that have global effects: data delivery to the application/network and acknowledgement generation. Although timeouts also have global visibility, they occur far less frequently. Throughout this work, the replication approach assigns all parallelizable work to a primary domain for each this request. This results in the performance improvements that we have seen in previous sections. Another strategy exists.

Rather than assign parallelizable work to the primary, the two operations could be split between two processors. One domain handles all payload operations (data delivery, inline processing, etc.) and another generates acknowledgments. This specialized adaptation of the protocol cannot produce large speedups because only two processors are used. However, it is simple to understand and relatively easy to implement. It is only possible because the replication approach gives lock-free access to the connection state.

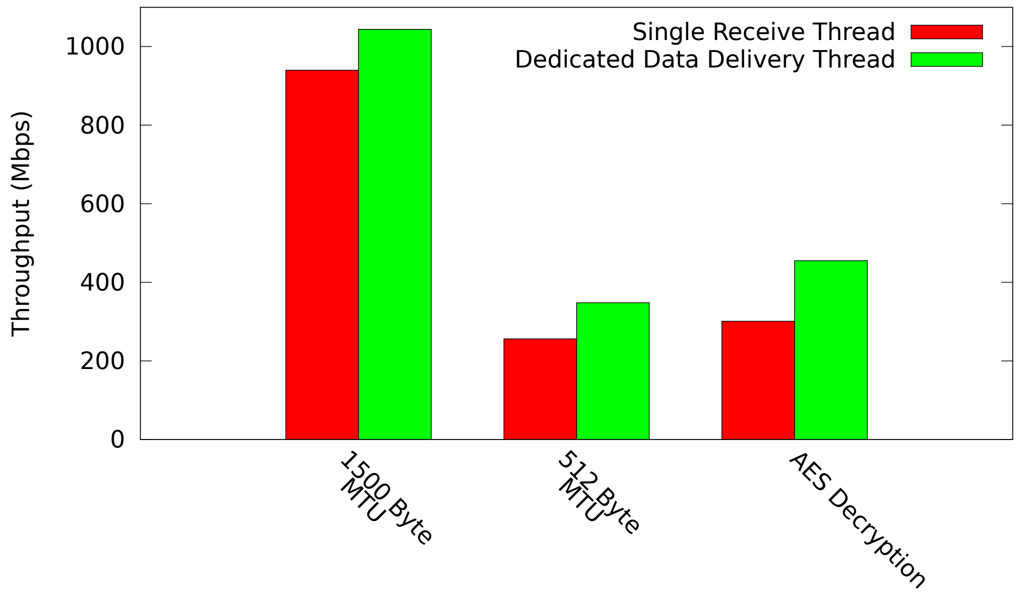


Figure 5.9: TCP Receive Throughput with Dedicated Payload Thread

In Figure 5.9, TCP receive throughput is tested with data delivery and acknowledgements assigned to different domains. With standard 1500 byte packets the gains are moderate: approximately 15%. With smaller packets, the marginal gains are much higher. Linear speedup (i.e., doubling) is not possible here because the two operations do not have the same cost. The third result shows the performance with AES decryption added to the data delivery function and 1500 byte packets used. The performance is much lower than the first test, but the gains are larger. This indicates that the cost of generating acknowledgements is expensive enough to be worth separating computationally. As an added benefit, the ordering manager is not needed in this test because a single domain copies all bytes to the application buffer.

5.5 Summary

This chapter presents the results from a spectrum of performance tests. The x86 testing platform is capable of high-speed networking with minimal hardware offloading. This matches a vast number of deployments from home PCs to enterprise environments and commodity HPC clusters. The Dominoes framework is able to serve millions of requests per second with many simultaneous subscribers.

The replication approach provides encouraging results on tests with synthetic loads. It achieves speedup of 5x with parallelizable to sequential work ratios of 10:1. Full protocol implementations also benefit. UDP performance scales well, although a single publishing domain can become a bottleneck at high speeds. TCP receive performance is improved, but TCP send has issues that make it unworkable.

Chapter 6

Analysis

The performance results from Chapter 5 illustrate both the success and failure of replication in providing scalable, high-bandwidth networking performance. Some results match the design expectations. Others give new insight to effectiveness of replication, the Dominoes framework, and the network protocol implementation.

This chapter analyzes the behavior of replication systems when applied to network processing. The first section discusses how Dominoes is effective in implementing replicated systems and where it proved to be a performance or design limitation. The following two sections discuss stateless and stateful protocol behavior.

