Domain Computing: The Next Generation of Computing

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Yi Xue

Permission to Use

Abstract

Computers are indispensable in our daily lives. The first generation of computing started the era of human automation computing. These machine's computational resources, however, were completely centralized in local machines. With the appearance of networks, the second generation of computing significantly improved data availability and portability so that computing resources could be efficiently shared among the networks. The service-oriented third generation of computing provided functionality by breaking down applications into services, on-demand computing through utility and cloud infrastructures, as well as ubiquitous accesses from wide-spread geographical networks. Services as primary computing resources are far spread from local to worldwide. These services loosely couple applications and servers, which allows services to scale up easily with higher availability. The complexity of locating, utilizing and optimizing computational resources becomes even more challenging as these resources become more available, fault-tolerant, scalable, better performing, and spatially distributed. The critical question becomes how do applications dynamically utilize and optimize unique/duplicate/competitive resources at runtime in the most efficient and effective way without code changes, as well as providing high available, scalable, secured and easy development services. Domain computing proposes a new way to manage computational resources and applications. Domain computing dynamically manages resources within logic entities, domains, and without being bound to physical machines so that application functionality can be extended at runtime. Moreover, domain computing introduces domains as a replacement of a traditional computer in order to run applications and link different computational resources that are distributed over networks into domains so that a user can greatly improve and optimize the resource utilization at a global level. By negotiating with different layers, domain computing dynamically links different resources, shares resources and cooperates with domains at runtime so applications can more quickly adapt to dynamically changing environments and gain better performance. Also, domain computing presents a new way to develop applications which are resource stateless based. In this work, a prototype system was built and the performance of its various aspects has been examined, including network throughput, response time, variance, resource publishing and subscription, and secured communications.

Contents

List of Tables

List of Figures

List of Abbreviations

BS	Browser Server
CS	Client Server
ESB	Enterprise Service Bus
FBP	Flow-Based Programming
IaaS	Infrastructure as a Service
IoT	Internet of Things
M2M	Machine-to-Machine
MCC	Mobile Cloud Computing
MS	Micro Service
P2P	Peer-to-Peer
PaaS	Platform as a Service
PC	Personal Computer
RFID	Radio Frequency Identifier
SaaS	Software as a Service
SOA	Service-Oriented Architecture
WAN	Wide Area Network

Chapter 1

Introduction

Seven years ago, Apple introduced the first iPod Touch to the world, which was the starting point in the growth of personal mobile computing. In the years that followed, varieties of mobile devices have continued to be introduced into our lives. Some examples are smart cell phones, tablets, the transform notebook and the emergence of smart wearable devices. Not only are these new mobile devices small and portable but they also have significant processing capability. Mobile computing has become a reality because of the convergence of two technologies: the appearance of powerful portable computers and the development of fast reliable networks [52]. So these portable devices are going to continue to make mobile computing more prevalent and widely used. According to a recent report from Cisco [23], in 2013 the number of smartphones increased 77 percent, average smartphone usage grew 50 percent, global mobile data traffic grew 81 percent, and, on an average, a smart device generated 29 times more traffic than a non-smart device. In the report, it was forecasted that by 2018 there will be nearly 1.4 mobile devices per capita. Another important trend in mobile computing is that individual users are going to own more and more devices. Comparing the increase to nearly 1.4 mobile devices per person by 2018 to just 10 years ago, most people only had a desktop and maybe heavy movable laptop as the primary computation resource. Today most users not only own multiple devices like smartphones, tablets, smart TVs, smart fridges and even smart sensors in houses but much more time is being spent on them.

Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services [4]. Cloud computing provides users a new opportunity to consume services in a reliable, flexible, energy-efficient and less expensive way. The cloud computing services are not necessarily only available in a local or a small geographic areas since these services are provided over the Internet. For example, a user can access the application services in the Google data center located in the US, hosting services on the Amazon EC2 platform in Europe or the storage services on Apple cloud located in China from anywhere in the world. In fact, the simplicity, flexibility, and efficiency of services that cloud computing provides to users renders the importance of the geographic location of services irrelevant. Additionally, the computational resources used for providing such services are much more widely distributed indeed. Not only does cloud computing provide services via the Internet, but these services have also become a core IT infrastructure. IDC's forecast [35] for the period of 2013 - 2017 shows that public IT cloud services will have a compound annual growth rate (CAGR) of 23.5%, five times that of the IT industry as a whole. As a result, there is a strong possibility that more business will adapt cloud platforms over desktop machines in the near future.

Cloud computing offers an infrastructure that supports large data processing as well as enabling highly scalable computing platforms. Additionaly, cloud computing can be configured on demand to meet constant application changes in a pay-per-use mode. Furthermore, cloud computing can, because of scalability, quickly and efficiently process a large magnitude of centralized data by simply adding more computing resources. However, when data sources are distributed across multiple locations and networks, low latency is indispensable and the centralized data processing in cloud fails to achieve real-time analytics and decision making. Another issue for cloud computing is privacy. The cloud is typically built and owned by a corporation. Because of corporate ownership, many users do not want to upload and store their sensitive and personal data, like the data gathered by activity trackers, into an uncontrolled cloud. Cloud computing can't help process the data because users are more likely to retain personal and private dataon their devices. The inability of cloud computing to process this type of data has created the need for an alternative computational paradigm that will utilize more capable devices, which are geographically closer to those devices on the edge of the network than to the clouds, to perform data processing. Data can be aggregated by building local views of data flows, which can then be sent the cloud for further analysis offline - this is where fog computing [14] fits. Fog computing is a distributed computing system that extends the services that are provided by the cloud to the edge of the network. Fog computing utilizes management, networking, and data storage in order to bridge the gap between data centers and those devices on the edge of the network.

The Internet of Things [8] environment connects devices via heterogeneous networks. The significant amount of data collected from IoT devices allows us to perform pattern detection, predictive analysis, and optimization and smarter decision making in a timely manner. Data flowing in the IoT is classified into two categories: the data stream and big data. The data stream is captured continuously from the devices, while big data refers to data and knowledge that is centrally stored in cloud storage. In recent years, the IoT has experienced significant growth. By 2011, the IoT had a market value of $44 billion. Moreover, the IoT will continue to grow at a fast rate in the coming years because more and more devices posses networking capabilities and the potential to connect directly to the Internet. According to market research, the IoT and Machine-to-Machine (M2M) market will approximately $498.92 billion by 2019 and the IoT market will reach $1423.09 billion by 2020. Figure 1.1 [17] shows that by 2018 the number of active devices will surpass 30 million, compared to less than 10 million in 2011. In addition to all of these fantastic and optimistic opportunities, in order for the current IoT to reach its potential there needs to be various innovations and progress in different areas. Furthermore, IoT adoption and market growth will increase with cooperation and information-sharing between leading companies, such as Microsoft, IBM, Google, Amazon, Samsung, Cisco, Intel, ARM, and Fujitsu, as well as smaller businesses and start-ups. For example, The UK government has initiated a $5 million project to spur innovation and technological advances in the IoT [61]. Similarly, IBM in the US has plans to invest billions of dollars on IoT research and its industrial applications. So, in the

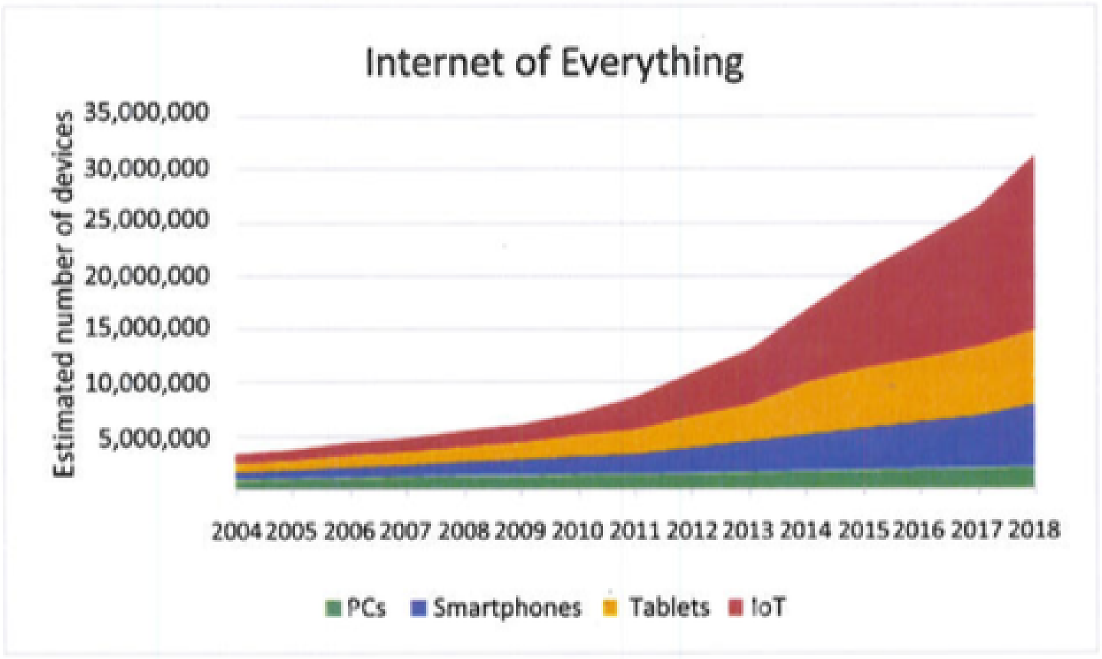

Figure 1.1: IoT Trend Forecast

near future, millions of devices will directly connect to the Internet and generate millions of computational resources, which will be distributed among ever widening networks.

1.1 Thesis Structure

- Chapter 2 - identifies and discusses the problem with current computation models
- Chapter 3 - reviews related works, including the history of computing
- Chapter 4 - proposes a domain computing architecture
- Chapter 5 - evaluates the performance of the prototype system and architecture
- Chapter 6 - summary of this research
- Chapter 7 - concludes this research and outlines areas for future investigation

Chapter 2

Problem Definition

Computational models have evolved since the invention of the first computing machine. Regardless of the generations of computing, computational resources have always been the key to applications. That is to says, computing tasks simply can't be functional or will run in a less efficiency way without the proper computational resources. For example, we can't monitor our houses without the proper sensors installed. Likewise, we will experience extremely slow random disk reads without SSDs. To some extent, computing is a war of resources: resources keep computing tasks running and computing tasks die when the resources run out. Resources can be identified in terms of their functionality. Figure 2.1 shows that there are three main categories of resources: (1) Unique, (2) Duplicate, and (3) Competitive. A unique resource offers the uniqueness of functionality in all reachable resources. A duplicate resource means that there is more than one instance of exactly the same functionality in all reachable resources. A competitive resource means that two or more resources serve the same purpose but have different performance response regarding response time and efficiency. Many computing resources are significantly increased when millions of small devices are used for IoT. As network and energy technologies become profitable, and millions small IoT devices (Figure 2.2) join the networks, millions of resources become available and reachable for applications. The increase in availability and reachability of resources for applications will continue to grow in terms of their type, number and locations. With this growth comes an increase in the complexity of locating, utilizing and optimizing computational resources, which becomes even more challenging. The current computing models miss the capability of dynamically utilizing and redistributing I/O in distributed systems in a flexible way. Additionally, current computing models are also very difficult to extend, group, and to connect functionality at runtime without changing code. Furthermore, current programming models are too complicated to be used to develop and deploy new applications, which rapidly evolve.

So, how can applications dynamically utilize and optimize the three categories of resources -unique, duplicate, and competitive - at runtime in the most efficient and effective way without code changes, yet provide a highly available, scalable, and secure and easy development service given current programming models. That is to say, with or without minimum code changes, how can we enable applications to dynamically optimize resource utilization to gain better performance when a new unique/duplicate/competitive resource is added at runtime? How do we ensure that they flexibly and seamlessly cooperate with multiple existing computing models like cloud and fog computing? How do we dynamically extend or modify an application

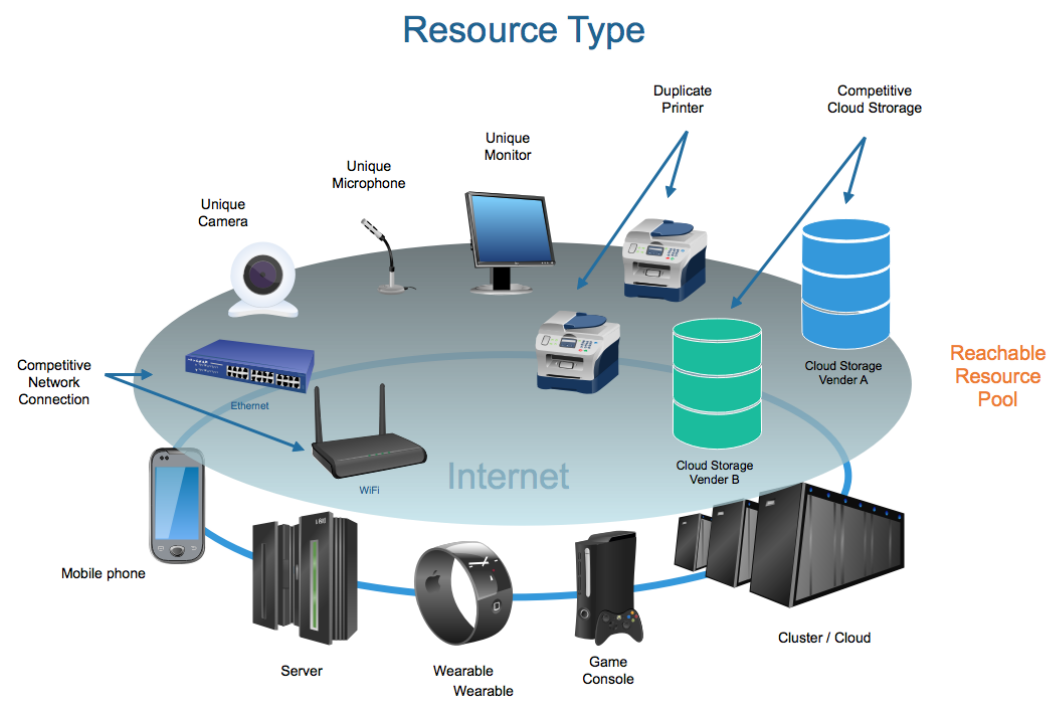

Figure 2.1: Types of Computational Resources

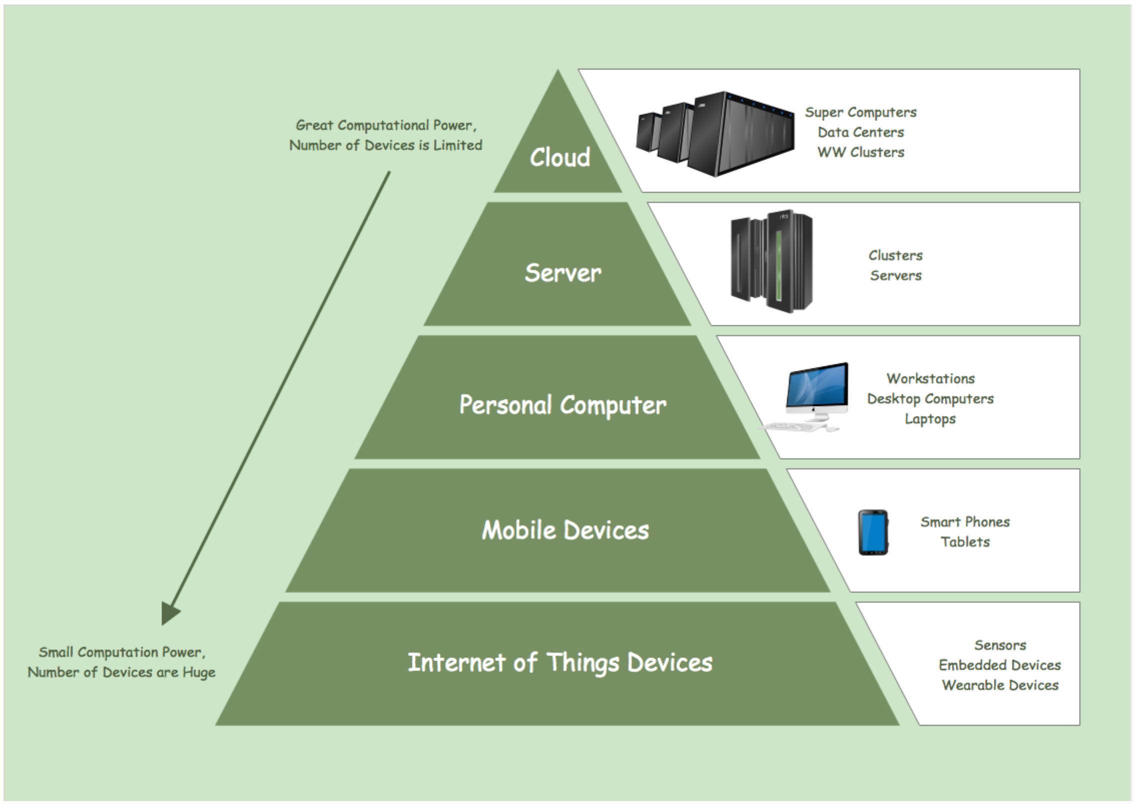

Figure 2.2: Number of devices

functionality by redistributing resources for high availability, performance, and scalability? How do we develop applications and evolve them at runtime along with improved or new technologies on the network, bandwidth, or with better/competitive resources available? How do we build applications without worrying about the coded resources that have been changed and replaced at runtime? Can we build applications that do not rely on the resources available at design time? How do we build applications that can run easily across different platforms? How do we help people, who do not have coding knowledge, to improve an application's performance or to extend its functionality? These questions, along with constantly changing network environments and available resources, present challenges that require the development of a new computing model that is flexible and dynamic.

In the following chapters, the following contributions are discussed in detail:

1. Propose a new computational model, called Domain Computing, in Chapter 4
2. Build a prototype of the Domain Computing model in Chapter 5
3. Examine the model's feasibility and effectiveness in Chapter 5 with respect to its performance of:
    - Network throughput, response time, and their variability
    - Resource publishing and subscription
    - Secured communications

Chapter 3

Related Works

A computational task involves multiple components to accomplish the task by using a computer, network, operating system, software, etc. They all can be seen as resources in general. Regardless the computational models, resources are always their fundamental part although they manage the resources in very different ways. In order to understand the difference with respect to resource management between domain computing and the previous generations of computing, it is necessary to review each generation of computing and how they utilize the resources. I/O is another critical component in the computational models. It connects the resources together and outputs the final results. Different I/O patterns are used in different scenarios. Applying an appropriate I/O subsystem to the domain computing is crucial. So in this chapter, different generations of computing and their representative models are reviewed including mainframe, PC, client/server, cluster, grid, ubiquitous, SOA, utility, cloud, MCC, fog, iot, and micro-service. Different I/O patterns are reviewed as well including UNIX pipe, message queue, stream, batch, ESB, and dataflow.

In the 1930s, when the first computers were invented, the computer as a new term had been used for almost a century [65]. The purpose of computation also has been moved far away from war services. Computers are used as control systems for a very wide variety of industrial and consumer devices. Their size is from room or factory size like a data center, portable size like a personal laptop to palm size including tablet, smartphone and even smaller wearable devices. Not only do they play an important role but also completely change the ways of our lives. File Management and Document processing software allow us to organize data/information in much more efficient and effective way. After networks, especially Internet, connect million of devices together, applications are not limited in local anymore. Services like web, mail, storage, trading information, calendar, weather forecast are widespread among the networks. We can consume these services from anywhere hence bringing about mobility and conveniency to our lives. Meanwhile, computers are changing our lives along with their decade's history, computation model relying on these computers has evolved many generations as well. In this chapter, different generations of computation model are discussed and the problems of those models are identified.