6.1 Dominoes Framework

6.1.1 Overview

The complete system used to generate the results in Chapter 5 is a large code base, with approximately 35,000 lines of ANSI C. The Dominoes framework effectively isolates nonreplicated functionality by encapsulating the replication mechanisms for resource allocation, scheduling, etc. This allows for quick and easy changes to the replication behavior by adjusting the framework or changing how the framework is used from within Scout.

The performance of the framework is very good. While there is room for improvements, channel throughput has shown to be adequate for both this research and for a production network protocol stack. Domains are sufficiently thin wrappers around scheduling functions that they provide good responsiveness to networking events as well as fast processing. They fall short of a full kernel thread because they do not gracefully yield the processor when no work is available. This could easily be added, but it is not the focus of this work.

There was considerable effort to develop and test Dominoes before work could begin on the network protocol implementation. This was rewarded with decreased effort in debugging the whole system as well as decreased time between experiments. For example, load distribution measurement, such as Figure 6.1, required only simple changes while instrumentation that is not at an API boundary is much more involved. Applying Dominoes to a networking implementation with real protocols revealed some shortcomings in its design.

6.1.2 Semaphores

Semaphores are an unexpected difficulty. The solution with Scout, given in Chapter 4, is not optimal. If a domain must wait because of a pending semaphore, the code trampolines back into the context scheduling routine. This allows other domains on the same context to continue servicing requests, such as signaling the semaphore. This solution is not generally applicable because recursive calls can lead to unrecoverable scheduling dependencies or even deadlock. It would be better to save the calling stack of a sleeping domain and designate another domain to handle the subscribed channels. This is very close to the behavior of a full process control system and has larger engineering scope than the initial Dominoes implementation. The current context model is too simple.

6.1.3 Dynamic Con guration

The framework is not dynamic. Domains and channels are setup once and cannot be easily reconfigured. This may seem appropriate in a research environment where the entire system can be reset for any type of failure. However, the system cannot adapt to network events. For example, it might be advantageous to reduce the number of replicas after a timeout occurs. Dominoes cannot change the number of subscribers once a channel is running. New network connections require full initiation of the framework as well as new channels and new domains. This is sufficient for throughput testing but likely increases the setup time for the connection. A more efficient system would reuse domains in the same way that web server reuses threads.

6.1.4 Load Balancing

Dominoes uses simple strategies to assign primaries, which is effectively load balancing. The publisher of a request can specify the primary directly. This is useful when directing a series of requests to a single domain, such as during connection setup in TCP. The default behavior chooses the first available domain as the primary. A domain that finishes requests more quickly will receive a high percentage of primary requests.

Figure 6.1 illustrates fluctuation of the load on the processing domains. The data is sampled every 27400 requests, which corresponds to time intervals of 100-200 ms. This is sufficiently long to capture several scheduling time slices. Primaries are assigned on a first come, first served basis. A domain will service more requests if it completes previous requests more quickly. In this example, the load is not distributed evenly across all five processors. CPU 1 tends to be very slow and is assigned as the primary for fewer requests.