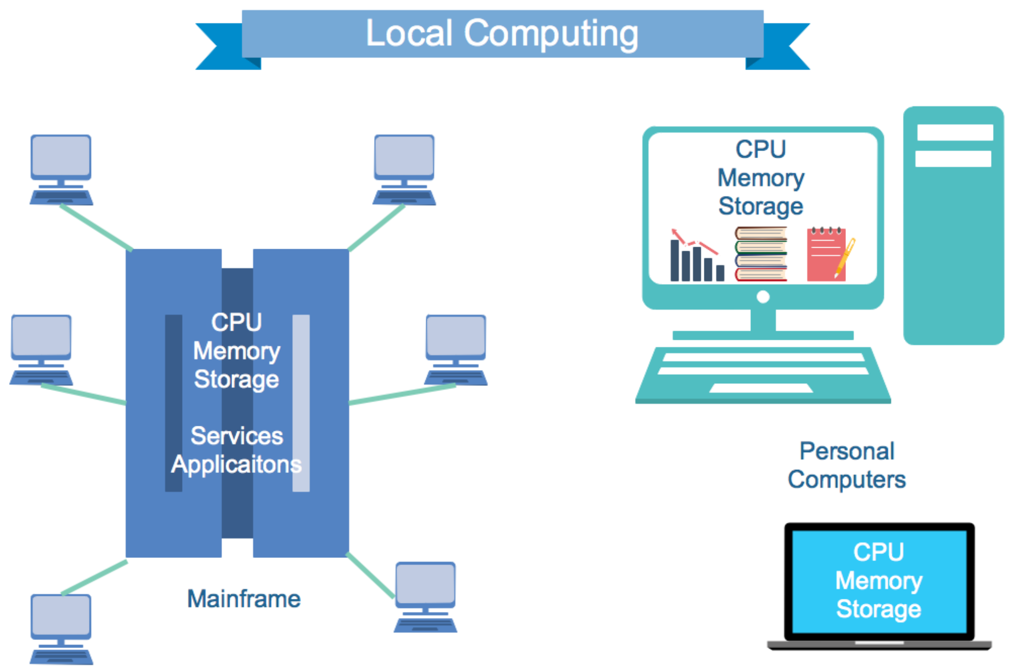

Figure 3.1: The First Generation: Local Computing

3.1 First Generation: Local Computing

In 1936, Alan Turing presented the formulation of a computational engine called the Turing machine, which was the first vision for computers and influenced the architecture of all future computational devices [65]. The world first real computer was built during World War II for message decryption purposes.

3.1.1 Mainframe

Since Turing's machine, large mainframe computers have been built. Large companies, such as IBM and DEC, developed many different types of mainframe computers [9]. Because of its complexity and cost at the time, mainframe computers were only bought and consumed by large corporations and used for critical tasks. For example, the US government has used mainframe computers for both its moon landing and military decryption of enemy messages. The price and size of mainframe computers put it out of the reach of affordability for individuals. The mainframe computer can automate much of the data processing and support business activities involving tedious calculations. Although a mainframe computer could support many users, computations were done within a single, local computer. All resources needed for such computations must be found in a local environment.

3.1.2 Personal Computer

From the late 1960s, visionaries recognized opportunities for improving human performance through individual tasks and computation [20], and a computer designed for individual use was produced. The most well-known symbols of the era of the personal computer were IBM PC and Apple II. For example, PC explored applications for individuals to automate various calculations, such as Spreadsheet. PC sales boomed in the 1980s after their price became more affordable as compared to the large and expensive mainframe computer. Corporations started buying PCs for their employees, which led to PCs finally moving into homes. Like mainframe computers, PCs accomplish all computation tasks on a local CPU and memory, meaning that the computational resources are centralized in a single local machine.

3.1.3 Summary

The first generation of computing started the era of human automation computing. However, computational resources were completely centralized in local machines, as shown in Figure 3.1, due to the various limitations of technologies at that time, including network, protocols, and the need for communications between computers.

3.2 Second Generation: Network Computing

The first generation of computing ran tasks using local resources. Although these first generation computers became more powerful in terms of higher CPU frequency, larger and faster memory and disk storage, they were missing an efficient way to exchange and share information. This limitation of first generation computers limited applications and the ability to accomplish complex tasks in efficient ways. Floppy disks could transport data but they were less effective, had less capacity and were very costly. In the 1960s, ARPANET was developed and symbolized the start of the second generation of computing, network computing. In 1965, Thomas Marill and Lawrence G. Roberts created the first wide area network (WAN). In 1969, a group of universities, including the University of California at Los Angeles/Santa Barbara, the Stanford Research Institute and the University of Utah, connected and exchanged data together despite their limited bandwidth of 50Kb/s, which looks tiny in comparison today's bandwidth. In 1973, Robert Metcalfe proposed Ethernet at Xerox PARC, which is one of the fundamental protocols for computer networking.

3.2.1 Client/Server and Browser/Server

Once computers can connect via networks, network applications are possible. Network applications run at the network application layer and above in the TCP/IP stack. Network applications provide data storage, manipulation, presentation, communication or other capabilities, which are often implemented using a client-server or peer-to-peer architecture based on application layer network protocols [11] [3] [57]. For example,

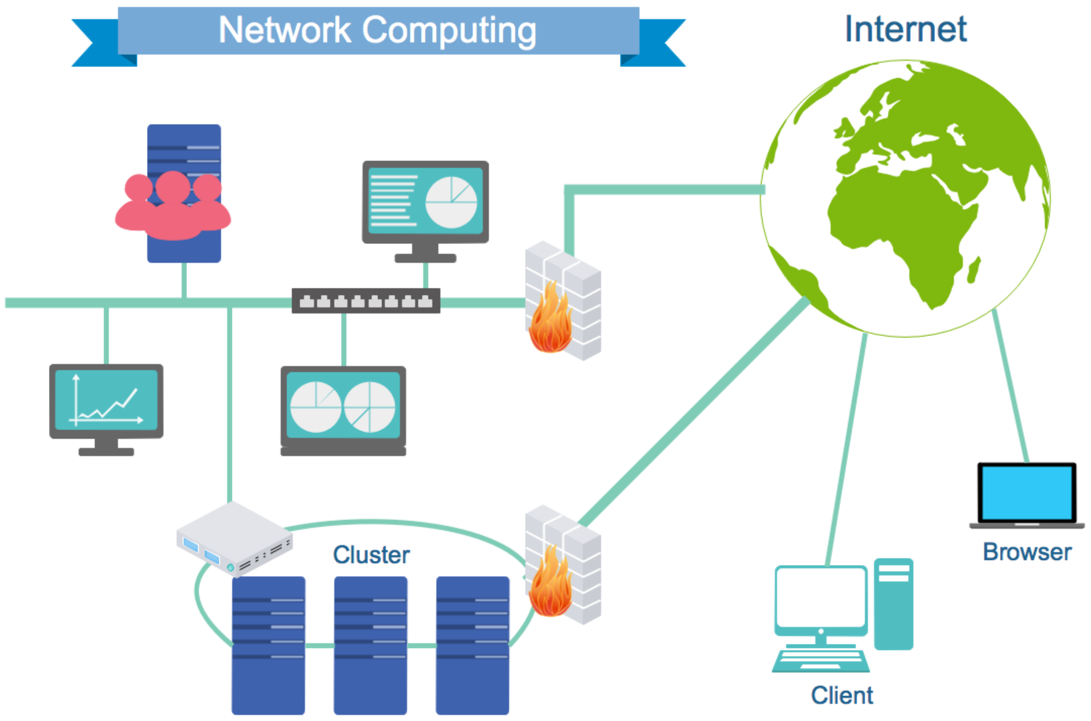

Figure 3.2: The Second Generation: Network Computing

file sharing and video broadcasting. In a network application, there are two distinct roles: server and client. Usually, the server runs on a dedicated powerful computer and offers different services, which can be accessed via networks by the client. Although the client and server can both run on the same machine in many cases, they typically run on different computers in order to achieve a better performance and balanced workload.

3.2.2 Cluster

As the computation tasks become complicated and hard to accomplish on a single machine, cluster computers were developed to solve such problems. A computer cluster consists of computers connected together via public or dedicated networks, and they can be viewed as a single system that provides services. Clusters are usually built for improving performance and availability [18] [71] [70] [24]. A computer cluster can be from the simplest form of a two-node PC system to the most complex form of a supercomputer. Although each cluster size is different, the cluster architecture can also be used to achieve very high levels of performance. The TOP 500 fastest supercomputers are often built on many clusters. A cluster relies on a centralized management approach which makes the nodes available as shared servers. A cluster differs from other methods such as peer-to-peer or grid computing, which also use many nodes, but with a far more distributed style.

3.2.3 Grid Computing

Each node in a cluster is set to perform the same task which is controlled and scheduled by software. For example, you can access and retrieve the data that is queried from any node in a database cluster. Each node does the same job to locate the data, grab the data and return the data to you no matter which node you connect to. Instead, grid computing is a distributed system that contains the collection of computer resources from multiple locations to accomplish a common goal [31] [10] [69] [16] [69]. Each node in a grid performs different tasks/applications. The computers and clusters in a grid are loosely coupled and connected via heterogeneous networks. The size of grids are quite varied and are geographically dispersed. Some of the grids can also be very large, spanning across corporations, cities and even states and provinces. Grids are often constructed as general-purpose systems via middleware software libraries. The middleware is used to divide tasks into pieces and distribute them on several computers or clusters. Coordinating different applications on grids is complex due to distributed computing resources. Grids are generally used for computationally intensive scientific, mathematical, and academic problems, and/or economic forecasting and data processing in support of e-commerce and Web services in commercial corporations.

3.2.4 Ubiquitous/Pervasive Computing

Unlike the server computing models discussed above, ubiquitous computing focuses on the user side. In the 1980s, Mark Weiser [64] articulated a new model whereby computing is anytime and everywhere. Weiser envisioned individuals owning and interacting with multiple devices possessing with different sizes and capabilities. Similar to personal desktop computing, ubiquitous computing can use any device, in any location, and in any format. The computers are not in a single form, and they could be laptops, mobile devices and even home appliances like TVs or fridges. The early forms of ubiquitous computing, such as PDAs (Personal Digital Assistant), were simply used to synchronize calendar and contact information from PCs via wired cables so that people could continue to process data and information when they were mobile [63]. As network technologies, especially wireless networks, improved and become available in more places, Synchronization not only became more efficient but many other applications were also developed, which significantly impact our lives. After the Internet had been widely adopted, portable devices were able to engage remote services around the world when users moved and traveled. Also, the processing power of mobile devices can now compete with PCs and the applications used to run on PCs are running on these mobile devices. Ubiquitous computing engages various technologies including networks, software middlewares, artificial intelligence, sensors, user interfaces, Internet, and protocols so as to allow a better and virtually seamless user experience [63] [39] [2] [64]. Sometimes, ubiquitous computing is also called pervasive computing. Pervasive computing engages smart software layers to detect changing computing environment, analyze situations through some AI components, and to deliver consistent and seamless services - although context and user-relevant data detection are complex [53] [22] [51]. Users are not concerned about how to adjust the software to adapt to new environments. Nevertheless, ubiquitous and pervasive computing share a vision that inexpensive and small devices with network connections are distributed at all scales throughout.

3.2.5 Summary

Because of networks, the second generation computing significantly improves data availability and portability so that computing resources can be efficiently shared among the networks. A lot of network applications have changed the way we read, the way we shop, and the way we live. Networks bring opportunities to consume computing resources around the world and, again, they are not limited to a local environment. The network applications allocate, utilize, and collaborate wider distributed network resources to provide a single and compound service, like file service, mail service, and web service for users to consume.

3.3 Third Generation: Service-Based Computing

While first generation computing only utilizes computing resources locally, the second generation computing can consume the resources distributed around the world via networks. Although network computing significantly improves data availability and portability, it primarily builds a single and compound service as a computing resource, which simplifies application development. One classic example is a web server. At that time, web servers provided a single front web page that integrated all news, stock, weather or the other information together. The problem of a single front page was that if we only wanted stock or weather updates, we had to download the entire web page, do our own page parsing, and then grab the information we needed. Third generation computing, also called service-oriented computing, tries to solve this problem. The basic idea behind this computing model is to break a gigantic application into smaller services, so that users can consume the services on demand. On-demand service consumption also pushes application development into consuming remote services instead of obtaining all the resources in local and run tasks locally. Services can be discovered from networks via Service-Oriented Architecture (SOA). This not only allows you to push complex tasks that run in remote but it also speeds up the application development cycle by focusing on message processing and result presentation.

3.3.1 Service-Oriented Architecture (SOA)

As time changes, services become fundamental components of applications, especially in enterprise environments. SOA is a design paradigm used for software development and deployment [46] [28] [38] [44] [12]. An application can be composed of single or multiple local and remote services via communication protocols and networks. A service can be seen as an independent unit of functionality and can be implemented and run on different programming languages, platforms and locations. Because of this loose coupling characteristic, services can be developed independently from different vendors, products, and technologies, which accelerates development and deployment cycles. A service can engage multiple services and combine them in an ad hoc

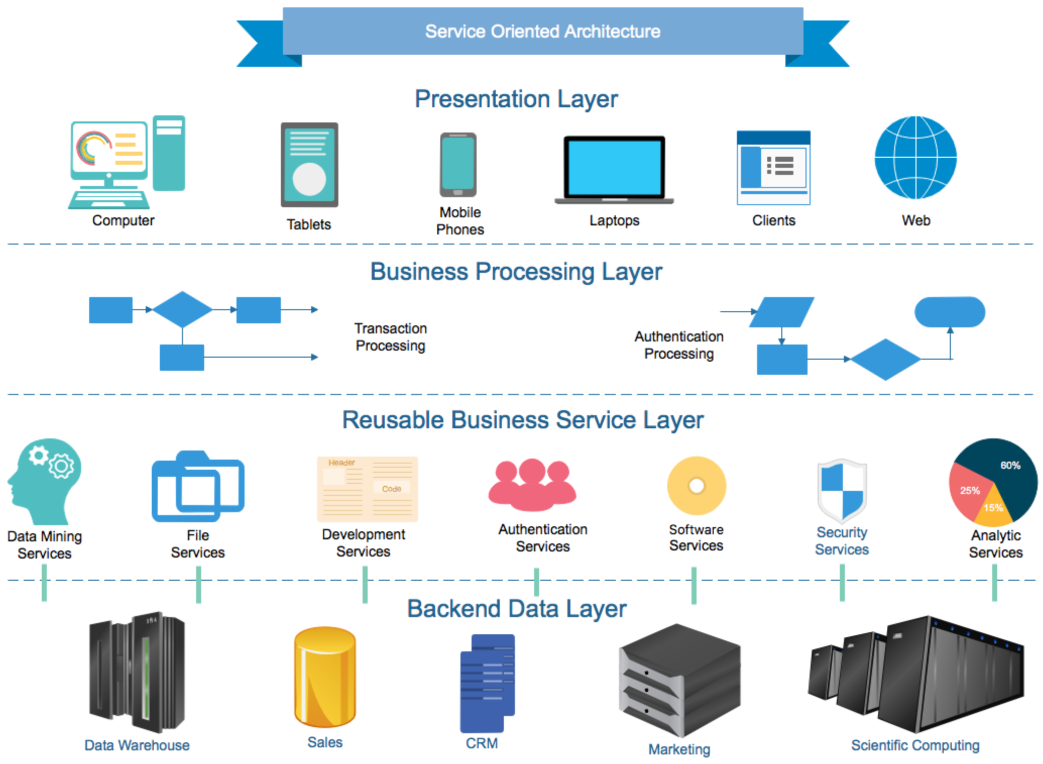

Figure 3.3: SOA Architecture

manner. SOA makes it easier to distribute services over networks, cooperate and collaborate with the other services so that it can build applications quickly from available services. For example, an online shopping website may utilize the services from its shipping partners and allow the customers to track their deliveries. This same shopping website may also utilize payment services from VISA or Mastercard organizations to speed up customer payment processes. However, all the underlying services are transparent to the customers. Customers can complete shopping via the website as a single integrated service.

In SOA, there are three distinct roles required: (1) service provider, (2) service broker, and (3) service consumer. A service provider is an entity who provides actual services and register these services to a service broker with description metadata, which indicate the characteristics of the service, data format, and how the service is called and how to parse messages. A service consumer asks a service broker to list all its registered and available services then it communicates to the service providers directly where the chosen services are running.

3.3.2 Utility computing

Mainframe computers used to dominate in the business world, particularly for large organizations. Mainframe computers offer strong computing power and storage to support critical business tasks, including business transactions and databases. However, mainframe computers are not always used nowadays because better x86 servers have changed the market. As Intel and AMD processors perform much better, PC architecture servers become even more competitive, not only on performance but especially also on price. A mainframe computer can easily cost more than a millions dollars, which can only be afforded by a few large corporations. As cost effective PC servers came into the market, they become very competitive for medium and small business. In fact, more PC servers are deployed in worldwide data centers.

Another problem in the business world is peak hours. For example, Christmas time is a busy season for promotions. Shopping volumes are increased beginning in early December, and this volume can be 3-5 times more than the normal volume for the other months of the year. Handling a large amount of requests is never easy in a short time . The companies have to build a "monster" system to support the transactions during this time. Unfortunately, the cost efficiency of such a business model is very low because the extra processing power is significant, and energy and maintenance cost are completely wasted during the non-holiday seasons, which are the majority of a year.

If there is a way to maximize the efficient use of computing resources and minimize associated costs, that would be vital for businesses. Businesses can add and remove computing resources on demand. Utility computing is a service-oriented computing model. Utility computing packages computing resources, such as computation, storage, and services, as a measurable service and charges based on usage [48] [45] [50] [15]. This is just like paying electricity, water and gas bills based on how much we use. The utility model reduces the initial cost to acquire computer resources and add more as needed later. Utility computing usually engages virtualization technology so that storage or computing power. can be constructed and rented dynamically or when needed.

3.3.3 Cloud Computing

Utility computing solves the on-demand computing problem and allows computations run in a pay-as-you-go fashion. Utility computing can dynamically add new computing resources like computers, networks, and storage as needed. Hardware virtualization allows the seamless extension of current computing capability by adding virtual machines, or VMs, as one of core technologies adapted in utility computing. Some of the utility computing providers also integrate software middlewares and development environments into the VMs, which allows developers to concentrate on application logic instead of building platforms and infrastructures. Nevertheless, utility computing provides both platform and infrastructure services in a pay-as-you-go way. However, utility computing lacks yet another important functionality which is software as a service. For example, database services or big data mining services. In recent years, cloud computing has become popular. The IDC forecast for 2013 - 2017 shows that public IT cloud services will have a compound annual growth rate (CAGR) of 23.5%, or five times that of the IT industry as a whole [35]. This growth presupposes that more business will move to a cloud platform in the near future.

Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services [4] [42] [19] [72] [34] [5] [32]. Cloud

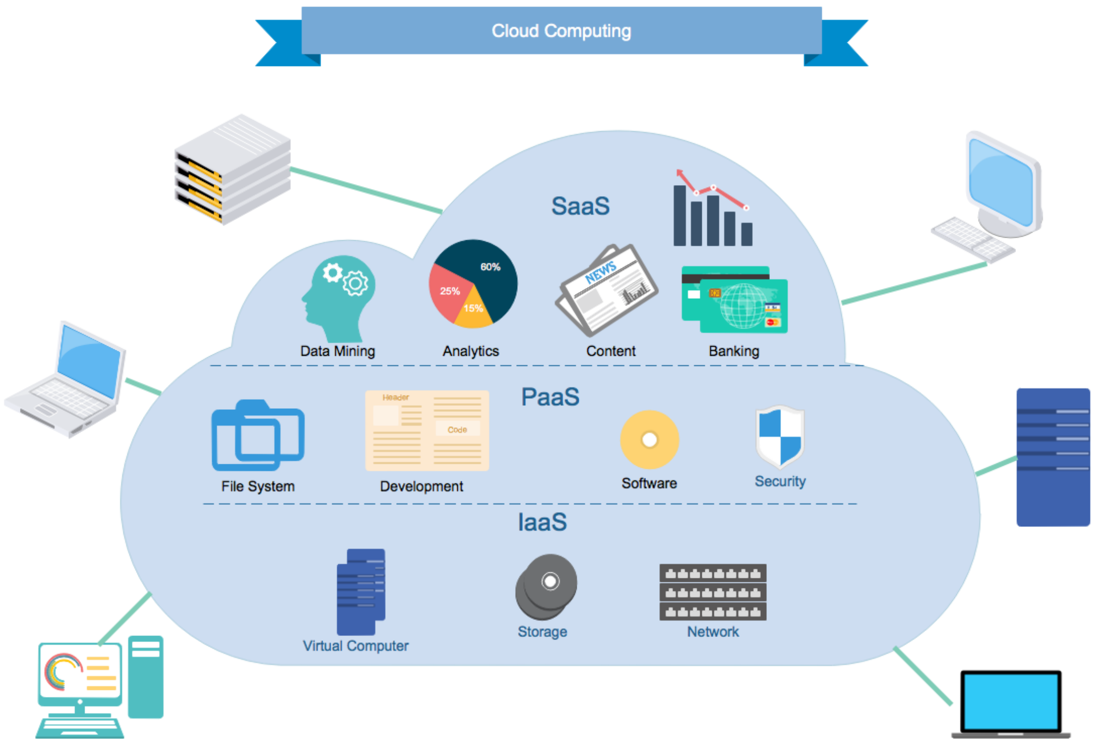

Figure 3.4: Cloud Computing

computing can enable ubiquitous, on-demand access to computing resources like computer networks, servers, storage, applications, and services, which can be rapidly deployed with the least amount of management effort. Furthermore, cloud computing avoids a company's upfront infrastructure costs like building servers or clusters so that the company can focus on its core businesses. Without the burdens of infrastructure, a business will have its applications up and running faster, with easier management and lower maintenance costs, and the flexibility to adjust computing resources according to workloads or new business demands. Cloud computing also pushes business models to service integration.

In cloud computing, service types are clearly cataloged. Cloud computing offers three major types of services: (1) IaaS, (2) PaaS and (3)SaaS. First, IaaS (Infrastructure as a Service) providers offer computing infrastructure, virtual machines, and other resources, as a service to users. The virtual machine abstracts the user from the details of infrastructure like physical computing resources, location, data partitioning, scaling, security, and backup. IaaS also offers additional resources such as a virtual-machine disk-image library, raw block or file storage, load balancers, IP addresses, firewalls, and virtual local area networks and software bundles. Cloud users install operating system images and the associated application software on the virtual machines. Although the OS and software are still needed to be maintained, the physical computer maintenance cost is removed.

Second, PaaS (Platform as a service) offers application developers a development environment. Necessary

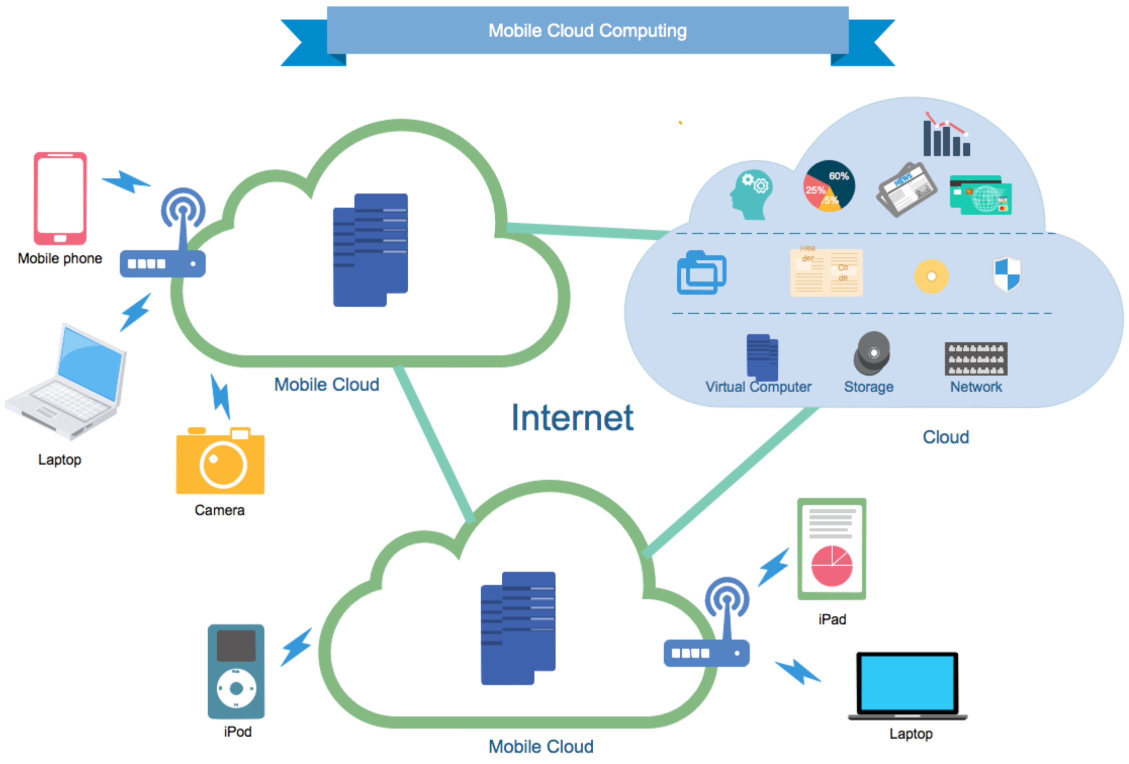

Figure 3.5: Mobile Cloud Computing

toolkit and standards for development are provided. The service delivers a computing platform, typically including operating system, programming language execution environment, database, and a web server. Some vendors even provide a scalable infrastructure to scale up. So, application developers can develop and run their software without infrastructure worries.

Third, SaaS (Software as a service) allows users to gain access to application software. Application software is running somewhere in the cloud, which is under management by the cloud infrastructure and platform. For cloud users, this eliminates the need for installation, execution, maintenance, availability, and scalability, and load-balancing. Cloud users can access the software as a single access point from cloud clients without knowing where it is running. SaaS is usually charged as a pay-per-use basis or a subscription fee.

When we benefit from cloud computing's simplicity, flexibility, and efficiency, we do not care about the locations where the services run. The services can be located in the data center in your city or another end of the world because a faster Internet makes it possible. So the computational resources used for providing services are much more widely distributed. Cloud computing does not only make services available over the Internet but also allows companies to build a global IT infrastructure.

3.3.4 Mobile Cloud Computing

When people start owning multiple devices, such as tablets and smartphones, a lot of daily-used applications have been pushed from the traditional desktop and laptop computers to those mobile devices. Also, wireless communication technologies like WIFI and cellular networks have become sophisticated with higher bandwidth, meaning that it is possible to access services on the Internet from such small devices. Cloud services used to be consumed by the relatively powerful personal computers, but are now being moved to smart devices. Mobile Cloud Computing (MCC) connects mobile devices with cloud services via wireless networks to explore and utilize more computational resources [26] [30] [47] [60]. Mobile devices' rich user experience, combined with the cloud's rich services and powerful processing capability, makes mobile applications more toward unrestricted functionality, storage, and mobility. Clouds provide mobile device services at anywhere and at anytime through the Internet, regardless of heterogeneous environments and platforms. Resource-constrained mobile devices can push intensive computational tasks to clouds and then simply present their results on screens.

3.3.5 Fog Computing

Cloud computing enables highly scalable and available computing platforms that can be configured to meet constant application changes in a pay-as-you-go mode. So, the large volume of centralized data volume can be processed in a fast and efficient way in scalable clouds. However, the centralized data processing in the cloud fails to achieve real-time analytic decision making when data sources are available across multiple locations and networks with low latency are unavailable.

Privacy is another security issue for cloud computing. The cloud is, generally, built and owned by large IT companies and, in many cases, that makes users not want to upload and store their sensitive and personal data into an uncontrolled cloud environment. For example, activity tracking data. In those cases where users do not upload their data, cloud computing can not help process it.

These issues create the need for the development of an alternative paradigm that is capable of bringing the computation to those capable devices that are geographically closer to those devices on the edge of the network than they are to clouds, building local views of data flows and aggregating data to be sent to the cloud for further offline analysis. This is where fog computing fits. Fog computing is a distributed computing paradigm that fundamentally extends the services provided by the cloud to the edge of the network [14] [56] [13] [59] [1]. Fog computing can function as an intermediate layer to accomplish tasks without sending to backend clouds. Furthermore, fog computing utilizes management, networking, and storage to bridge data centers and edge devices. Fog computing helps reduce network traffics, which is particularly useful when the network bandwidth is restricted and a fast response time is required. While a lot of simple tasks and decisions can be processed quickly in fogs without unnecessary delay in clouds, it provides a better control of privacy and optimization by utilizing all of the local computational resources.

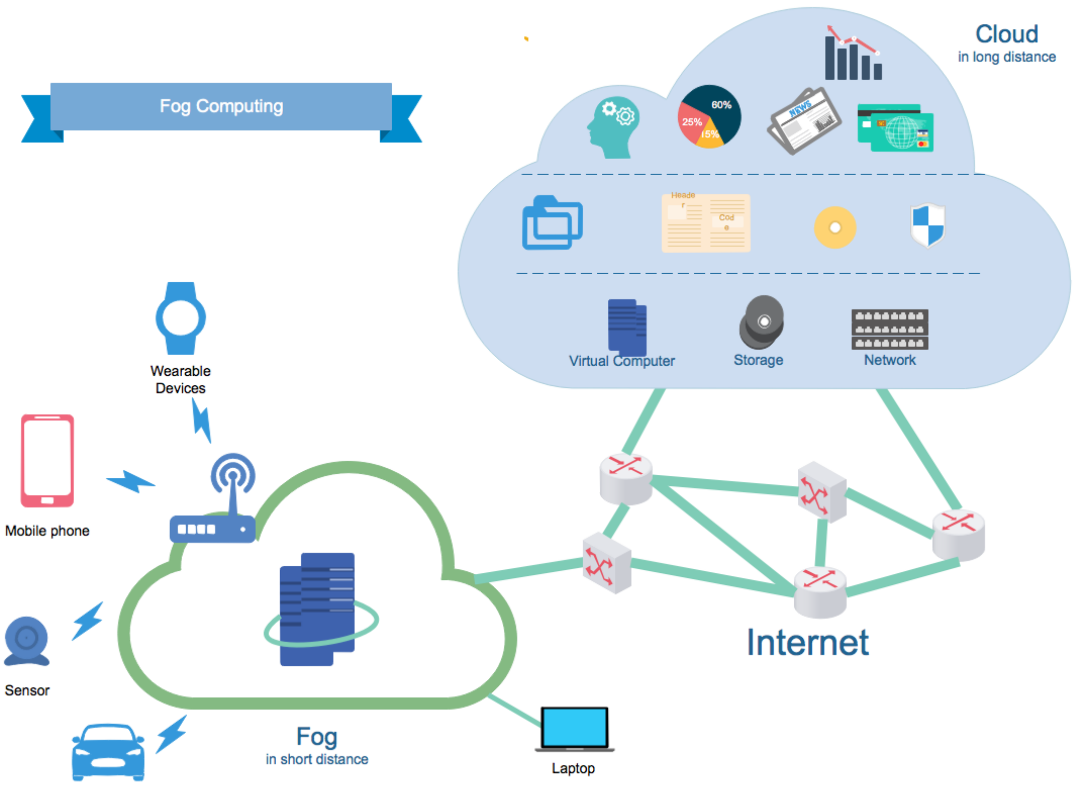

Figure 3.6: Fog Computing

3.3.6 IoT

Although networks have been developed for decades, and millions computers are connected to Internet already, there are still many devices that do not have connections due to limitations in functionality, energy, and size. In 1999, Kevin Ashton raised an idea of a global network of objects connected to RFID (Radio-Frequency Identification.) This global networkof RFID-connected objects is considered the original form of IoT (Internet of Things), although this concept has been far extended. The IoT in recent years has grown very fast. From 2003 to 2010, the number of devices connected to the Internet increased by 25 times [29]. The IoT interconnects heterogeneous devices together from different manufacturers with diverse functionalities [37] [62] [7] [33] [58] [43] [40]. The IoT expects good connection coverage to different kinds of devices and services. The connectivity of the IoT even extends the Internet to those devices without Ethernet interfaces like sensors and wearable devices. Additionally, this connectivity enables advanced applications and smart cities, smart homes, and smart health. For example, people can monitor their health via their smart watches or wearable monitors. The history health data is analyzed through remote data mining services in clouds. The IoT not only utilizes machine-to-machine (M2M) communications, heterogeneous networks, protocols, and domains but also engages an enormous number of objects. The communicating objects are shifting from computer systems to sensors, actuators and smart embedded devices in vehicles, buildings, and public facilities, which become principal objects in the IoT context. The physical objects can be represented as sensors that convert

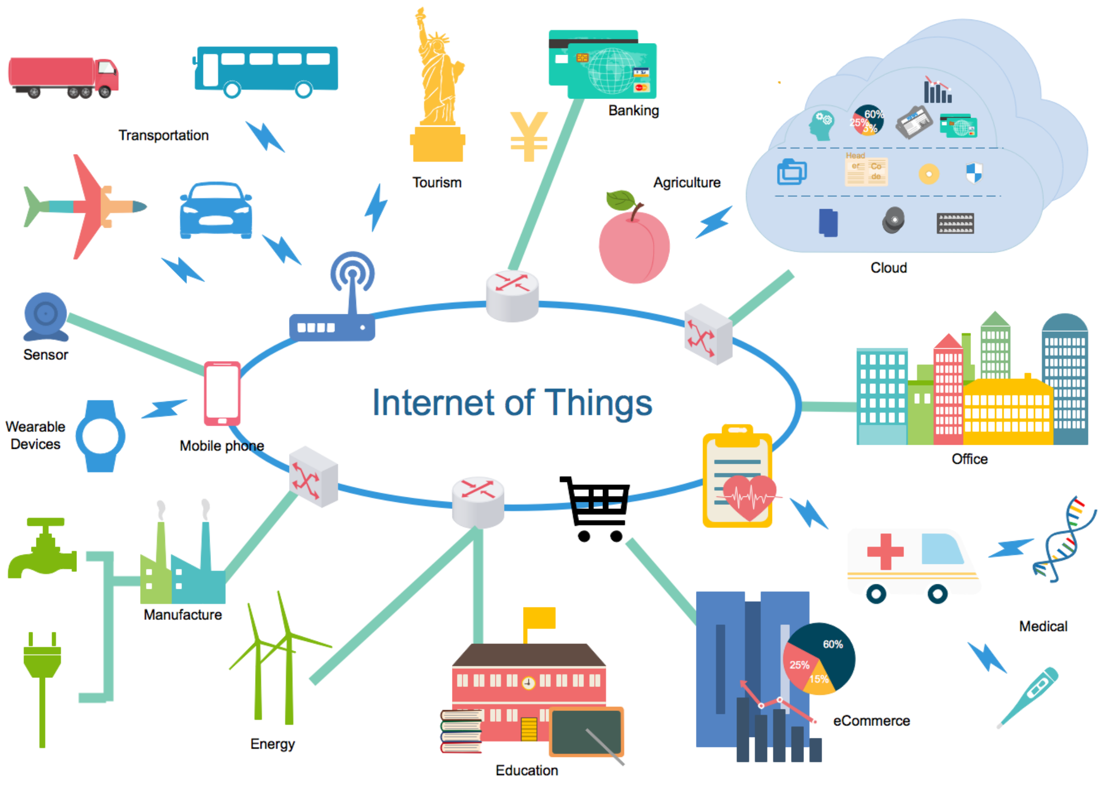

Figure 3.7: Internet of Things

physical world dimensions to cyber worlds. Because of this capability of mapping/conversion, the term of 'Things' can now refer to various devices such as heart monitoring implants, biochips on farm animals, pollution monitors in city waters, automobiles with built-in sensors, safety analysis devices for environmental and food monitoring or even field operation devices that assist police in search and rescue operations. The IoT offers many possibilities to enable advanced applications that will result in improved efficiency, accuracy, and economic benefits. With the expansion of enormous sensors, the IoT generates massive amounts of data from diverse devices and locations, which challenges the way we index, store, and process data. Another big challenge is managing a large volume and variety of computational resources that are connected to heterogeneous networks.

3.3.7 Micro-Service

Although Peter Rodgers introduced "Micro-Web-Services" in 2005 [49], it was not until 2011 that micro-service was first described in a software workshop [27]. Micro-service was then explored as an extension of SOA. Micro-service, seen as a fine-grained SOA architecture, can be used to build distributed systems in a fast, flexible and scalable way. The principle of micro-service is to build an atom service that provides a fine-grained form of a single complete functionality, so that it is easy to test, deploy, and replace. Because micro-service is a symmetrical architecture instead of hierarchical one, it is elastic, composable, and used for

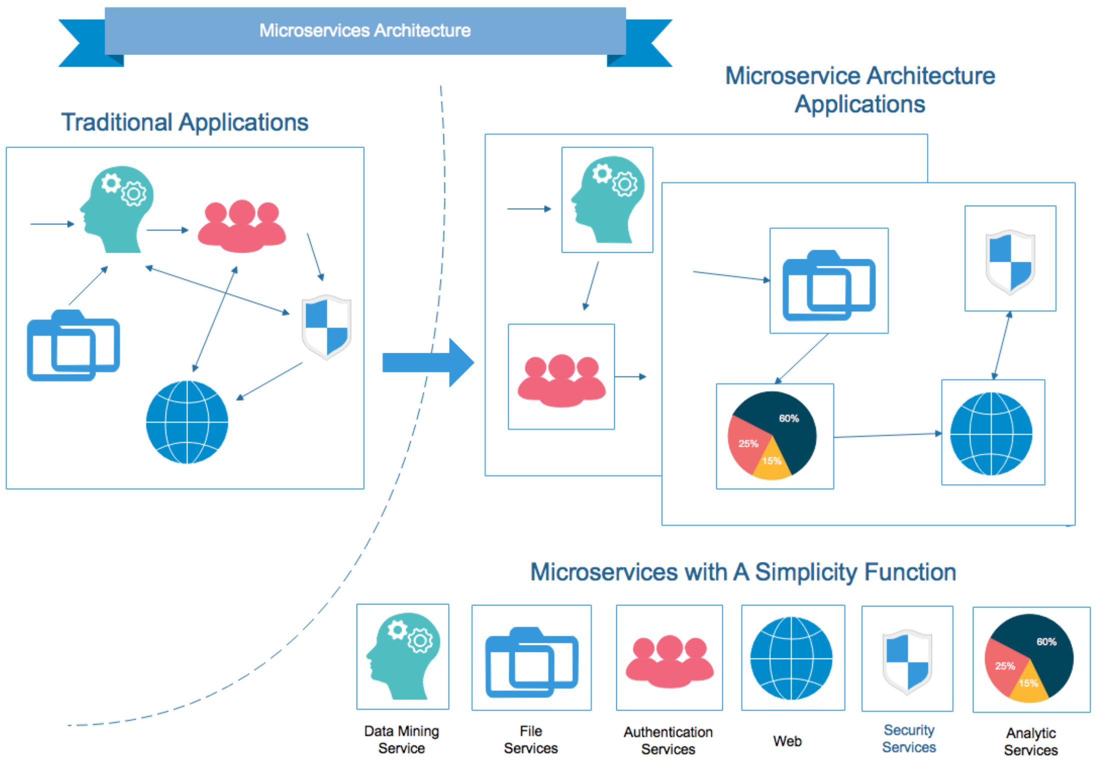

Figure 3.8: Microservice Architecture

building modular structure applications. The benefit of breaking down a big compound service into smaller services is that the cohesion is enhanced while the coupling is decreased. This benefit not only gives the flexibility to scale a system up and down at any time, but also to accelerate the cycles that we implement and deploy applications by continuous refactoring. As extended from SOA, micro-services have some similar features with SOA. Both services communicate with each other over a network and can be implemented in different programming languages and platforms as needed. However, micro-service differs slightly from SOA. Both are service-oriented architectures but with different goals. SOA focuses more on reusing and integrating various existing services, whereas micro-service focuses on building an incrementally evolving system, which means that at the beginning of building a system we do not have to build everything but, as time progresses, we can easily add, improve, and redeploy functionality.