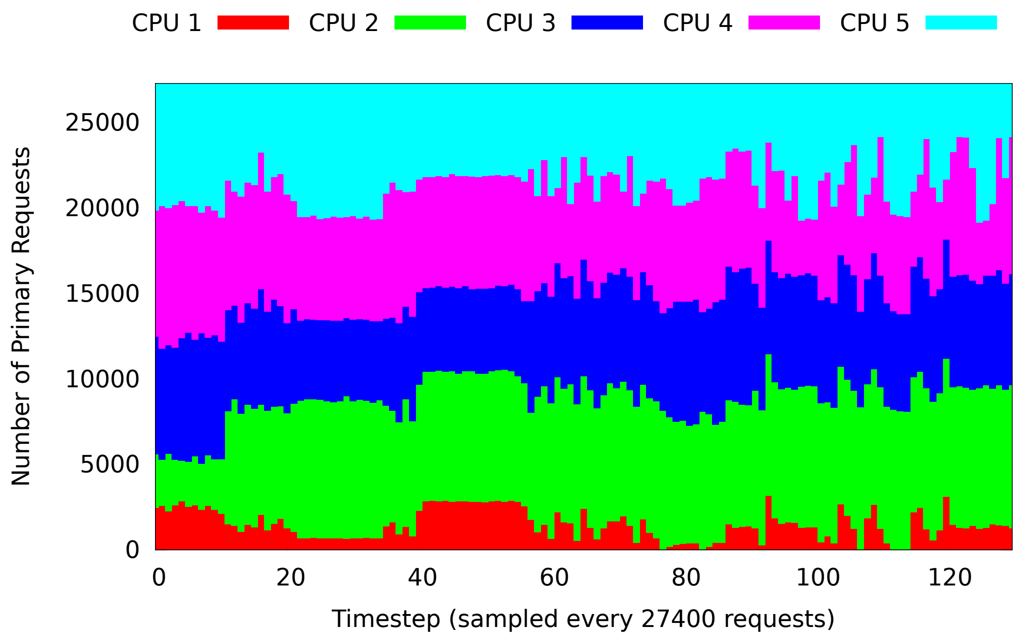


Figure 6.1: Load Fluctuation of TCP Receiver with Five Processing Domains

Figure 6.1 shows that "greedy" behavior distributes the work from the channel without starving any domains. This distribution is dynamic. The portion of work for each domain changes based on how much processing time is available – how many times it is first in line to dequeue a request. However, this is not a comprehensive study of the load balancing problem. Further work on domain load balancing is discussed in Chapter 7.

6.2 Stateless Protocols

The performance behavior of stateless protocols, such as UDP/IP, mirrors the Dominoes framework. The sequential work is essentially zero and each packet can be processed independently. Replication provides speedup similar to the traditional threading methods that are used in almost all mainstream implementations. More importantly replication does not incur heavy overhead for this type of system. It can be used with small penalty, even if the throughput considerations do no merit it.

For performance, stateless protocols are not able to take advantage of the principal strengths of the replication approach. With no state, there is nothing to replicate and the system behaves as a multithreaded request processor with a centralized, although lock-free, queuing structure. For a single application, this does not provide a compelling reason for the added complexity. However, it does allow for simple out-of-band processing, such as a traffic monitoring.

6.3 Stateful Protocols

Stateful protocols show performance gains that are not possible using synchronization techniques such as locks. TCP sending is inherently more difficult than receiving, although receive processing tends to be more expensive. Overall, the approach achieves better scalability than other network protocol parallelization techniques. Careful management of request ordering maintains consistency in some cases but is not sufficient to allow speedup of all operations.

The Dominoes channel object guarantees ordered delivery of requests. Initially, this seems like a small benefit because data can be reordered by the network fabric. The real power of this guarantee is that all domains will see requests in the same order. As a result, all state updates happen in the same order. Local states may be inconsistent because of scheduling variance or unbalanced processing load but they do not diverge. This inconsistency is bounded by the length of the channel queue. When the length is one, all the replicas must contain the same state because they have processed the same set of requests. For TCP, this queue length can never be larger than the receive window.

Given that the sequence of state updates produce the same effect, the replicas generate the same response to events. If an arriving packet is out of order, then it is out of order for all replicas. Each replica makes the appropriate state changes (such as congestion window reduction) and generates a response, if necessary. These semantics are sufficient to keep the local state functional. Unfortunately, they fall short of providing good performance in all cases.

6.3.1 TCP Receive

Incoming TCP processing receives a significant performance benefit from replicated processing. Ordered delivery to the application is slightly delicate, but the rest of the processing functions well when the channel ordering semantics are maintained.

Scalability misses the target of 10x speedup from the theoretical maximum. This is not surprising as the theoretical result ignores all architectural considerations. The test architecture requires several memory copies for each incoming packet, which are expensive.

6.3.2 TCP Send

TCP send performance does not take advantage of replication as easily as receive. Two issues cause performance collapse as the number of processors is increased. The Dominoes architecture introduces a race condition for network sends and the buffer management code in Scout is not thread safe.

Out of Order Transmits

When multiple domains process packets simultaneously, more than one packet may be ready for transmission at the same time. This results in a race among the domains for access to the networking hardware. The API for `libpcap` allows concurrent access and will serializes send requests. However, Dominoes does not provide a mechanism to guarantee ordering. If packets are sent in the wrong order, TCP interprets this as lost traffic and requests a retransmit by generating a duplicate acknowledgement. The unsynchronized send test from Chapter 5 (Figure 5.7) shows the poor performance in this scenario.

With more than one thread, reordering happens so often that the effective congestion window size is only one MTU. The small window reduces the throughput to only a few kilobytes per second. This problem can be mitigated using a technique similar to the ordering manager, where each domain waits until all previous bytes are sent before it generates a transmit. However, the results of that test are limited by another synchronization issue in the buffer management system.

Buffer Management

TCP semantics requires buffering for all outgoing data. After data has been sent, it must be saved to handle the case where the corresponding packet is lost and needs to be retransmitted. TCP is a byte stream protocol. To minimize protocol overhead, the protocol sends as many bytes as possible with each packet, typically the link MTU. The application can perform send operations of any size and these writes are rarely the same length as the MTU. Therefore, the send buffer performs the secondary function of segmenting sent data.