3.3.8 Summary

The service-oriented third generation computing provides functionality by breaking down applications into services, on-demand computing through utility, and cloud infrastructures and ubiquitous accesses from wide-spread geographical networks. As figure 3.9 shows, services as primary computing resources are far spread from local to worldwide. Service-oriented third generation computing loosely couple applications and servers, which make services easy to scale up with higher availability. As network technologies become prosperous,

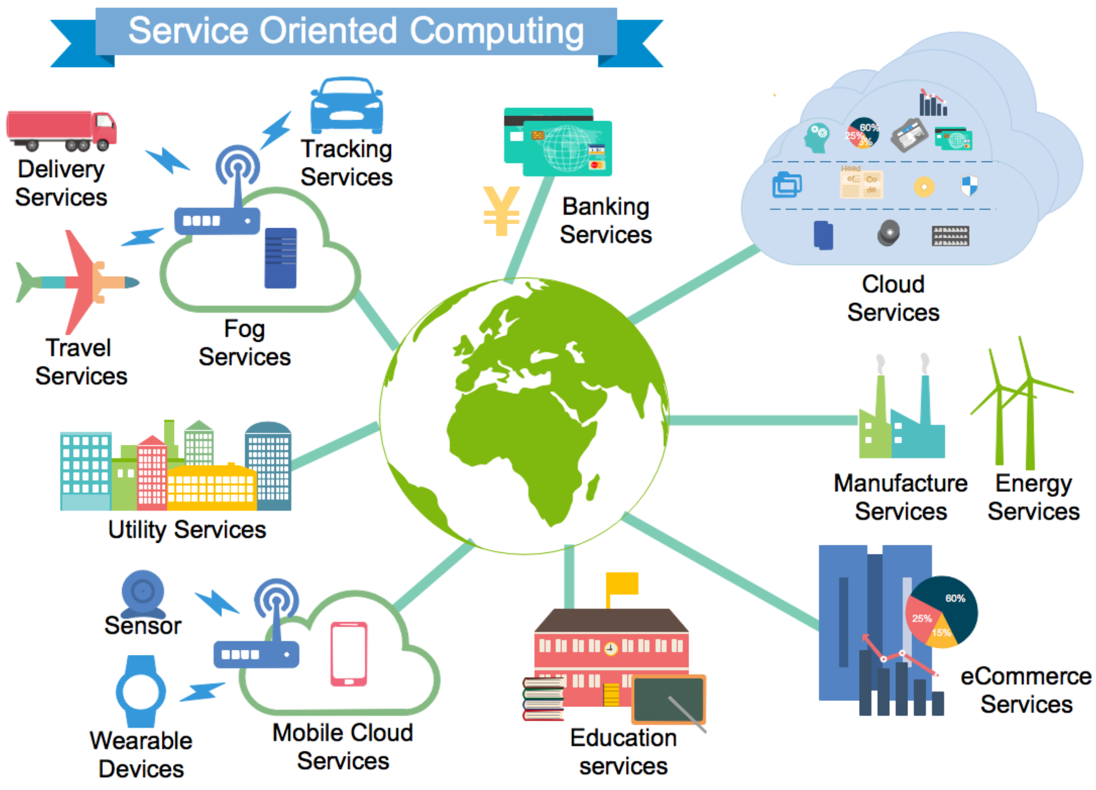

Figure 3.9: The Third Generation: Service-Oriented Computing

and millions of small IoT devices connect to networks, a variety of services will continue to bloom in terms of their type, number, and location. The complexity of locating, utilizing, and optimizing computational resources becomes even more challenging.

3.4 Input/Output Patterns

3.4.1 UNIX Pipeline

The concept of pipelines was proposed by Douglas McIlroy during the UNIX development at Bell Labs [41]. The UNIX pipeline is named by analogy to a physical pipeline because of conceptual similarity. In Unix systems, pipelines are widely used to connect processes' inputs and outputs. Unix allows the grouping of processes by connecting them together with a special commands so that the output of each process directly feeds into the next one as input. For example, this UNIX command line:

$$ls - al \quad | \quad grep \quad key$$

filters out the file names containing the key in the current directory where the command runs. Although UNIX allows pipelines to establish connections between processes, only standard streams can flow in the pipes including stdin, stdout, and stderr and they can only build in a local system.

3.4.2 Message Queue

As a different paradigm to communicate between processes or machines is message queue which provides an asynchronous communication paradigm. This paradigm refers to when a sender sends a message, it is not bound to a receiver at the same time. The receiver can receive the message at another time. This loose coupling between senders and receivers gives a more flexible and efficient way to communicate. The messages sent out by a sender are put into a queue before a recipient receives and removes it from the queue. The sender's thread or process runs even more efficiently because it is not blocked to handle the next instructions. Message queues have been widely used in a variety of systems, especially heavily used in communication systems and the communication industry. In a heterogeneous system environment, the message queue is a better way to communicate by removing differences among systems and platforms. Although the message queue has advantages on asynchronous communication, its effectiveness is limited by the capacity of the queue.

3.4.3 Stream and Batch Data Processing

From a data processing perspective, there are two main methods to handle data: streaming and batching. The data can be a message, raw binary or have a finite length. A stream is a sequence of data elements that are continuously available over time. A batch job is a series of data elements where each is on a set of inputs, rather than a single or infinite input. It takes a series of steps to process the input set. Streams are processed differently from batch jobs. The system can not handle streaming data as a whole all at once because the data continues flowing into the system in a potentially unlimited stream. Streaming data can be converted, transformed or filtered as new outputs or can become inputs of the next processes. IoT devices typically produce streaming data because millions of sensors continuously generate a massive amount of data every second. Batch processing is very different than streaming data. Unlike streaming data, batch data will not be processed until an entire set of data has been received because the size of inputs can be calculated and managed. For example, bank transaction systems typically shift a large batch of data for processing after business hours. In general, batch processing has a better efficiency and flexibility of data processing than stream processing but, in many cases, stream data is the nature of data that is produced by sensors.

3.4.4 Enterprise Service Bus

In service-oriented architecture (SOA), services are distributed not only over different types of networks but also heterogeneous systems. Different types and generations of networks use variety of hardware and software protocols to communicate, and heterogeneous systems have their own software stacks, metadata, and protocols in order to exchange information. Hybrid systems run together to provide business support due to reasons like legacy, vendors, and functionality, particularly in an enterprise production environments. An enterprise service bus (ESB) is a software architecture model that is used for designing and implementing communication between heterogeneous and complex applications and systems [21] [54]. ESB connects various services and components over networks, and allows them exchange data by using a standard, structured, and general purpose concept for describing implementation of loosely coupled systems because of its similarity to computer bus concept. That is to say, ESB promotes anagile and flexible communication infrastructure for enterprises.

3.4.5 Dataflow

Dataflow is a term used in computing that focuses on stages instead of actors so that dataflow oriented programming becomes a data-centric programming model [36] [6]. Dataflow models a program as a directed graph of the data flowing between operations. Data is exchanged and managed through connections and flow policies defined by users. In a traditional program data is treated as a series of instructions that occur in a specific order and then the program executes these instructions one by one. A dataflow program, however, focuses on data movement and the connections between the input and output operations. Operations are like black boxes and produce results when all of the necessary inputs are ready. The boxes can be reconnected by flows to form different applications without the internal changes.

There are two major programming categories in a dataflow programming paradigm: reactive programming and flow-based programming. Reactive programming refers to data orientation and change propagation. Any change applied to the source data will be propagated along the flows automatically once a data flow is established. For example, z in the expression z = x + y will be automatically updated if any value of x and y is changed. A classic application is Spreadsheet, and if one cell is changed then all of the other cells that reply to this cell are changed too.

Flow-based programming (FBP) is the second programming category in the dataflow programming paradigm. FBP defines an application not as a single, sequential process. Rather, FBP defines an application as asynchronous processes connecting and communicating by data streams to form a network as an application. FBP focuses on the data generated, flow directions, and their transformations to the desired output. In FBP, the network is defined externally to the processes. Different processes in the network execute the same or different functions, depending on the computing contexts. The data belongs to either a process or a transit status between two processes at any given time. Because it treats the processes as a black box, FBP's loose coupling of the processes means that they can be distributed over networks via proper protocols. Dataflow works well in parallel, large, and decentralized systems. Dataflow can be used for concurrent and stream processing because of its data-centric nature. Since SOA is a loose coupling architecture, dataflow fits it well. Examples of cloud dataflow platforms are Microsoft Azure Data Factory, Google Cloud Dataflow, and Apache NiFi.

3.4.6 Summary of Related Works

As shown in Table 3.1, computational resources have been changed dramatically in terms of variety and distribution along the previous generations of computing. The first generation of local computing provided and linked the resource locally, which greatly limited the possibility of applications. The second generation of network computing significantly improved data availability and portability so that computing resources could be efficiently shared among the networks. The second generation of network computing contributed a lot to the resource's distribution but not much on the variety. By breaking down traditional applications, the third generation of service-oriented computing took the advantages of network computing including the utilization of different networks, and provision of numerous services, and functionality precision. Furthermore, the third generation of service oriented computing loosely coupled applications and servers, which make services easier to scale up with higher availability. However, all three generations of computing lack the ability to connect to the selected resources and form a virtual computer, or a domain. This ability to connect resources to form a virtual computer becomes a necessity, particularly in the environment where a number of the available resources becomes larger and larger thus making them disperse worldwide, where they cross heterogeneous platforms and networks. Also, current I/O models are unable to connect to such resources together in a efficient and effective way. Although some I/O patterns, like UNIX pipe, and ESB and dataflow, can connect to some types of resources, none can connect resources to form resource networks across systems and platforms with access control within a functional and defined domain level instead of an application level. A new generation of domain computing is needed because of these missing functionalities in the current generations of computing. A new generation of domain computing is discussed in the next chapter.

Local Computing	Mainframe, PC	1) provides and links the resource completely in local 2) greatly limits possibilities of applications
Network Computing	CS, BS, Cluster, Grid Computing, Ubiquitous Computing	1) significantly improves data availability and portability 2) resources can be efficiently shared 3) contributes a lot on the resource's distribution but not much on the variety
Service-Based Computing	SOA, Utility Computing, Cloud Computing, Fog Computing, MCC, IoT, Micro-Service	1) breaks down traditional applications and provides numerous services with precise functionality 2) create a variety of resources via heterogeneous networks with a higher bandwidth and faster response time 3) loosely couples applications and servers which enables services to easily scale up with higher availability 4) still lacks the ability to connect to selected resources over networks and form a virtual computer
I/O Patterns	UNIX pipeline, Message Queue, Stream, Batch, ESB, Dataflow	1) connect to some types of resources in a certain way with regards to scenarios 2) none of them can connect resources to form resource networks across systems and platforms with access control within a unit

Table 3.1: Summary of Related Works

Chapter 4
Domain Computing and Architecture

Domain computing proposes a new way to manage computational resources and applications (Figure 4.1.) Domain computing dynamically manages resources within logic entities, domains, and without being bound to physical machines so that application functionality can be extended at runtime. The domains can be created for different purposes, and users can have as many domains as needed. The computational resources in domains can be added, removed, and updated locally and remotely, even from existing computing models like the cloud, fog, and IoT devices. A resource in the domains can be seen as a function with self-executing capability. Furthermore, a resource in the domains, unlike traditional resources, is not a static resource. Since it can be executed, this resource can be easily redistributed to various platforms and systems. Also, the data link between resources can be relinked at runtime. This relinking is an excellent feature because, not only does it allow dynamic function extension, but it also changes the application development model. A traditional application development model requires every resource to be implemented precisely at coding time. However, domain computing development model will allow applications to start with a small amount of functionality but then evolve with time without any changes to the code. Due to this dynamic feature, users can perform a lot of runtime optimization for performance, scalability, and availability as well as function extension by best utilizing the resource pool and relinking data flows between resources. In this case, the dynamic feature becomes a critical to functionality when the computational resources change dramatically in terms of variety, number, and function with respect to IoT devices.

4.1 Architecture

Domain computing consists of different parts and these parts perform various roles in the system, as shown in Figure 4.1.

An application is the top layer in the architecture. Applications are the programs that take the inputs and transforms them to different I/O ports following a logical sequence. An I/O port indicates a data flow for a certain functional purpose, and it can be a local file, a sensor, a transformation function or a remote stream. An application can be viewed as a logic set of I/O ports plus their connections. All data flows in and out of the domain layer underneath.

The domain layer is the second layer in the architecture, following the application layer. The domain is

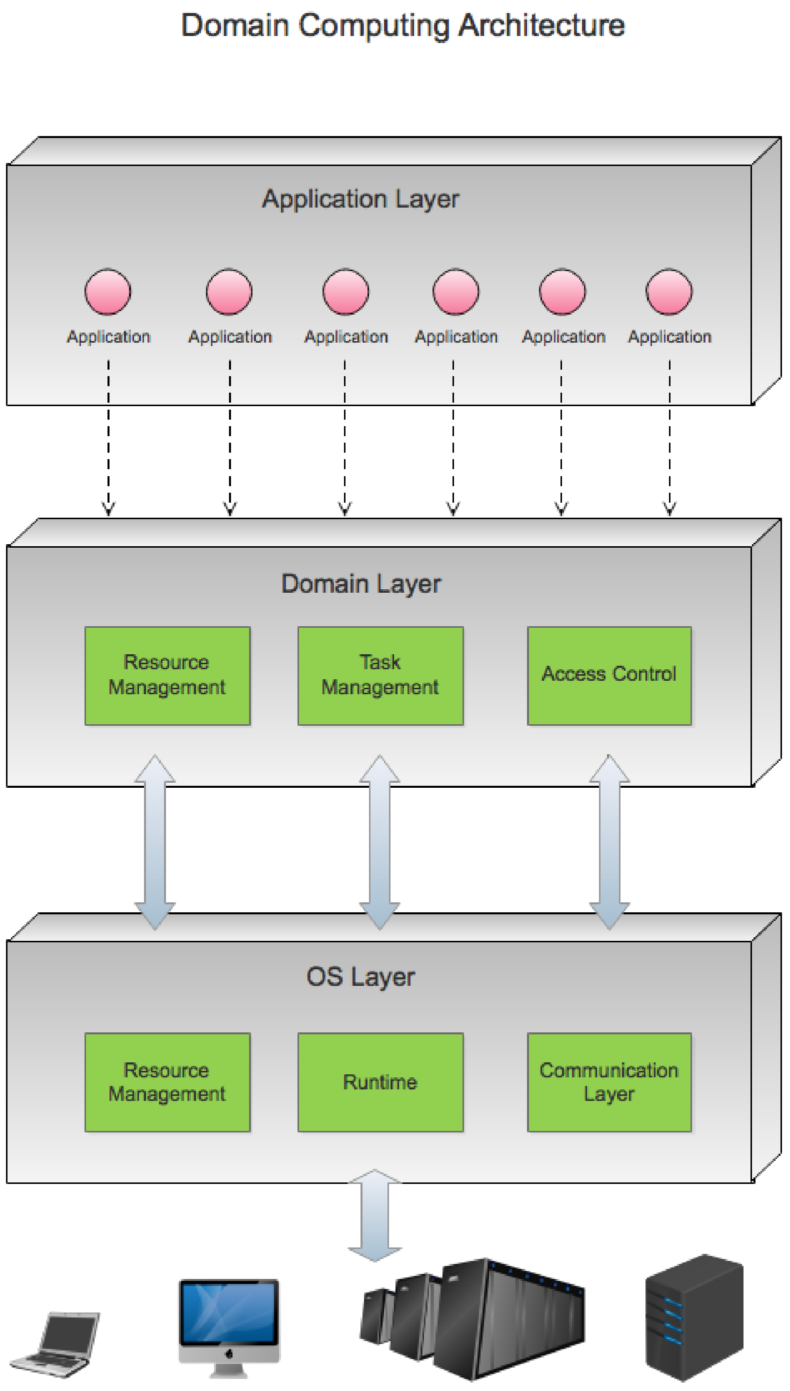

Figure 4.1: Domain Computing Architecture

a logical entity that represents a computer in traditional programming, and it can be viewed as a virtual resource pool that includes all necessary computational resources needed for an application. A domain can be defined in many ways depending on context. A domain can be all of the devices and their functions owned by a single person, all of the resources required for a project, or even all of the computational assets in an organization. The extent of a domain is at various levels and it ranges from an individual up to a large organization. The domain layer provides inputs to the application layer and takes the applications' outputs for further processes. Since a domain is similar to the concept of a computer, the domain layer consists of many components that manage the operation of applications and return the results from physical devices. Task management, Resource management, and Access control are three main components of the domain layer. Therefore having these three components in a domain will enable the management of all registered resources, allow data and I/O ports to access based on user rules, and to schedule applications to run on different devices and platforms.

The operating System (OS) layer is under the domain layer and all applications run within this layer. As a traditional OS, it controls a physical machine and provides the ability to run the user's programs. A physical machine, like a PC, laptop, tablet or even a smartphone, has the real computation capability. However, in this system, the OS layer extends the capability of the traditional OSs in order to provide services for upper layers. In addition to providing existing functions, the OS layer also should provide resource management in fine-granularity level so that all of the resources on the machine can be accessed and shared for different computation purposes. A flow control subsystem is used to manage the I/O port connections between local and remote resources at runtime. The flow control subsystem also maintains a complete list of data pipes in the local. The OS scheduler extends functionality to handle different requests on the same resource by time sharing or by another fashion. The resource conflict can be resolved in traditional OSs. All communication I/O ports, especially network I/O ports, are placed in a different layer in an OS. These ports are underneath of the flow control system so that applications can choose different ports for communicating, which will later allow smart I/O routing in terms of availability and throughput.

Different key components in domain computing are discussed in the coming sections.

4.2 Components

Domain computing consist of several key components such as resource management, task management, and runtime. These key components are distributed in different layers. Components in the application layer will be discussed first, whereas components in the domain layer will follow in the next section. Finally, components in the OS layer will be discussed.

4.2.1 Application Layer

In previous generations of computing, the focus was on local resources, networking resources, and service resources. Although networks are faster and more widely spread, computer processing is more powerful, and application development models change along the resource supply chains. Computation and relevant research has, so far, focused on the resource and service provision. There is no doubt that resources' availability and scalability are much more improved than in the past. In facts, availability is up to .9999999% failure time per year, and is easy to scale up to hundreds of node clusters.

Although they are used for different applications, the technologies behind utility computing, grid computing, and cloud computing are mainly used to provide services. For example, Cloud computing utilizes clusters to provide uninterrupted services around the world, no matter IaaS, PaaS or SaaS, so that we can consume services from anywhere with an Internet connection. This provision of uninterrupted services makes calendars, emails, stocks and weather information, as computation resources, available regardless of time. A client application running on a desktop, tablet or smartphone simply connects to the services via RESTful interfaces to represent the results. The main computation task is pushed and accomplished on the cloud side. Although, in many cases, a client can consume multiple services at the same time, and then filter and aggregate the results to form a new presentation and screen output, it becomes a central point to control the whole data flow, which means that all of the services have to flow to the central device first to be processed.

Networks have grown incredibly during the evolution of subsequent generations of computing. Networks can be easily found nearly everywhere in our lives, like utilities such as water, gas, and electricity. Although they have been used for different purposes, a variety of network technologies for short and long distance communication have been developed, including Ethernet, Bluetooth, WiFi, and Celluar. Additionally, networks provide a facility to connect and distribute resources within a large geographic area with high bandwidth and fast response time.