The Scout buffer management code is not designed to handle replication processing. It is implemented as a list of message fragments. Concurrent access to the list of buffers can cause corruption. Straightforward adaptions are not sufficient to correct the problem. If multiple domains try to access the same buffer with explicit synchronization (such as a lock), the throughput drops dramatically. This is seen in Figure 5.7 from Chapter 5. The alternative is to provide a per-domain replica of the buffers. Unfortunately, the send requests from the application do not directly correspond to transmitted packets.

Primaries are assigned to send requests, which correspond to `write()` calls from the application. This indicates who is responsible for delivering the data to the network the first time. If the data needs to be retransmitted, that event is triggered by an incoming duplicate acknowledgement, on a different channel. There is no way to indicate who was initially responsible for the packet. This also leads to performance collapse because multiple domains can be sending the same packet at the same time. To correct these problems, the buffer management code needs to be designed for replicated processing.

6.3.3 Checksums

Checksum calculation may seem like a likely candidate to benefit from replicated processing. It is required by many protocols and depends only on local data. In fact it is one of the few operations that has been successfully offloaded to dedicated hardware. Yet, checksumming has requirements that make it difficult to replicate.

A simple strategy for replicated checksums would be to include the calculation only as part of the primary work. This means that the checksum is only calculated on one processor and that each processor only generates checksums for a fraction of the total packets. This performs well, so long as the checksums are correct. If a checksum is *not* correct, then errors are not detected by most replicas. This breaks the semantics of checksum routines. This leaves two options. Checksumming can be performed by every replica – which eliminates any benefit of parallelization – or the primary can notify other replicas of the result of each checksum calculation.

Checksumming Cache Performance

All incoming TCP packets include a checksum on the header and payload. Scout performs this check in software, but many high-performance NICs can provide this feature in hardware. Figure 6.2 illustrates TCP receive throughput with and without hardware checksumming. Recall that the Intel Xeon processor has a shared L2 cache for each set of four hardware threads. Using this memory model, the software checksum routine is quite expensive. It requires that every replica inspect every byte of data. The overhead is low, so long as all the replicas share the same cache. When the number of replicas increases to require a second socket [1], performance drops significantly because of additional memory bandwidth requirements and cache thrashing.

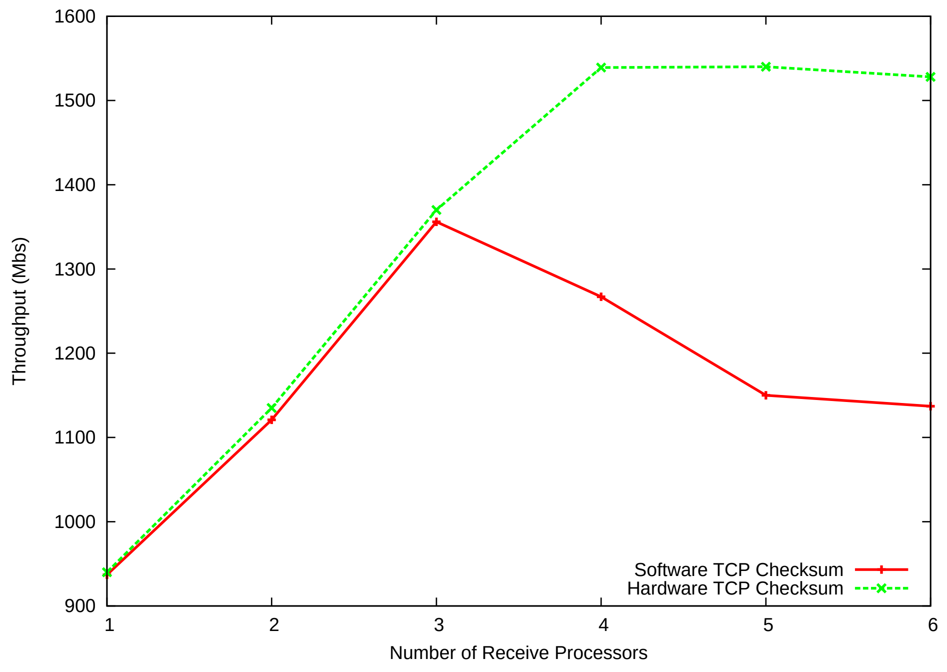


Figure 6.2: TCP Receive Throughput on Intel Xeon E5410

Figure 6.3 illustrates the performance difference between the Intel Xeon E5410 with a shared L2 cache and the AMD Shanghai CPU with individual caches. The slower clock and bus frequencies of the AMD processor result in poor performance with a single replica, but it is able to outperform the Intel architecture when caching effects become dominant.

Hardware checksumming avoids this problem. It reduces memory lookup for *non-*

[1]Four receive replicas plus one driver thread result in *ve* total threads.

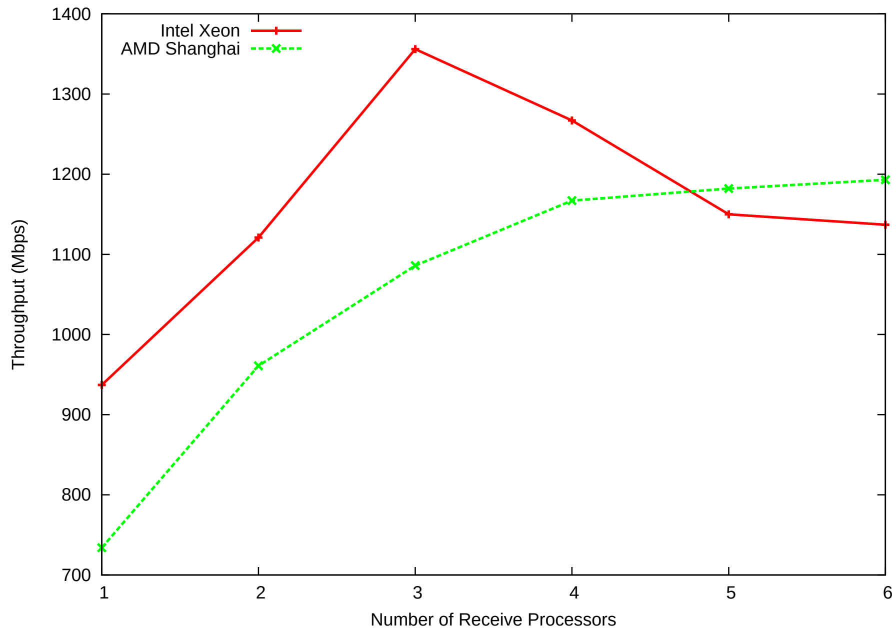


Figure 6.3: TCP Receive Performance with Software Checksumming for Shared-cache Intel and Individual-cache AMD

primary replicas to header data only[2]. It allows speedup, even without shared cache. Subsequent results in this chapter are given with hardware checksumming enabled, as virtually every 10 Gigabit Ethernet NIC provides this feature.

6.3.4 RX/TX with Different Replicas

TCP connections are bidirectional. TCP headers contain both a sequence number for identifying the sent data and an acknowledgment number for received data. An acknowledgement bit is provided in the TCP header flag field to indicate whether the acknowledgement number is significant, although the convention is to always set the bit after the initial connection handshake. The result is an essentially useless flag bit as well as an acknowledgement in every sent packet. Figure 6.4 illustrates this behavior between two

[2]The primary must copy each byte of the payload to the application

Linux hosts. Viewing the resulting trace with the Wireshark [76] tool shows that only one side of the connection sends data, but the ACK field is set in every packet.

```
No | Source       | Desintation | Info
--------------------------------------------------------------------------------------
1    192.168.2.2 -> 192.168.2.3   51484 > 1234 [SYN] Seq=0 Win=5840 Len=0
2    192.168.2.3 -> 192.168.2.2   1234 > 51484 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0
3    192.168.2.2 -> 192.168.2.3   51484 > 1234 [ACK] Seq=1 Ack=1 Win=5888 Len=0
4    192.168.2.2 -> 192.168.2.3   51484 > 1234 [ACK] Seq=1 Ack=1 Win=5888 Len=2896
5    192.168.2.2 -> 192.168.2.3   51484 > 1234 [PSH,ACK] Seq=2897 Ack=1 Win=5888 Len=1200
6    192.168.2.3 -> 192.168.2.2   1234 > 51484 [ACK] Seq=1 Ack=1449 Win=8704 Len=0
7    192.168.2.2 -> 192.168.2.3   51484 > 1234 [ACK] Seq=4097 Ack=1 Win=5888 Len=2896
8    192.168.2.3 -> 192.168.2.2   1234 > 51484 [ACK] Seq=1 Ack=2897 Win=11648 Len=0
9    192.168.2.2 -> 192.168.2.3   51484 > 1234 [ACK] Seq=6993 Ack=1 Win=5888 Len=2896
10   192.168.2.3 -> 192.168.2.2   1234 > 51484 [ACK] Seq=1 Ack=4097 Win=14592 Len=0
```

Figure 6.4: Trace of Simple TCP Connection with Only One Side Sending Data, Every Packet Contains Acknowledgment

Likewise, the sequence number field cannot be marked insignificant and will always indicate the current, expected data segment – even when packets are sent purely for acknowledgement. This leads to the requirement that every outgoing packet must contain both the current sequence number and current acknowledgement. However, regular data segments and acknowledgment packets are generated by different sources.