Applications can be seen as running on a resource network. Each node in the network represents a resource, and the edge between the node is a data link or flow. An application has to have its resource network to run. The current computing models contribute to the creation of many resource nodes and they will continue to create even more in future. However, connections between these resources are not easy to build. Physical networks have been broadly built and applied, but data flow involves much more of the networks and includes the connections between systems, platforms, software, and protocol stacks. For example, utility computing and cloud computing have created many services over the Internet, but as users we have to connect to different services from our client devices, grab all of the results from each service, and then aggregate or analyze the final results in the local device. An application has to run locally as a central control procedure despite computational resources being widely distributed over networks

So, there is a need to build a resource network, specifically a virtual resource network, to link all required resources together, regardless of physical machines and locations. Applications do not run on a physical machine anymore. Rather, they run on whatever resources are available in the network. Applications running no longer running on a physical machine is the one key feature that differentiates domain computing from all of the other computing models. Actually, a traditional application can be seen as a special form of domain computing where all of the resources are located on the same machine.

4.2.2 Domain Layer

The domain, as a key part of domain computing, is used to define the boundary of your computational resources. A domain can be created for different purposes and it crosses multiple physical machines. A user can have as many domains as needed. These domains contain and provide the resources needed to support applications. The relationship between applications and domains is flexible. Many applications can run on the same domain if the domain includes all of the resources required for each application. Or a domain only run a dedicated application as a one-to-one relationship. The one-to-one relationship between a domain and a dedicated application is related to application scenarios. For example, if you register every resource you own from all your devices and machines, you can certainly run multiple applications using it. Conversely, if you isolate a special set of resources from all of your resources, only a dedicated application will be able to run on it. Thus, domains provide the flexibility and opportunity to manage resources and applications as demand.

Resource Management

The resource management component, as a part of the domain layer, is used for managing all resources. A resource in this context is a broad concept and any resource used for computation is considered a resource. For example, a file operator, sensors, functions and communications are all considered resources. A finer granularity of a resource is better to build dynamic applications, but a compound resource as a black box still can be treated as a whole and viewed as a resource. The complexity of functions determines the granularity of a resource. If a resource performs only a single and simple function, it has a finer granularity than the one providing a single service that groups multiple functions internally. For example, a function returning a file list in a current directory that is sorted by size is considered a compound resource because it includes two functions: one function that lists all of the files in current directory and another function that sorts a list according to the input list. Resources can also be virtual, meaning that they do not have to be physical devices. Virtual resources provide great flexibility to redistribute data flows and extend the functionality at runtime in dynamic environments.

As mentioned earlier, a domain is a logical entity that represents a traditional computer. The resources required by the applications can be registered in a domain so that they can run tasks over the resources. The number and type of resources registered in a domain are determined by applications, management, and different scenarios. The more resources that are registered provides greater flexibility for different applications while fewer resources that are registered are reserved for dedicated purposes.

In network computing, the appearance of networks has made a variety of resources possible, and this has been further extended in the service-oriented computing. Domain computing continues to extend resource varieties by establishing a virtual resource network that crosses machines and networks. The resource management component builds a resource network within the domain, and each resource registered possesses the possibility to communicate with the other resources within the domain boundary.

Resource management allows a user to add and remove resources dynamically with security checking. For example, a new cloud service or a new computer can be added to a domain dynamically whether or not there is an application running on the domain. In the case of removal, resources can be instantly removed if there is no application that relies on a given resources. Also, resource management maintains the status of all registered resources within the domain so that every application can utilize the necessary resources based on their status.

The resource management manages all resources in a domain. Resource management maintains both resources and virtual resources for applications' inputs and outputs by adding, updating, and removing operations. Resource management forms a domain resource network across machines, devices, and platforms so that every resource can communicate with each other. Resources can be dynamically linked and replaced in dynamic environments and demands because of their fine granularity and virtualization.

Task Management

The task management component is used to schedule applications that run on different devices and platforms within a domain. Although it has some similarities to a traditional OS scheduler, the task management component differs in a few ways. In a traditional OS scheduler, the tasks running on a machine are scheduled by time slices and priorities, and it allows multiple users to share resources and can achieve certain goals like maximizing throughput, reducing response time by minimizing latency or meeting time deadlines. However, the task management component in domain computing behaves differently.

The task management component maintains a list of all of the applications running in a domain. While resource management builds a virtual resource network, task management maintains path sets on the network for each application so that an application's data can flow between different resources. The running application list is used for tracking purposes so that users are aware of which applications are running, what resources are used and also how they connect. A domain is just a logic entity. The real data does not flow in a domain but, instead, flows right into the next resource. The task management list is also used to adjust applications' runtime behaviors. A user or AI resource can change its data flow on the fly by connecting to different resources available because of the knowledge of an application flow network from the list, . For example, changing a video stream from a single-peer to multi-peers broadcast.

As described, the task management does not schedule applications as a traditional scheduler. Each application has its priority as it does in a traditional application, and one task can have a higher priority than the others. However, in the domain layer, the priority only represents a relative number among applications within the same domain. The priority level information is used in the OS layer instead. So the priority

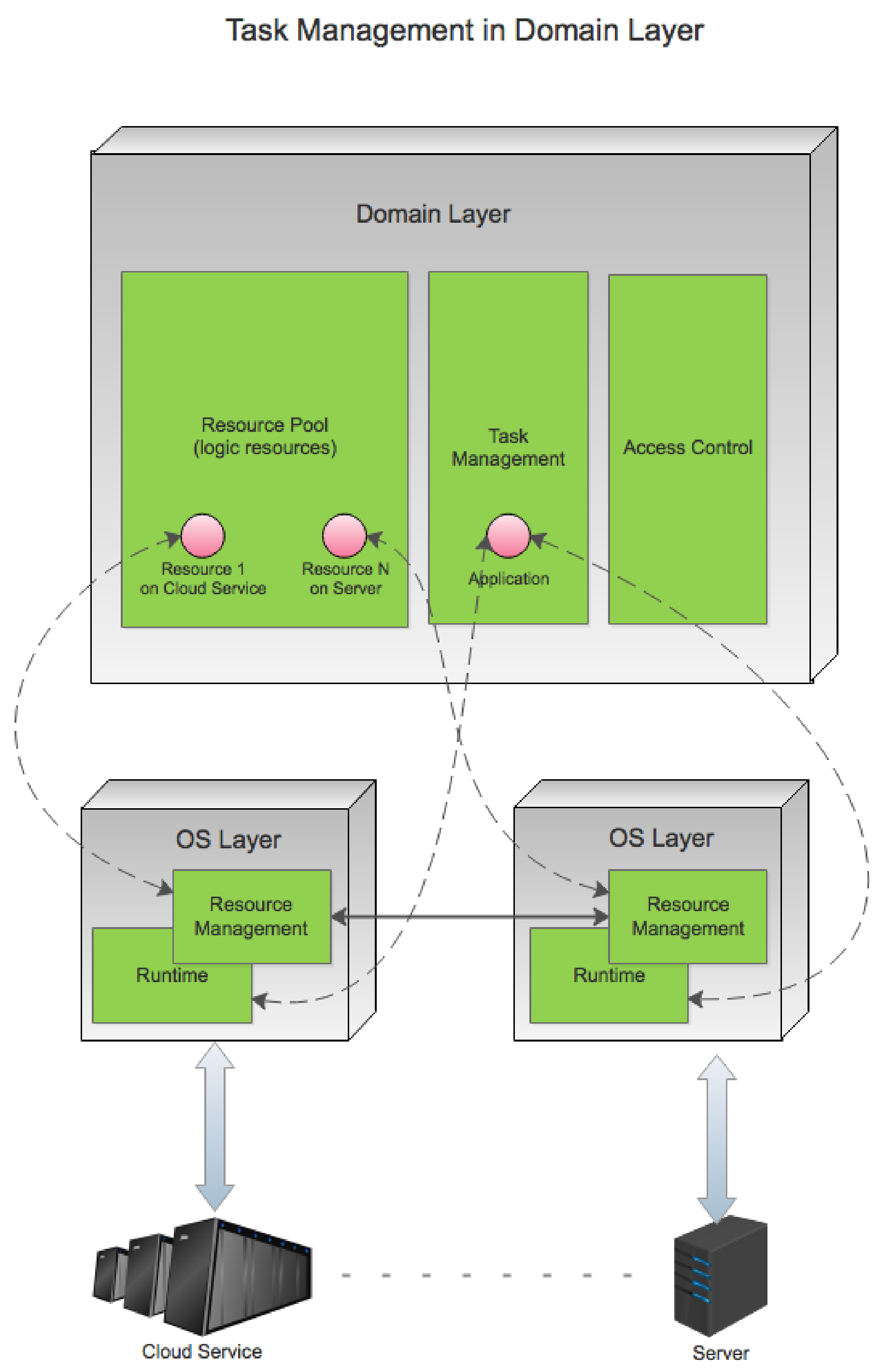

Figure 4.2: Task Management

is domain-bound. In the OS layer, the OS scheduler can schedule functions at a given priority locally. Prioritizing is only required when two or more applications from the same domain run on the same OS instance.

Task management needs a key function to make applications execute and collaborate with the OS layer. Applications and their dynamic features are implemented by communicating and negotiating with OS layers. The OS layer maintains local resources and functions and registers them into domains. Task management can communicate with the OS layer and dynamically request modification to the data flows between resources. This communication between the OS layer and task management makes function extension and data redistribution happen at runtime when an application logic flow is changed.

4.2.3 Access Control

The access control component is responsible for maintaining the permissions for resources within a domain. Within a domain, the resource management component builds a resource network, while the task management component builds and maintains path networks for different applications. The access control, however, provides the capability to control how a resource can be accessed. For example, access control determines the owner of a particular resource or who has permissions to read, write or read/write and so on.

A resource belongs to the domain that physically owns it, and that domain has ownership of the resource. For example, when an OS boots up, a default domain is automatically assigned to own all local resources. As a service, a resource can be shared with other domains. Each resource maintains a list to check who and how it can be accessed, and the shared domain registers the resource as its resource with permissions and authentication information. In other words, a domain can own and share a resource. The basic access unit is a domain. Access can be managed based on different functionalities that include reading, writing, reading/writing, duplicating, and granting.

In previous computing models, like service-oriented computing, services are provided by processes or applications. Whereas in domain computing services are provided in the function level. Due to services being provided in the function level, the way in which domain computing manages the access to resources is different. Unlike traditional access control, which is bound to users and passwords, domain access control is based on the domain. Since applications run on domains, different users have their own instead of sharing the same domain and the permissions are granted for those individual domains. In a traditional rule, a user is bound to a system. Although we have the same username on different systems, logically we consider them to be the same role. These usernames are actually considered different users with authentication. Domain-based access control removes the physical bounds between users and systems. Domain computing allows a user to have as many domains as needed so that they can build complex relationships between their domains instead of their systems. For example, one of the user's domains do not have access to a particular resource but another one does. That domain, which has the access, can share the resource with the other one as a new resource. The access control also requires that the OS layer identifies and registers the local resources to

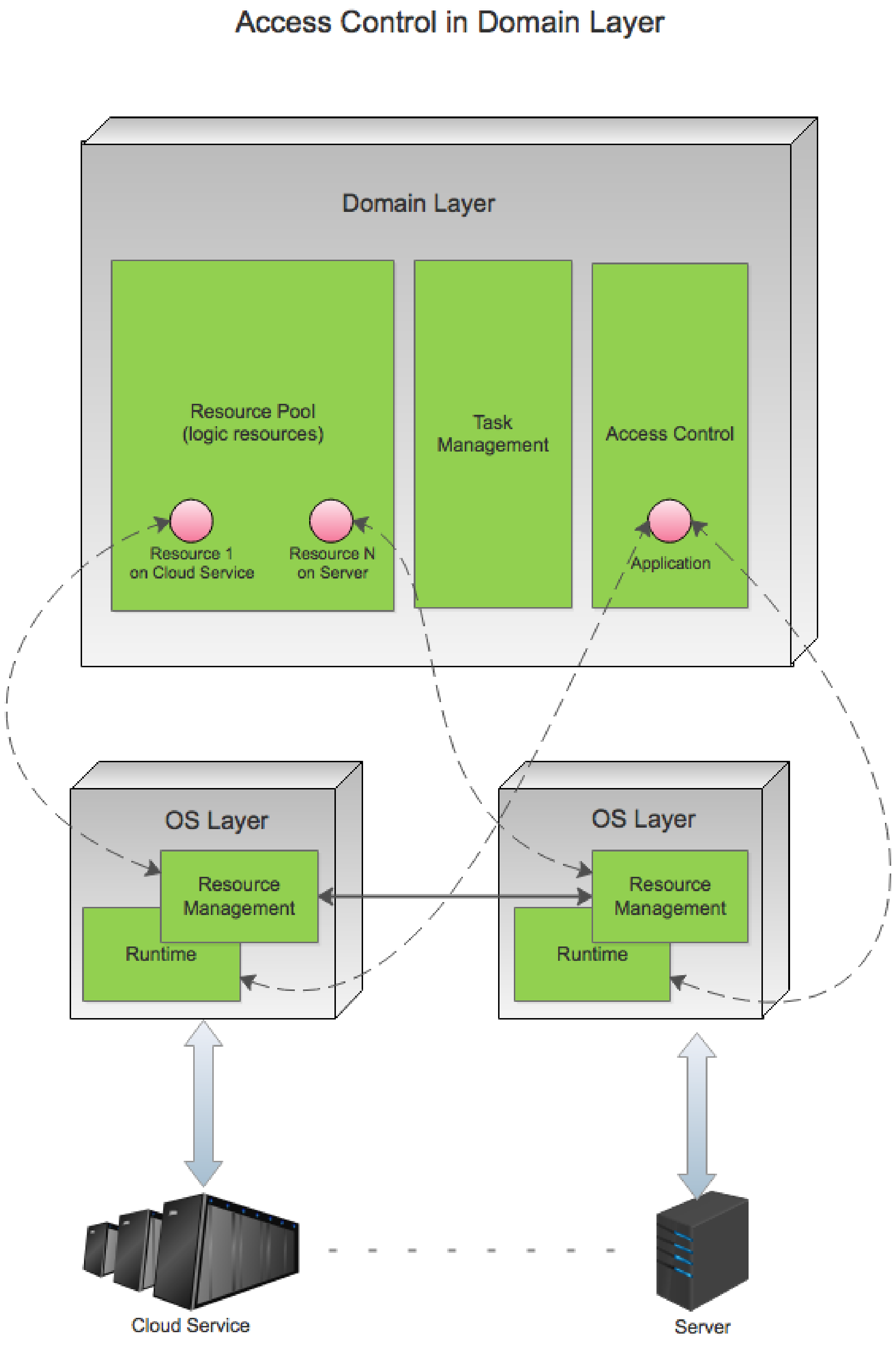

Figure 4.3: Access Control

the domains so that the resources can be used and shared with all of the other domains. This detail of the resource sharing will be discussed in the OS layer section.

4.2.4 OS Layer

The OS layer is the bottom layer of the domain computing architecture. The application and the domain layer is on top of it. The application layer is a conceptual layer consisting of running applications, although they are not traditional applications. Applications run on domains, and the domain layer provides the resources that the applications require, maintains access to the resources, and schedules the applications. However, all physical resources have to be provided by an underlying layer, OS layer.

The OS layer runs directly on physical computers. Like a traditional OS, the OS layer provides many services that includes memory management, communication, and scheduling so that we can utilize the physical devices attached to it and execute computation tasks. However, the OS layer behaves differently than a traditional OS, which manages everything. The OS layer's goal is to optimize local resources, share the local resources at different domains, and to communicate with the domain layer for the purpose of relocating, rescheduling or relinking the resources. The OS layer performs as an execution engine in domain computing architecture instead of a manager. The OS layer can be assigned and run tasks from the domain layer with the proper permissions. A domain is a logical entity, as such it requires physical execution engines in order to run applications and produce results. The OS layer consists of multiple components in order to provide computational resources to the domain layers. Each of the components is discussed in the following subsections.

Communication Layer

The communication layer includes a physical communication device and its related device drivers. Any device that can exchange information is considered a part of the communication apparatus. For example, Ethernet, WIFI, Bluetooth, NFC, and even bus devices like USB and Thunderbolt are all a part of the communication apparatus. These communication devices are considered data transportation facilities like cars and buses, which are transportation tools for people.

The device driver controls the physical devices and ships data to the upper layers. All of the devices are registered into domains and used for data transportation. These domains can be considered a communication pool. The benefit of forming these domains into a single communication pool is that thae tasks can utilize the best communication channels at runtime in terms of speed, scalability, and fault tolerance. Dynamic routing between different communication devices makes tasks more efficient and effective. For example, applications can choose a faster communication device or parallel transport data via multiple communication paths available in the communication pool at runtime. If the current communication devices crashes, the application can choose another available path in the pool.

As mentioned above, communication has been extended to a broader scope in this context. The communication layer does not treat local, remote, logical and physical communication differently. This lack of differentiation allows applications to focus on building logical data flows instead of binding to a particular communication stack, for example TCP/IP. Virtual resources, as an important part of resources, makes this possible and the details of virtual resources are discussed in the next section.

Resource Management

The resource management component was discussed in the domain layer. The resource management layer is used to manage resources within a domain as a resource pool. Resources, including virtual resources, can be dynamically added, updated and removed from the pool. The resources can then be distributed across machines or networks, can communicate with each other and also form a resource network in order to run computational tasks. However, the resource management stays on the domain level. In the OS layer, there is also a resource management component. This component maintains and optimizes precisely local physical resources. Compared to the domain resource management component, which performs global resource optimization and management, the OS resource management performs resource management for local resources.

The OS resource management maintains local resources as its primary goal. An OS must have a domain and the domain can be either dependent or independent. A domain is independent if it does not own any other resources from other machines or devices. This independent ownership of resources by a domain is similar to that of a traditional OS and computer where the OS manages everything on the machine. However, the domain can be be dependent if the registered resources are part of other domains.

The OS has a responsibility to register all of the identified local resources into the domain and the domain then automatically becomes the owner of these resources. At registration, the capability of the resources is registered in the domain inventory. For example, a keyboard has input capability, a screen has output capability, and a file system has both input and output capabilities. Since it has the ownership of the resources, the domain can share the resources with the other domains. The OS ensures that all of the physical devices are working, and maintains their status in the domain so that applications can utilize the devices based on their status at runtime.

The resource management manages both physical and virtual resources, and virtual resources play a critical role in function extension at runtime. A virtual resource possesses certain functionality but no physical device or mapping to a physical device with a different identity. Format transformation, protocol conversion or data flow replication are examples of virtual resources. The virtual resource is extremely important because

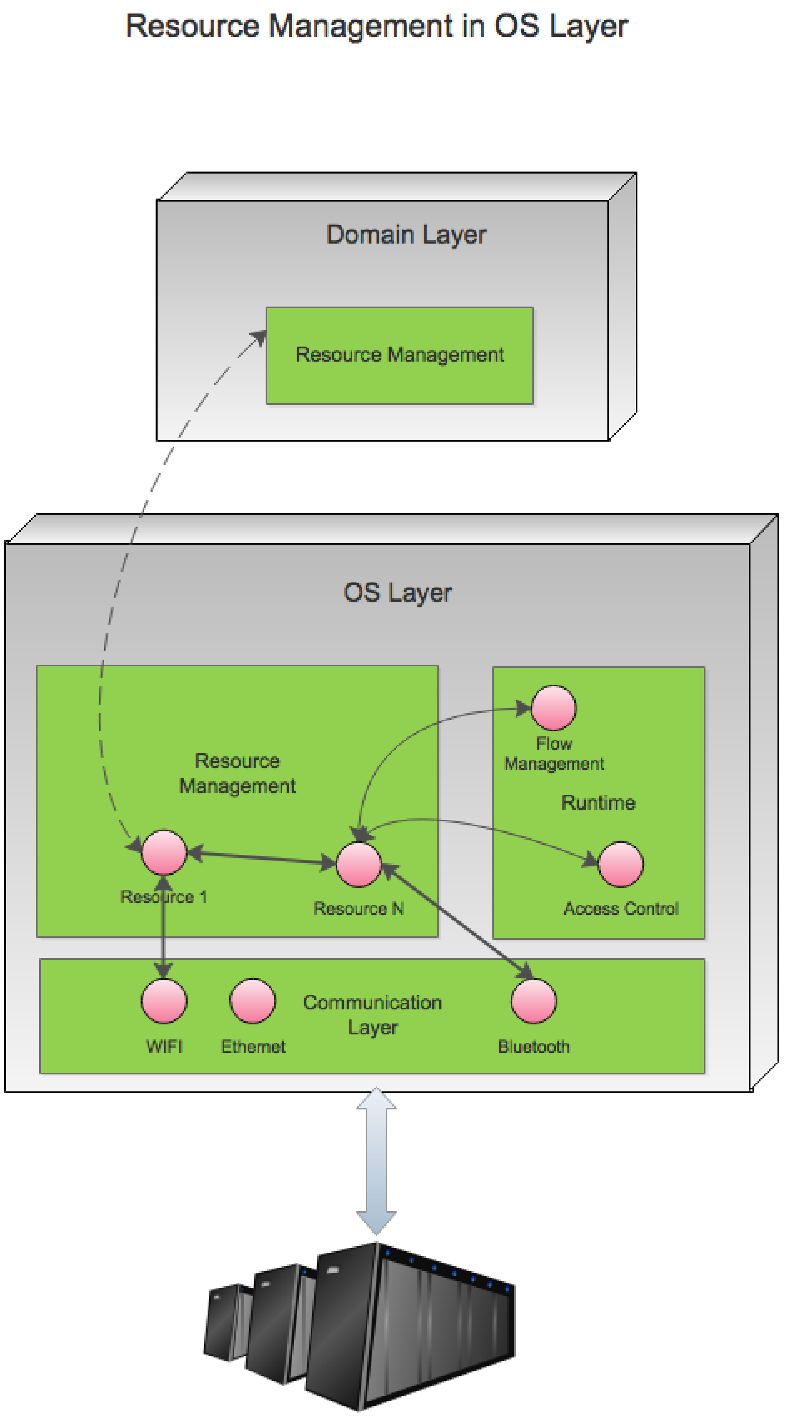

Figure 4.4: OS Resource Management

it builds intermediate layers between inputs and outputs as they are needed. The inputs and outputs can be either a physical or a virtual resource. A virtual resource is used a lot for data transformation including data, file or protocol. For example, building a protocol within a protocol requires an application to send CoAP data over a TCP/IP network. A CoAP virtual resource can then be built in such way that it transforms user data as the CoAP payload and adds the proper CoAP header information to form a complete CoAP package. The package can then be sent to a TCP/IP resource and communication driver and finally sent out. The transformation is a general term and is not limited to a one-to-one relationship. For example, when an application has a continuous data stream from a sensor as its input, a virtual data mining resource is applied and its output is used for decision making. The data mining resource possibly requires more information to draw a correct conclusion, so multiple sources of information might flow into the data mining resource in order to produce a single precise output.

Additionally, a virtual resource can be built for data redistribution purposes. Data does not always flow into a single resource. Fault tolerant and parallel computation, for example, duplicate data streams. A virtual distribution resource can redistribute a single output to multiple resources like broadcasting or multicasting at a local, remote or mixed machine. Also, a virtual distribution resource does not only forward output but also loop output back to any resource. Another interesting example is smart virtual resources with AI capabilities. An intellectual routing virtual resource can modify the data flow at runtime, and then select the best communication path based on the status of multiple resources in terms of speed, bandwidth, efficiency, and availability. Virtual resources give applications the flexibility to adapt to dynamic environments and demands. Also, an application functionality, availability, and performance can be easily extended by these virtual resources.

Scheduling is a standard component in an operating system, and it schedules multiple tasks running in a preferable way in terms of priority and timing. Scheduling is also a part of resource management although the resources and functions are scheduled instead of processes. In a traditional OS, a process or thread is a basic unit to schedule while a function or resource is the basic unit in the domain computing. A loaded resource is used to distinguish from the resources saved on disks and it runs in memory. The saved resources can be invoked at any time and run in memory. As traditional processes, running resources in the domain competes for CPU cycles and memory. All loaded resources have to be prioritized and run in such an order. Particularly for those resources shared with domains, resources should be scheduled in such a fashion that the resource is best utilized and isolated. Those resources saved on disks do not participate in the scheduling because they are not yet running. The scheduling is based on priority, status, and other factors. Any OS scheduling algorithm can be used for such scheduling but with some modifications.

Like any other resource, it is necessary to have a mechanism to control how a resource is accessed. Access control is included in this layer too. Access control allows the resource owner, usually the default domain, to share this resource with the other domains and control who, when, and how they can be accessed. Not all resources have to be shared but local resources can only be used for building local data flows. The access granularity is based on the domain. As explained previously, a domain is a logic entity that run applications, and it can represent a user, group, project or organization. A domain unifies different roles into one unlike traditional user and group control. Users can create different domains to match different levels of accesses. Once an OS is booted up, it identifies all local resources and registers them into a dependent or independent domain. That domain automatically becomes the owner of those resources. The domain can share them with the other domains by maintaining their access control lists. For those domains which are not the owner of the resource, but share them from another domain, they still can share the resource with the other domains as a resource proxy.

Runtime

Except for the communication and resource management components, the runtime component is another important component in the OS layer. The communication component maintains local devices by drivers, and registers them as resources into a domain while resource management maintains a resource pool in order to schedule and share the resources. Along the time line, the resources could be changed, the applications could require different resource paths in order to accomplish their tasks, or the resources could be shared with more domains. A runtime component is required and is responsible for this management.

When resources or their status is changed, the runtime component can add, modify and remove any local resources from the domain, including their status and metadata such as capability and ownership. Similar to a traditional OS, maintaining all of the devices and resources is critical for all tasks and domains because availability determines if the tasks can continue running or if the domains have the capability to support the tasks.

Access control will keep changing as the resource are changed. Although there is an access control component in the domain layer, it is a logic entity. Tasks like permission granting and authentication have to be performed by the access control component in the OS layer. When a domain sends a request asking for access to a resource, the runtime component can accept and deny the request, meanwhile, when the domain has the permission and queries data, it checks if the request is valid with the proper credential.

The most dynamic changes are linkages, or flows, between local and remote resources. In fast paced and rapidly changing environment, computational tasks are required to quickly respond to dynamic changes. Each application has its paths to process data and accesses to different distributed resources. When domains contain a greater variety of resources, and better resources become available, the application needs to adapt to the new environment in order to gain better performance by relinking the resources at runtime. The access path in the domain resource network keeps changing not only because of different applications, but also because of the same applications running and using different resources. A resource represents a node in the resource network and a data flow represents a link between nodes. Once a new flow is required, a domain sends the link request to all resources involved and the runtime components on corresponding OS layers try to link to the other resources. The dynamic linkage feature is the key to adapt dynamic environments, demands, and applications and, with this feature, all applications can run within the domains where all required resources are available.

4.3 System Discussion

Domain computing introduces domains as a replacement to the traditional computer for running applications. Domain computing links different computational resources that are distributed over networks into domains so that a user can significantly improve and optimize the resource utilization at a global level. By negotiating with different layers, domain computing dynamically links different resources, shares resources, and cooperates with domains at runtime so that the applications can adapt quickly to dynamic and changing environments as well as improve performance. Also, domain computing presents a new way to develop applications, which are resource stateless based.

Domain computing is a computing system possessing various properties. In the previous sections, the system architecture of domain computing and its core components have been discussed. In this section, some important system properties are discussed, including scalability, availability, and security and the application development model. Some user cases are also included in the following sections.

4.3.1 Scalability

Scalability is one of the most important properties of a system, and it measures the process capability when a significant amount of work comes. Poor scalability of a system fails to support applications and business opportunities. Also, it wastes investment into computational infrastructures. There are two major ways to measure scalability: Horizontal scaling and Vertical scaling. Horizontal scaling, also known as scale in/out, measures performance changes when additional independent computational nodes are added onto the system. For example, a system can handle 50% more user requests by adding two servers into the file clusters. Another scaling method is called vertical scaling, also known as scale up/down. Vertical scaling measures performance changes after extra computational resources, like memory or CPU, are added into a single computational node. For example, a system can handle 30% more user requests by adding 2GB memory and replacing the old hard disks with faster SSDs.

Domain computing system can scale both horizontally and vertically. By adding local resources to a computational node, its domain has more resources to handle tasks. The added resources can be either unique, duplicated or competitive. Any new unique resource added to a domain provides a new opportunity to enable different applications. When a duplicated resource is added, it gives the ability to the node to balance workloads from the original resource. For example, if an additional word counter resource is added, a document can be calculated close to 50% faster, that is if the CPU is not a bottleneck because these two tasks are running in parallel and each word counter only handles half the document. A faster resource, in terms of speed and capability, will provide better performance if a competitive resource is added.

Domain computing systems scale horizontally similar to vertical scaling because of its decentralized architecture. In general, a centralized system has limited scalability because the central point becomes a bottleneck. A decentralized system has the advantage of scalability since the central point is removed, thus removing the bottleneck domain computing treats all computational resources the same, regardless if they are local or remote, and tries to utilize all the resources in the domain the best way possible in order to get better performance, availability, and scalability. When a new node is added into a domain, the domain can be extended by adding unique, duplicated or competitive resources on the new node, which is same as vertical scaling. Since the domain sits across multiple machines, no matter where the resources are added, the resources become part of the domain. The original workload can be distributed on the nodes, including the new one.

Runtime system scalability is also important for many applications. Since some applications can not be stopped, functionality extension is extremely challenging. For example, a large amount of sensor data is needed for two different purposes and this data must be processed and analyzed. However, the current system only handle one portion of the analytic work. The second portion of the analytic work can be done by adding a new computational node. In the domain computing system, that sensor data stream can be easily duplicated by adding a virtual distribution resource and forwarded to the new node. The new node can completely handle the analytic work without an interruption to the original node and its flows. The domain computing system is based on a distributed resource network. The domain computing system can easily duplicate or relocate resources and dynamically build data flows at runtime so as to flexibly redistribute and even redesign workload dynamically.

4.3.2 Availability and Reliability

Another important system property is availability, which measures the ratio of the total time that a system continues to function. Availability can significantly improve user satisfaction, continue operations and competitive advantage but at the same time can lead to less productivity, less revenue, increased operation cost, bad profile, and even lawsuits. For example, a long system downtime of an eCommerce website can lead to fewer customers interested in continuing to shop, and subsequently result in less revenue generated. The business's reputation suffers because customers do not trust that the service can be delivered consistently. High availability is critical, especially for applications such as banking, manufacturing, and utility production and scientific research. Availability has various characteristics but reliability, continuous operation, error detection and recoverability are the most important ones. To improve a system components', reliability is important because an increase in availability and reliability of each part within a system translates to a higher availability of the system overall. The continuous operation ensures that a system has no, or a minimum, downtime so that users can access the data and services at any time. Timely error detection helps to identify system errors or potential problems in order to reduce system downtime. Once an unexpected event happens, and results in a system failure, recoverability provides the ability to bring a system up and running in a short period of time and minimize loss.

Domain computing provides the flexibility to manage the level of availability in a system. Each reliable system component makes the entire system more reliable. Although software component reliability is about improving the code's structure and avoiding logic traps, like memory leaking or deadlocking, domain computing makes these coding tasks simpler and easier because a resource only has a small functionality instead of a highly complex functionality. Small functions require small pieces of code, which is easier to identify and correct than larger pieces of code. In a domain, users can easily add duplicated or competitive resources so that any resource failure can be quickly replaced by redundant resources, which will allow applications to continue operating. Also, a domain computing system becomes fault tolerant because of its runtime flow routing feature. All data flow can be rerouted at runtime if a resource failure is identified. Because the rerouting is simple and easy to do, the applications do not need a complete stop and can be fixed at runtime, which reduces the system maintenance cost. One such example is when an Ethernet connection is down: the applications can still continue to work if there is a WIFI or BT connection. Alternatively, among all existing connections, the best channel in term of bandwidth and speed is chosen. Timely error detection is important to continuous operations. Problem identification that happens more quickly will enable a system to last longer. However, error detection can sometimes be a costly operation sometimes. Domain computing allows data redistribution online so that users can easily duplicate a data flow without interfering the current flow for monitoring purposes. Even the monitoring can be relocated to a separate machine if the current machine does not have enough CPU cycles. Once an error or failure has been detected, a user or a smart relocation resource is notified. The smart relocation resource can resolve the problem automatically by relocating the data flow to its duplicated or competitive resource without human interference. As discussed above, the domain computing system can easily add and reroute data flows at runtime. A problematic system can be quickly recovered by dynamic resource relocation, which enhances the recoverability of the system.

4.3.3 Security

As more digital threats arise, security becomes a critical part of a system. Security issues are harmful to users' privacy and sensitive data, particularly in distributed systems. A secured system includes layers of security components from communications to access controls.

Communication is a fundamental component for all network applications. Communication traffic is easily exposed to 3rd parties. Although a wired network makes its traffic harder to be exposed, due to physical accesses, a wireless network is much more vulnerable to this threat. Exposing a wireless network's traffic does not even require physical access. If a 3rd party user sits close enough to a wireless network, he can pick up its signals and start listening. An easy and effective way to reduce such a threat is to encrypt all communications from end to end. Domain computing can easily support communication encryption by adding an encryption resource regardless of communication protocols underneath. The data is sent to the resource for encrypting before it flows to a protocol and communication resource so that all communications can be encrypted. Users can add a variety of encryption resources like AES128 and AES256 for different purposes.

Local encryption is also critical because mobile devices and network accesses have become popular targets of security breaches. Another big security problem for mobile devices is data breaching when mobile devices are lost. Encryption of local data is necessary in many cases. Like communication encryption, the encryption resource can be applied before any data is written to storage, which means that a system can share the encryption resource that will prevent both communication as well as local breaching.

As network access becomes dominant, access control becomes a fundamental component to guard against who, when and where with regards to access. In the domain and the OS layers, the access control component has been discussed. A domain can own a resource and share it with the other domains. The resources can only be accessed with proper permission like reading, writing, r/w or duplication. Although the permission is not on the user level as in a traditional system, the domain rather than the user, optimize security management globally across distributed resources over networks. Resources in domain computing are individual units so that domain security has an even finer access control on each individual unit.

4.3.4 Development

Domain computing redefines applications and does not run like it does in a traditional system. A traditional application has all of its functions and resources locally, though it is possible some resources are remote. The results produced in remote have to be in local eventually because all of the functions are in local. Instead, domain computing runs applications on a domain resource network where the computational resources are available. The resources can be located on a single machine or multiple machines across networks.

Because the application definition is changed, the application development is changed correspondingly. The domain applications consist of a series of functions and resources to form data flows, so that the development focuses on individual functions and connects them at runtime. Ideally, a resource performs a single function. The single function can be simple like encryption or an unbreakable unit built by multiple functions. More integrated functions in a single function are less flexible and efficient to share and utilize. A single function is easy and fast to develop. A single function's simplicity makes it easy to unit test, identify logic defects, and to trap it quickly, which speeds up development cycles. Developers have less runtime features that should be considered at coding time and these features can be shifted and built at runtime. For example, fault-tolerant can be supported at runtime instead of development time by dynamically adding and connecting resources via proper policies. A small and simple function is easy to deploy because of its minimum requirement to environments.

Chapter 5
Evaluation

In this chapter, the prototype system and its architecture are discussed. The feasibility and effectiveness of the prototype system is examined by evaluating key system performance attributes including network throughput, response time and their variability. Furthermore, this work has been published in papers and journals Xue et. al [67] [68] [66]. The IoT brings millions of resources together via heterogeneous networks and it is the future computing environment. It is interesting to examine how the domain computing system performs in such an environment. That is to say, the performance of resource publishing and subscription is one of the fundamental mechanisms in IoT, and is also examined. Security always plays a crucial role in networks and protects users' privacy and their sensitive data. Encrypted communication is necessary in many applications because network communication is easily exploited,. Naturally, the prototype system performance under secured communication becomes a part of its evaluation. This chapter is organized in the following way. Following the system overview is evaluation motivation. Experiment design, design criteria and evaluation metric are included. In the experiment setup subsection, the system's setup hardware is described. A description of a dedicated network and a public network is also included. Following that, the performance data that was collected is discussed.

5.1 System Overview

The prototype system does not implement all components. Security related components, like access control, are not implemented though an encryption mechanism is used. The prototype system consists of client, server and network infrastructure. The client and server are the roles that initiate sending data and receiving the data. A client application was built and used to collect select sensor data from a mobile device and then sends this sensor data to a remote server over a wireless network. Basically what the server does receives a resource observation request and returns a series of sensor data to the client until the client sends a RST request to cancel the subscription. Both client and server run the same prototype application, one is in client mode and the other is in server mode, so that they can communicate. Sensor data is generated via a special hardware event. Once the application captures the event, it reads raw sensor data, puts it into a CoAP package and sends it out. Regarding different test scenarios, all of the communications between client and server are accomplished in both a dedicated wireless network as well as public wireless network. The details of hardware infrastructure is included in the experiment setup section.

Computational resources used in a domain computing could be widely distributed over networks. Furthermores, as the Internet of Things grows exponentially, the portion of the resources distributed over networks becomes dominant and their distribution is wider. Identifing a resource becomes more challenging. So, in the prototype system, the CoAP protocol is implemented as the communication facility to transport the data. The Constrained Application Protocol (RFC7252) was initially proposed by ARM and Universitaet Bremen TZI at June 2014. CoAP is a specific transfer protocol for machine-to-machine communication and for use with constrained nodes and constrained networks. The constrained nodes often have small microcontrollers with small amounts of ROM and RAM, while constrained networks often have low power and high packet error rates with a typical throughput of 10s of kbit/s. CoAP is based on UDP protocol and package size varies from a minimum of 4 bytes to a maximum of 1024 bytes. CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP can also easily interface with HTTP for integration with the Web[55].

There are many application protocols but two emerging protocols are designed for high resource-constrained environment, the Message Queuing Telemetry Transport (MQTT) and the Constrained Application Protocol (CoAP). By applying these protocols on the resource-constrained environment, we can improve device and application performance in terms of bandwidth consumption, battery lifetime and communication latency. Although they both can achieve such goals, they have different application scenarios to adapt to [25]. MQTT and CoAP have very different communication patterns. MQTT is a pub/sub protocol and CoAP is a RESTful protocol with built-in pub/sub mechanism. Since RESTful protocol uses HTTP verbs, CoAP can be easily integrated with the web application through an HTTP-CoAP proxy, which is an extremely critical feature for distributed applications running in the Cloud. From a transportation protocol perspective, both MQTT and CoAP provide basic congestion control by retransmitting messages that are not acknowledged. However, CoAP is based on UDP while MQTT is based on TCP. TCP provides a more reliable connection but with a lot of extra overhead as compared to UDP so that the same thing can be expected on MQTT and CoAP. The reliable connection provided by TCP is not necessarily an advantage in a sensor network. Sensor application generally tolerates few missing data packages since a sensor can generate thousands data packages in every second. Such a large data generation rate can degrade application performance by TCP due to extra overhead in its protocol. A reduction in extra overhead can be easily amplified in such a high sending rate. CoAP protocol uses UDP so it has more advantage on the extra overhead. Except for the above differences, CoAP has unique advantages that MQTT does not support. Since CoAP is a RESTful protocol and can be easily transformed into HTTP, a CoAP application can obtain a better response performance by HTTP caching. CoAP also provides a mechanism called Blockwise, which is a solution for transfer of large set of data in a block fashion. One of the most important features for a sensor network is resource publishing and subscribing. CoAP has a built-in mechanism called Observe to discover resources or services in the network.

5.2 Experiment Design

In Table 5.1, all evaluation goals are listed and followed by their corresponding experiments.