Acknowledgements are driven by data from the network while new, outgoing data originates at the application. Performance might be increased by splitting the send and receive processing between processors. One set of cores may handle incoming data and another set handles outgoing. This optimization is not possible with TCP because senders and receivers must access the same state.

Small changes to the protocol could allow parallelization based on data flow direction. For example, an additional flag bit could indicate whether the sequence number field is valid. Combined with a convention to only use the acknowledgement bit when needed, this would allow a processing thread to only maintain *half* of the processing state. Another thread could maintain the other half independently.

Chapter 7

Conclusions

7.1 Summary

This dissertation applies replication to network protocol processing. The replication approach provides an alternative to classical parallel programming models which use explicit synchronization techniques such as locks. Locking techniques limit the scalability of parallel networking protocol processing. Replication allows for lock-free processing for data on high-speed networking links.

To implement replicated protocol processing, this dissertation introduces the Dominoes framework. Dominoes provides the basic mechanisms for replicated request processing. It provides replicated queuing through the channels abstractions. Domains provide scheduling and state management. Dominoes supports millions of requests per second on commodity x86 hardware.

Dominoes is combined with the networking system from the Scout research operating system to create a replicated network stack that supports the standard IP suite of protocols. Replication increases the throughput of the protocols in some cases. Receive performance for UDP/IP and TCP/IP is substantially improved by replication-based parallel processing. Addition loads such as AES decryption also show good scalability. Some operations, especially TCP send, are not able to take advantage of parallelization as it is available in this system.

7.2 Future Work

7.2.1 Dominoes Improvements

Dominoes was developed to test the viability of replication for protocol processing, but there are still many open questions about improving the framework. Dominoes is divorced from any networking specific code and can be used to implement replication for other applications or system services. This is being explored currently.

The RingChannel object is used throughout this work to manage request queues. Its reference counter is updated with CAS operations. More sophisticated reference counting techniques such as sloppy counters [14]) could potentially improve the scalability of this implementation. The results of Chapter 5 show that while channel performance is sufficient for networking, it may not be for other applications.

The domain subscribe/de-subscribe operation is globally synchronous. When domains are added or removed from a channel, all data delivery stops. This prohibits dynamic domain assignment. For example, the system cannot add additional domains to a TCP connection as the throughput increases. An alternative is to add subscribe operations in a control channel queue and perform the operation when scheduling occurs. Conceptually, the idea is simple but requires heavy re-architecting of the system.

Dominoes uses a first-come, first-serve policy to assign primary domains to requests. This allows the system to quickly adapt to domains that stall. Greedy algorithms have the benefit of being easy to implement and they rely only on local information. A comprehensive study of load balancing/assignment may result in better strategies. Several other strategies exist and the greedy algorithm can be changed to operate on groups of requests or groups of domains.

7.2.2 Kernel Implementation

The userspace test framework has a number of drawbacks. Most importantly it requires additional copies that in-kernel networking implementations do not. Kernel implementations can process data directly, in the same buffer that the hardware DMA system uses to store incoming network traffic. This copy is required by the semantics of POSIX Sockets, that `libpcap` uses to read and write data. This requires three copies of the packet.

1. Kernel to userspace sockets
2. `libpcap` buffer to the Scout message
3. Scout to the ordering manager/app

Adapting the Dominoes-Scout implementation to run in the kernel will eliminate two of these copies and should significantly increase throughput at all scales. A zero-copy sockets implementation or direct NICs access could achieve similar results. This would be the first major step towards using this system with production applications.

A kernel implementation may abandon Scout and use mainstream protocol implementation such as Linux or Solaris. This has the benefit of additional development and optimization for modern networks. Production IP implementations have been tested by large teams of developers, something that is not possible for research vehicles such as Scout. The biggest drawback of using implementations like Linux is that there are complex locking systems that much be removed in order to use replication.

7.2.3 Selective Acknowledgements

RFC 2018 [49] provides selective acknowledgements for TCP. Rather than generate duplicate acknowledgements that indicate the last segment received, selective acknowledgements allow the receiver to indicate receipt of discontinuous blocks of data. This feature is designed to reduce the number of transmits when compared to the original cumulative acknowledgement strategy.