Evaluation Goal	Experiment
I/O throughput, Response time, and Their variance	1) Dedicated Network: Sensor Data Throughput, Response Time, and Variance 2) Public Network: Sensor Data Throughput, Response Time, and Variance
Resource Publishing and Subscription	1) Dedicated Network: Subscription Throughput, Response Time, and Variance 2) Public Network: Subscription Throughput, Response Time, and Variance
Secured Communications	1) Dedicated Network: AES128, AES192 and AES256 Encryption 2) Public Network: AES128, AES192 and AES256 Encryption

Table 5.1: Summary of Experimental Design

The experimental aspects are first identified and will allow us to obtain wireless network I/O performance data. The experiment aspects help us to obtain and understand consistent performance results. Here is a list of the aspects identified:

- Resource discovery and I/O subscription
- CoAP payload size
- Packet sending Interval
- Type of wireless network
- Network protocol
- Data source
- Repeat packet sending times
- OS environment

All of the above aspects are divided into two major groups. The first five aspects help us to better understand wireless network I/O performance. The last three aspects help us to produce consistent experiment results. Every aspect is briefly introduced here. Resource discovery is an important mechanism in a sensor network. CoAP has the built-in mechanism called Observe. The client sends out a GET request to a unique URI (./well-known/core) on a server. The URI is part of the CoAP standard. The server receives such a request and it knows that this is the request to obtain all resource available on this device. The server returns a resource list to the client. Once the client makes a decision on which sensor (I/O resource) to subscribe, it sends out an Observe request with the chosen sensor id to the server for receiving the sensor's update. The server adds the client to a subscription list on the chosen resource and once the sensor data is changed, all subscribers on the list are updated. The update does not require any further requests from the client. The server keeps sending sensor data to the client instead of the client pulling. If the client no longer has interest in the sensor data, it sends out an RST request to the server to cancel the sensor I/O subscription. No further sensor data will be sent to the client. CoAP payload size is used to investigate the relationship between package size and performance. The maximum payload size in CoAP is 1024 bytes; multiple payload sizes will be used in the experiments. Sending interval is used for measuring sensor I/O throughput. The server sends sensor data to notify the client after a fixed time interval or anytime the sensor produces data. We need to find out the upper and lower boundary of the throughput since these experiments are designed for performance evaluation. The lower boundary is obviously 0 and the upper boundary depends on not only network delay but also how fast a server can send out the data. So, sending intervals in all tests are 0, which means that once a sensor data is sent out from the server then the next sensor data is ready for sending. Both dedicated network and public network will be tested in order to understand how different types of wireless networks impact the I/O performance. A dedicated network, as its name indicates, has no other network traffic except the experiment's traffic. A public network will be used to simulate a real world environment. UDP protocol is used in the experiments since the experiments are based on CoAP. As mentioned above, the last three aspects were considered for getting consistent results. For data source, a consistent and stable data generator should be chosen for every experiment. Otherwise, the result will not be consistent. Also, a conclusion should not be made based on a strong bias from individual sampling, so packets were repeatedly sent over multiple times to see if there was a variability of their response time and throughput. The status of the current running OS impacts network and I/O performance as well. There was no other applications running, especially background applications, that would impact the performance due to the overhead of scheduling and competing network bandwidth.

Once these vital aspects for the experiments were identified, the experiment sets were designed as below.

For A Dedicated Network

Dedicated Network - an ideal environment for testing and simulating a home wireless network.

For each I/O payload size in the size list 16, 32, 64, 128, 256, 512 and 1024 bytes, the following experiments were tested:

1. Sensor I/O subscription - sending a request to the server to subscribe specified sensor I/O data. The

sensor I/O data transfer repeats 100 times. Data collected were:

- Subscription response time
- Throughput for I/O subscription
- Sensor I/O data transfer rate
- Service overhead for subscription
- Network overhead for subscription

2. Repeat above subscription - to see if there is variability between each I/O subscription. Subscription repeats 30 times. Data collected were:

    - Subscription response time
    - Throughput for I/O subscription
    - Overhead for cold subscription
    - Overhead for warm subscription

For A Public Network

Public network - a real complex environment for testing and simulating a public WIFI network. The public network experiment repeated the same set of experiments for the dedicated network.

To evaluate the performance result, measurement metrics were identified as:

- Subscription Throughput - maximum number I/O subscriptions per second
- Subscription Response time - I/O subscription response time in milliseconds
- Sensor Data Throughput - maximum sensor I/O data per second in KB
- Sensor Data Response time - I/O data transfer response time in milliseconds
- Subscription service time at cold server - time spent on server to handle the I/O subscription for the very first time
- Subscription service time at warm server - time spent on server to handle the I/O subscription, not the first time
- Variability - between each I/O data transfer and also between each I/O subscription

Throughput is measured by the number of bytes or number of subscriptions can be sent or made every second and is based on different payload sizes. Response time is measured in milliseconds. It was interesting to see if there was a big variance between repeated experiments. We may have found something else if a big difference exists.

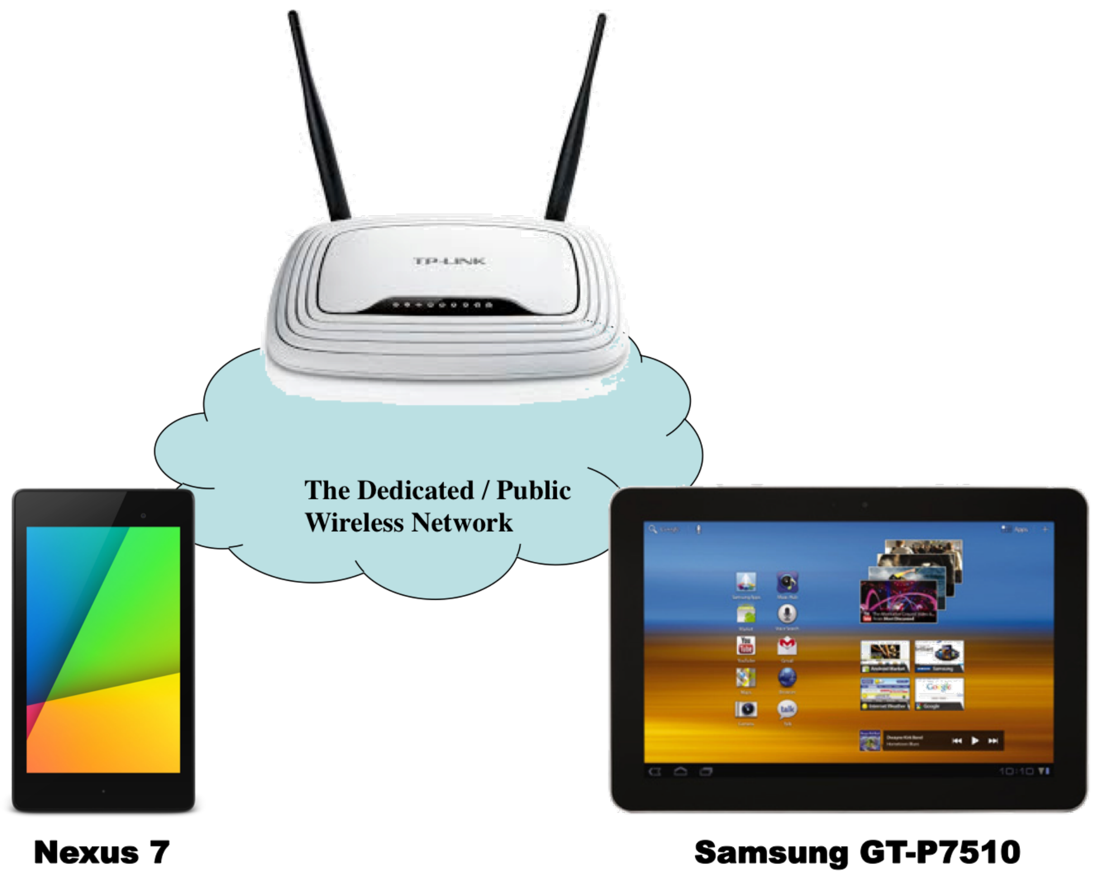

Figure 5.1: System Setup

5.3 Hardware Setup

There are two major sets of experiments: a set tested on a dedicated network and another set tested on a public network. The dedicated network consisted of a dedicated wireless router and two Android devices. The wireless router was a TP-Link TL-WR841N wireless router with 802.11b/g/n support and a maximum 300Mbit/s bandwidth. Two Android devices both supported 802.1a/b/g/n protocol and with a 100Mbit/s bandwidth. One device was a Google Nexus 7 that was running on Android 4.4.3, and another one was Samsung GT-P7510 that ran on Android 4.0.4. The system diagram is listed in Figure 5.1. During the experiments, there was no background applications running on either of the devices in order to eliminate interference and possible delay of sending packets. And there were no other devices in the network, so there was no interference in the network. Both devices and the router were within 3 meters, and wireless signals on all devices were excellent and stable at all times, so we could assume that there was no delay due to a bad signal or an unstable connection. For the public network, the wireless network in a Starbucks coffee shop was chosen. All devices had an excellent wireless signal and the wireless router should have been within a 10-meter range. Both devices directly connected to these networks.

The accelerometer sensor was chosen as a data source for the experiments because a consistent data generator was needed. Otherwise, the performance measurement would be different due to the different input rate, which was not an apples-to-apples comparison. For example, low throughput could have been caused by a low input rate and network delay but we do not know which one, or if both was the root cause until we removed one from the test. It was critical that all experiments used the same consistent and stable data source. The accelerometer continuously generated more than 1000 events per second, and each time it generated exactly 12 bytes of data.

5.4 Data Collection

In this section, how requests were sent and how performance data was collected are described. In the experiment environment, there had two Android devices. One device played a client role and another one ran as a server. A Samsung GT-P7510 was the client that sent all of the requests and all raw data was collected on it as well. Before moving into the measurement phase of a test, the Samsung GT-P7510 connected to a remote server, which was Nexus 7 and discovered all of its available sensors. A single sensor was chosen and used in all of the experiments. I/O subscription and I/O transfer were repeated multiple times in order to remove bias on individual sampling. The subscription was repeated 30 times and the I/O transfer was repeated 100 times for each I/O subscription.

The type of data that was collected are listed below:

- I/O packet sequential number
- Payload size in bytes
- Response time for the subscription and data transfer, in milliseconds
- Throughput for data transfer KB/s and for subscription number per second
- Service time for cold and warm subscription, in milliseconds

In each experiment, all of the data collected was recorded in a log file. When an experiment finished, mean response time was calculated from individual subscription or data transfer response time, and the experiment's throughput was calculated from the sum of the sent I/O data size divided by its total elapsed transfer time.

5.5 Result Analysis

Following the experiment design set out above, all of the tests in different networks were run and all performance data was collected for analysis. The collected raw data were imported and processed in Excel. The

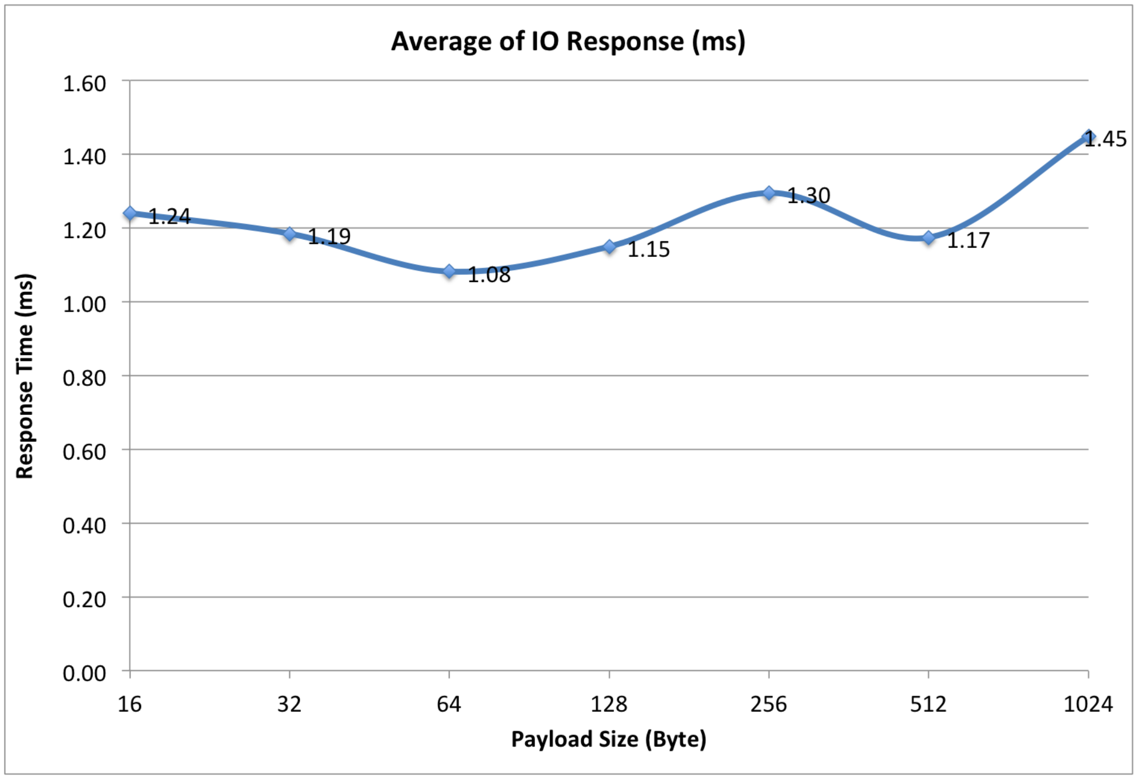

Figure 5.2: Dedicated Network - Average IO Data Transfer Response Time

following performance graphics were produced based on the raw data. For all graphics, except where indicated, the horizontal axis represents the payload size from 16 up to 1024 bytes. The vertical axis represents KB/s in throughput graphics and millisecond in response time graphics.

Dedicated Network

Figure 5.2 shows that there is no difference on data transfer response time. Data transfer response times all fall within a small range, 1.08 to 1.45 ms. Since all of the response times are very fast, any network interference or OS activity could have easily caused the response time to fluctuate. Although at 64 and 512 bytes payload the response time drops a bit, there is a general trend of response time increasing slowly as the payload size increases. The data transfer rate increases along with payload size, and the throughput ends up with 791 KB/s on payload size 1024 bytes (Figure 5.3). This result implies that bigger payload size has a higher I/O throughput. Looking closely at the data, we can see that the I/O throughput approximately doubles as the payload size doubles. For the subscription response time, until 128 bytes of payload size is reached, the response time remains low and is less than 100ms (Figure 5.4). However, starting from a 256 bytes payload, the response time increased rapidly and ended up with 515ms at 1024 bytes payload. The response time was subject to payload size because it measured the elapsed time, including the time that the first sensor data was received. Before the 128 bytes payload, although the size doubles every time, the payload sizes are still considered relatively small. After the 128 bytes payload, the doubled payload size