With large window sizes, selective acknowledgements reduce the number of required retransmits when a segment is lost or arrives out of order. Selective acknowledgements may reduce the need to order packets inside a single window when transmitting. This ordering currently limits scalability for the TCP send operation. Scout does not support TCP selective acknowledgements. Therefore, this function will have to be added or testing must be done with an alternative protocol implementation.

7.2.4 Latency

The Dominoes publish/subscribe mechanism increases latency because data is delivered asynchronously. Publishers cannot immediately notify subscribers that new data is available on a channel. There is also no way to preempt a domain with higher priority requests. This effect on latency tends to be less important with high-throughput systems because many, many requests are handled quickly. However, it is important for other systems that must handle small, short lived connections quickly. It is also important for communicating for distant links and for high-performance computing.

Some protocols, such as TCP, provide mechanisms for delivering urgent data out-of-band with normal messages. If the replication approach can be adapted to provide latency numbers that are competitive with existing network implementations, then this work can be used to test speedup of urgent message delivery. Replicas provide an appealing approach for urgent message delivery because they provide the ability to work independently on a separate part of a given workload.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, 1986.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Tuecke, S. O. T. Memo, L. Liming, and S. Tuecke. GridFTP: Protocol extensions to FTP for the Grid. *GWD-R (Recommendation)*, page 3, 2001.

[3] I. T. Association. Infiniband architecture specification. Technical report, 2004.

[4] S. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. *CoRR*, abs/cs/0412017, 2004.

[5] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44. ACM, 2009.

[6] A. Baumann, S. Peter, A. Schupbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. why isnt your OS? In *Proceedings of 12th Workshop on Hot Topics in Operating Systems (HotOS 09)*, 2010.

[7] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Netw.*, 7:789–798, December 1999.

[8] K. P. Birman. Replication and fault-tolerance in the Isis system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 79–86, Orcas Island, WA, 1985.

[9] K. P. Birman and R. Cooper. The Isis project: Real experience with a fault-tolerant programming system. *Operating Systems Review*, 25(2):103–107, 1991.

[10] K. P. Birman, R. Renesse, and W. Vogels. The Ensemble distributed communication system. http://simon.cs.cornell.edu/Info/Projects/Ensemble/, 1996.

[11] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *SIGCOMM 93: Conference proceedings on Communications architectures, protocols and applications*, pages 74–83, New York, NY, USA, 1993. ACM.

[12] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 2010 USENIX Symposium on Operating System Design and Implementation*, 2010.

[15] L. Brakmo, S. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM 94*, 1994.

[16] P. G. Bridges, D. Sizemore, and S. Levy. Exploiting MISD performance opportunities in multi-core systems. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA, May 2011.

[17] R. Brightwell, M. Levenhagen, A. B. Maccabe, and R. Riesen. A performance comparison of Myrinet protocol stacks. In *Proceedings of Third Linux Clusters Institute Conference on Linux Clusters*, St. Petersburg, FL, 2002.

[18] S. T. Chanson, D. W. Neufeld, and L. Liang. A bibliography on multicast and group communications. *ACM Operating Systems Review*, 23(4):20–25, 1989.

[19] J. Chu and S. Inc. Zero-copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 253–264, 1996.

[20] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications*, June 1989.

[21] B. Cohen. The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html.

[22] D. DaSilva, O. Krieger, R. W. Wisniewski, A. Waterland, D. Tam, and A. Baumann. K42: an infrastructure for operating system research. *SIGOPS Oper. Syst. Rev.*, 40(2):34–42, 2006.

[23] M. Dobrescu, N. Egi, K. Argyraki, B. gon Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[24] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP 93)*, pages 189–202, 1993.

[25] A. Dunkels. Full TCP/IP for 8-bit architectures. In *MobiSys 03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM.

[26] A. Earls. TCP offload engines finally arrive. *Storage Magazine*, 2002.

[27] L. Eggert, J. Heidemann, and J. Touch. Effects of Ensemble-TCP. *ACM Computer Communication Review*, 30(1):15–29, January 2000.

[28] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *Networking, IEEE/ACM Transactions on*, 7(4):458 –472, Aug. 1999.

[29] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1:397–413, August 1993.

[30] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP performance revisited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 70–79, 2003.

[31] D. Freimuth, E. Hu, J. Lavoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server network scalability and TCP offload. In *In Proceedings of the 2005 USENIX Annual Technical Conference*, pages 209–222, 2005.

[32] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[33] D. Giarrizzo, M. Kaiserswerth, T. Wicki, and R. C. Williamson. High-speed parallel protocol impementation. In *Proceedings of the IFIP W 6.1 /WG 6.4 1st International Workshop on Protocols For High-Speed Networks*, pages 164–180, 1989.

[34] M. A. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. Technical Report 94-28, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[35] M. A. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 288–295, Vancouver, BC, Canada, 1995.

[36] N. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[37] N. Hutchinson, L. L. Peterson, S. O'Malley, and M. Abbott. RPC in the *x*-kernel: Evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, 1989.

[38] IEEE. IEEE802.1ax: Standard for Local and Metropolitan Area Networks – Link Aggregation, 2008.

[39] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM 88*, pages 314–332, 1988.

[40] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, 1992.

[41] V. Jacobson and R. Felderman. A modest proposal to help speed up & scale up the linux networking stack. *Seminar, Linux.conf.au*, 2006.

[42] O. Krieger and M. Stumm. HFS: A flexible file system for large-scale multiprocessors. In *In Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, 1993.

[43] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building block composition. *ACM Transactions on Computer System*, 15(3):286–321, 1997.

[44] D. Lin and R. Morris. Dynamics of random early detection. *SIGCOMM Comput. Commun. Rev.*, 27:127–137, October 1997.

[45] J. Lofstead, F. Zhang, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 09)*, Washington, DC, USA, 2009. IEEE Computer Society.

[46] M. G. Luis. TCPDump/libpcap public repository @MISC. http://www.tcpdump.org/, Feb. 2011.

[47] C. Ma and K.-C. Leung. Improving TCP reordering robustness in multipath networks. In *LCN 04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 409–410, Washington, DC, USA, 2004. IEEE Computer Society.

[48] J. Martin, A. Nilsson, and I. Rhee. Delay-based congestion avoidance for TCP. *IEEE/ACM Transactions on Networking*, 11(3):356–369, 2003.

[49] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. RFC 2018, 1996.

[50] J. Mauro and R. McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[51] M. M. Michael and M. L. Scott. Fast and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, 1996.

[52] F. P. Miller, A. F. Vandome, and J. McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[53] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[54] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, 1987.

[55] A. B. Montz, D. Mosberger, S. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation*, 1994.

[56] D. Mosberger. Scout: Map library design notes. Technical Report TR97-18, Dept of Computer Science, University of Arizona, 1997.

[57] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 153–168, 1996.

[58] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[59] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[60] D. of Defense. Internet protocol. RFC 760, 1980.

[61] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Performance evaluation of the Quadrics interconnection network. *Journal of Cluster Computing*, 6(2):125–142, 2002.

[62] D. Plummer. An ethernet address resolution protocol. RFC 826, 1982.

[63] J. Postel. User datagram protocol. RFC 768, 1980.

[64] J. Postel. Transmission control protocol. RFC 793, 1981.

[65] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 367 – 378. ACM Press, New York, 2004.

[66] S. A. Rago. *UNIX System V network programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[67] K. K. Ramakrishnan and S. Floyd. A proposal to add Explicit Congestion Notification (ECN) to IP. RFC 2481, 1999.

[68] R. Raman, M. Livny, and M. H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, 1998.

[69] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, 1984.

[70] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell Systems Technical Journal*, 57(6):1905–1929, 1978.

[71] J. H. Saltzer, D. A. Reed, and D. D. Clark. End-to-end argument in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[72] W. R. Stevens. *Advanced Programming in the UNIX environment*. Addison Wesley, 1992.

[73] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, page 84. IEEE press, 2003.

[74] The Linux Foundation. TCP offload engines and the Linux kernel, 2009. http://www.linuxfoundation.org/collaborate/workgroups/networking/toe.

[75] S. Tripathi, N. Droux, T. Srinivasan, K. Belgaied, and V. Iyer. Crossbow: a vertically integrated QoS stack. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 45–54, New York, NY, USA, 2009. ACM.

[76] E. W. Ulf Lamping, Richard Sharpe. Wireshark user's guide. http://www.wireshark.org/docs/wsug_html_chunked.

[77] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *OSDI Symposium*, pages 139–152, 1994.

[78] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 25–38, New York, NY, USA, 1987. ACM.

[79] J. Viega, M. Messier, and P. Chandra. *Network security with OpenSSL*. O'Reilly Media, Inc, 2002.

[80] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. *Linux Network Architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[81] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 91–96, June 2006.

[82] H. Zou, W. Wu, Z.-H. Sun, P. DeMar, and M. Crawford. An evaluation of parallel optimization for OpenSolaris network stack. In *Proceedings of the 35th Conference on Local Computer Networks*, Denver, 2010.