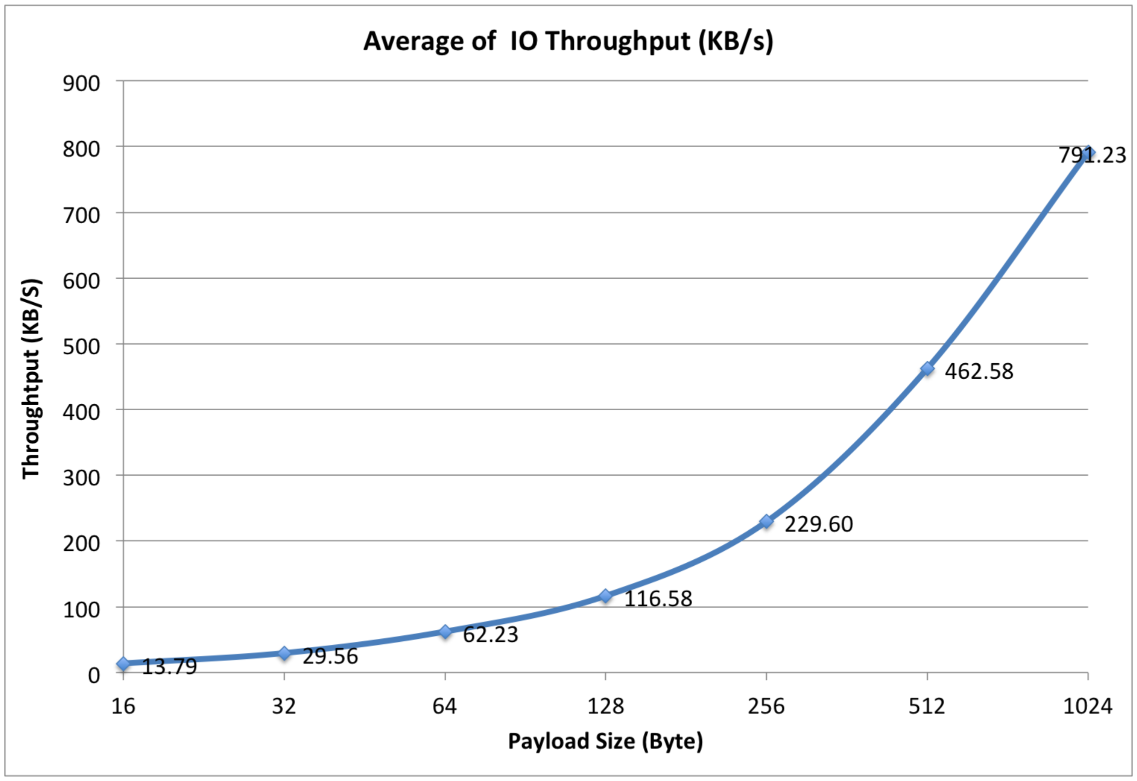

Figure 5.3: Dedicated Network - Average IO Data Transfer Throughput

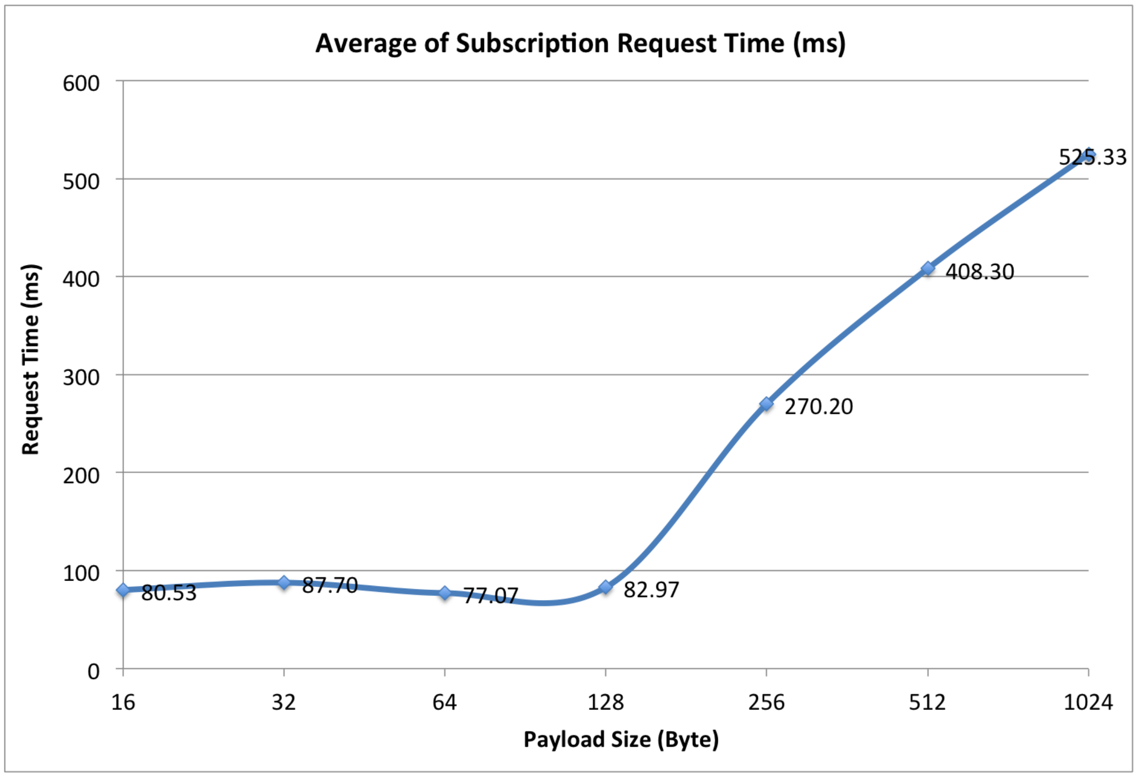

Figure 5.4: Dedicated Network - Average Subscription Response Time

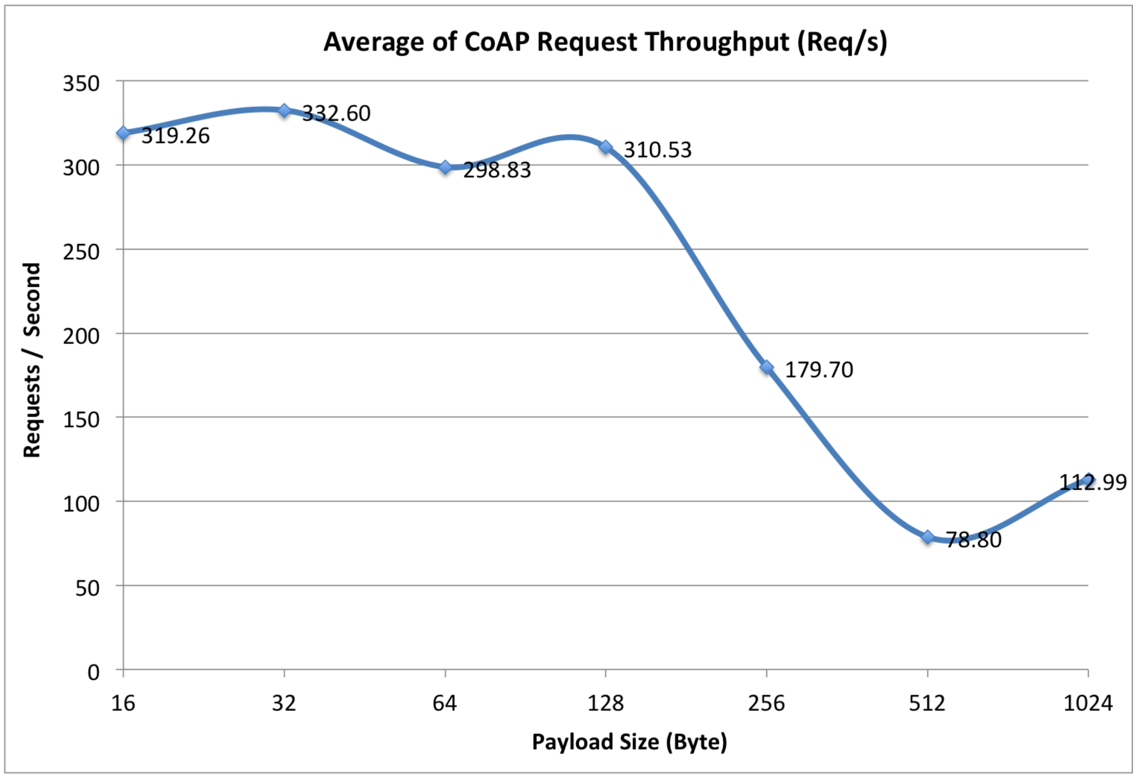

Figure 5.5: Dedicated Network - Average Subscription Throughput

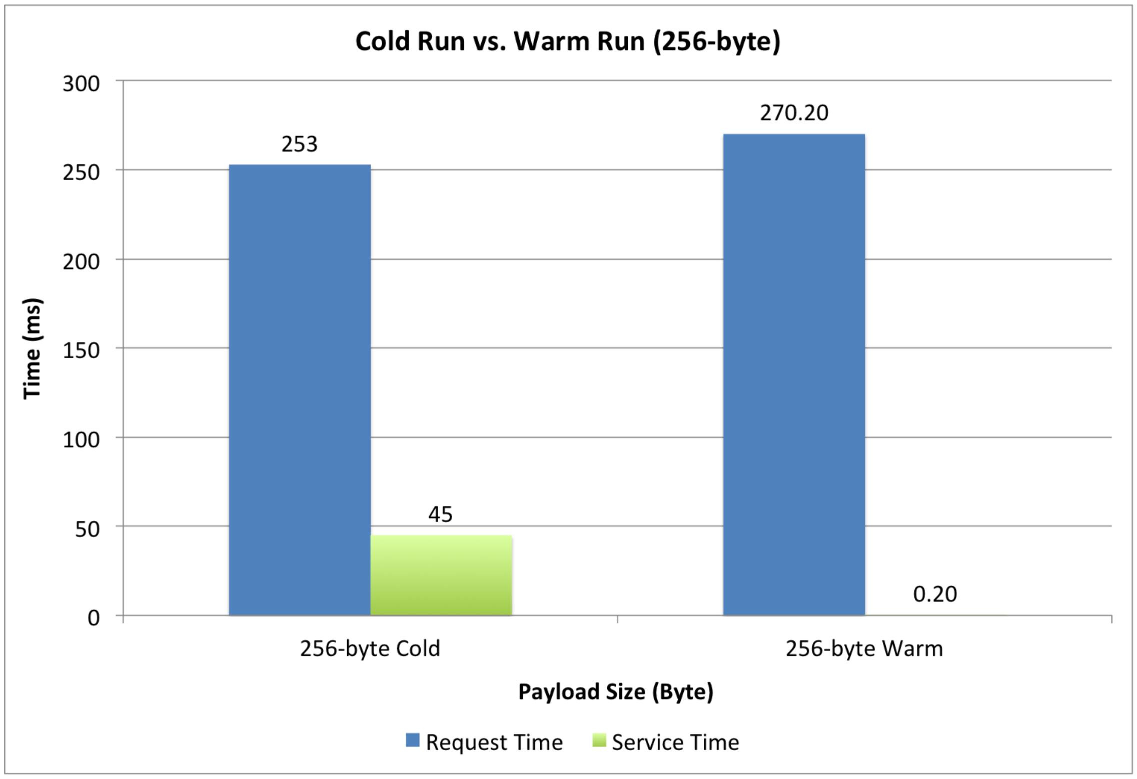

Figure 5.6: Cold Subscription versus Warm Subscription on Response Time

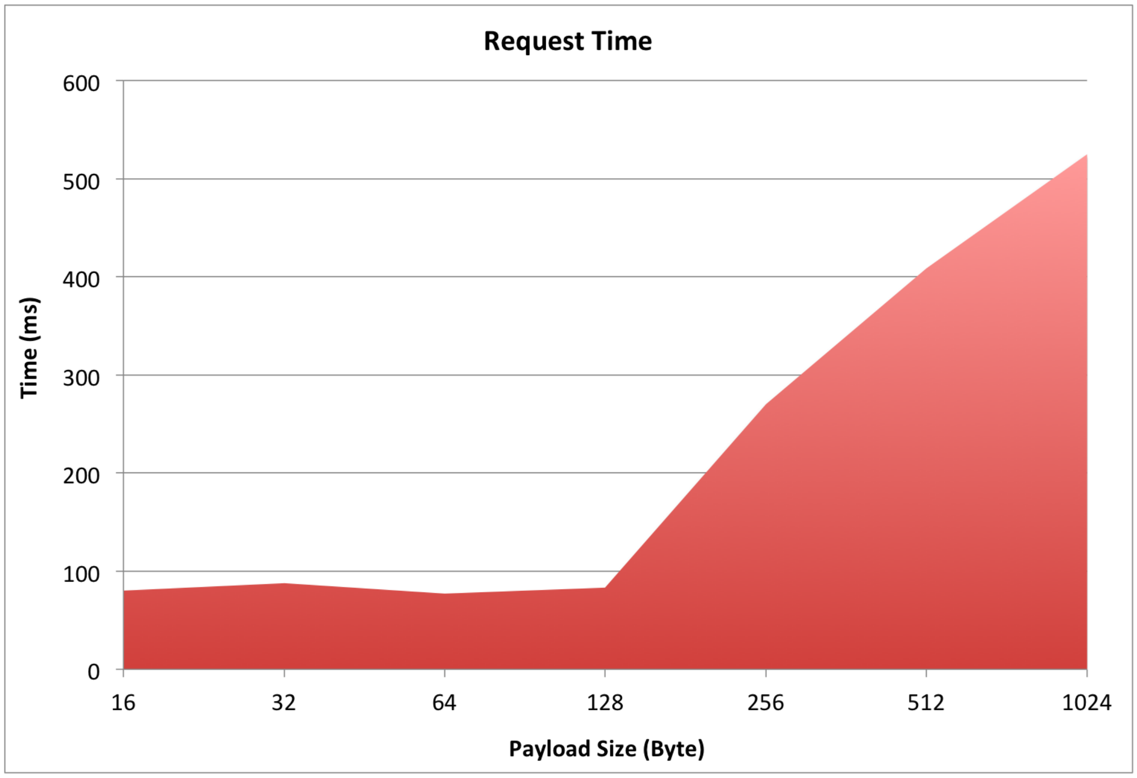

Figure 5.7: Subscription Response Time

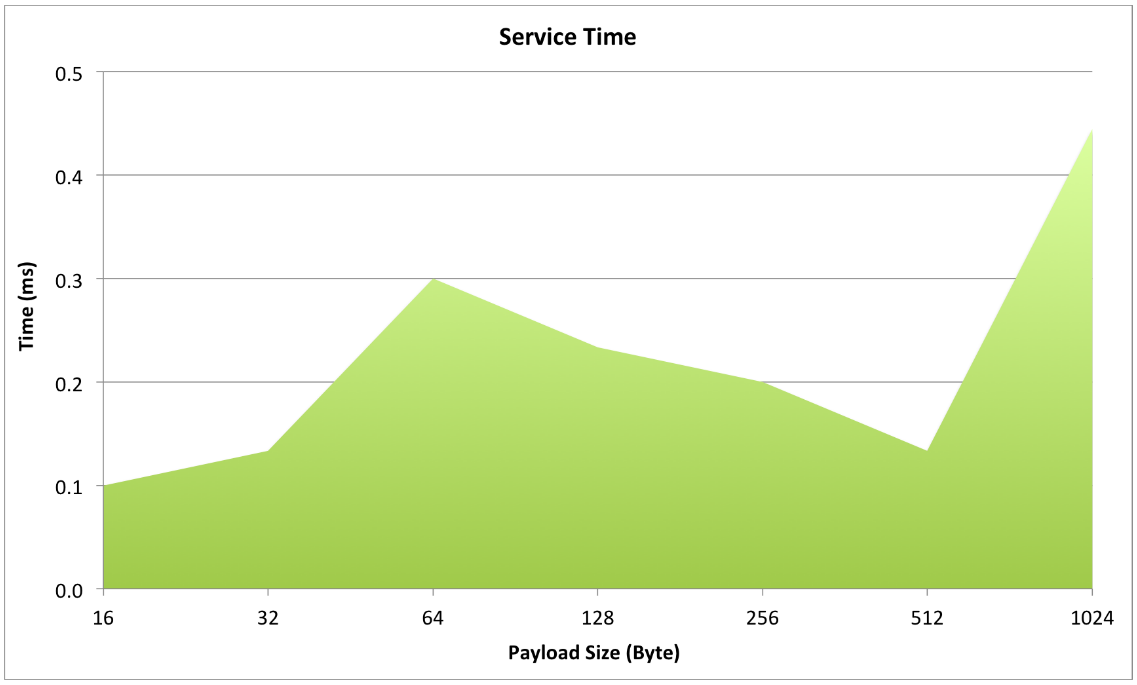

Figure 5.8: Subscription Service Time

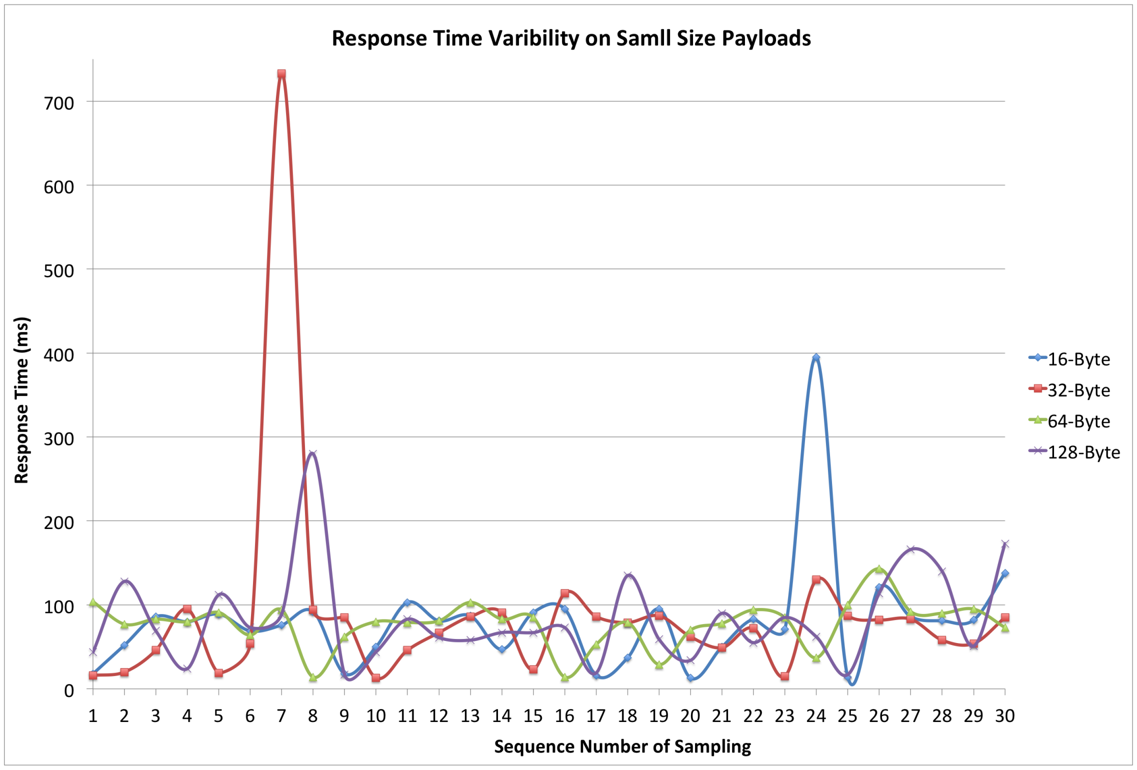

Figure 5.9: Response Time Variability on Small Size Payloads

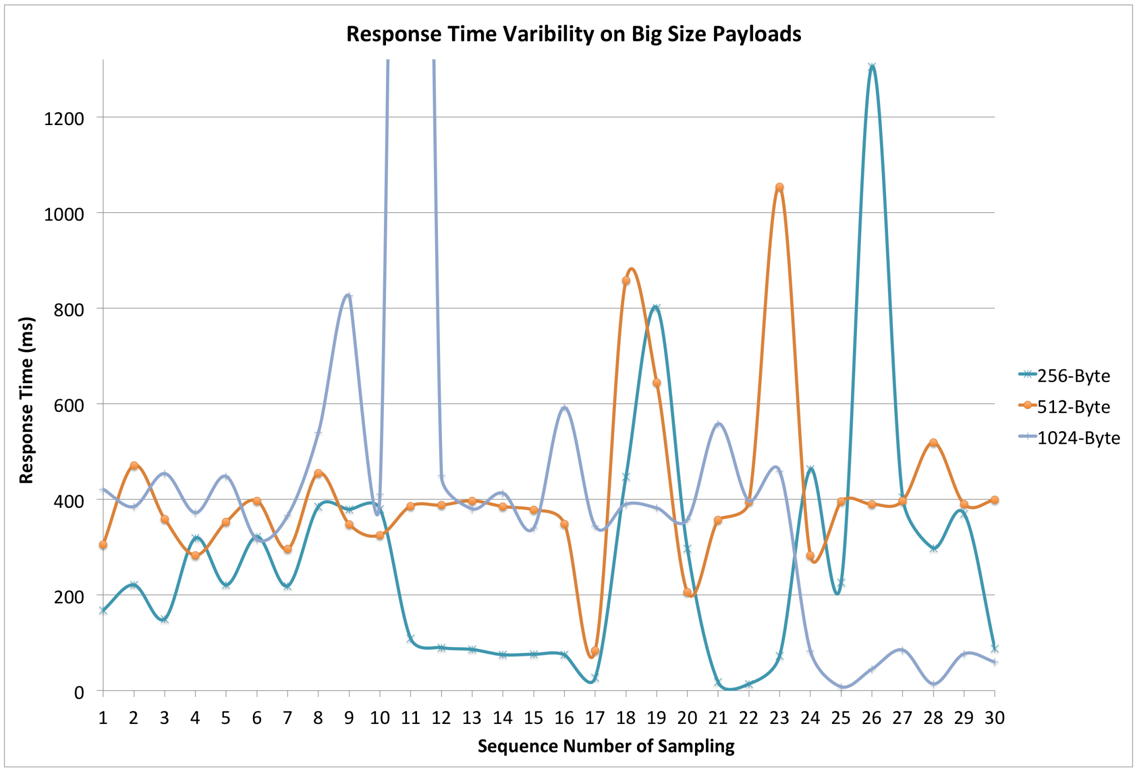

Figure 5.10: Response Time Variability on Big Size Payloads

introduced some extra overhead required to finish the request. The extra overhead could have come from memory allocation, network transfer, or their combination. If the application is sensitive to subscription request time, we need to choose a payload size that was less than and equal to 128 bytes. Figure 5.5 tells the same story as the subscription response time, although there is a bigger fluctuation. A 128 bytes payload was found to still be a magic turning point, and the throughput decreased since then. Is there any difference between first-time subscription (cold subscription) and a later subscription (warm subscription)? In the test, it showed that there is a big difference on service time (Figure 5.6.) The service time for the first subscription took 45ms but for a later subscription, service time was reduced to less than 0.5ms. The most interesting information presented in Figure 5.6 is the request time. What we expected was that the request time would be reduced when the service time was reduced. This reduction can be explained by individual sampling. Since the cold subscription is the very first subscription, and it is one-time event, the response time could not be accurate and it is lucky enough to run at a lower network delay. However, the service time was accurate on both measurements because it did not depend on any network activity, but only measured the elapsed time on the server. Figure 5.7 and 5.8 show the amount of time spent on network delays and how much time was spent on the server for an I/O subscription. The service times were very short and all were less than 0.5ms. These figures also show that the major overhead for subscription response time on a large payload size >128 bytes (Figure 5.4) came from the network delay. The last thing learned from the experiments was the variability in the response time. Figures 5.9 and 5.10 show such variability on small and big payload size, respectively. As we can see, a bigger payload size has more variability as compared to a small payload size. All 256, 512 and 1024 bytes payload had big variation during their repeated runs. In contrast, the small payload size (<256 bytes) kept the variability within 100 to 200ms. Only a few cases had an increased response time. Since a bigger payload size had a much higher variability, it could explain why a payload larger than 128 bytes had a noticeably increase in response time. Larger payloads can respond in faster time but with a higher variability than the average response time.

Public Network

Figure 5.11 shows that there is no difference on data transfer response time. All tests responded very fast and their range was from 1.15 to 1.3 ms. The maximum response time of 1.3ms still happened on a maximum payload size of 1024 bytes. The data transfer rate grew along with payload size growth, and the throughput ended up with 865.11 KB/s on a payload with a size 1024 bytes (Figure 5.12), which is a little bit faster than the maximum throughput on the dedicated network. Again, we can see that the I/O throughput approximately doubled as the payload size doubled. For the subscription response time, a 128 bytes payload size was still the turning point where the response time with bigger payload sizes increased in time (Figure 5.13.) The response time with >128 bytes payload was increased rapidly and ended up with 307ms at a 1024 bytes payload. The maximum response time was much faster than what we saw in the experiment over the dedicated network but the trend stayed the same except if the line shifted lower. Figure 5.14 shows a similar

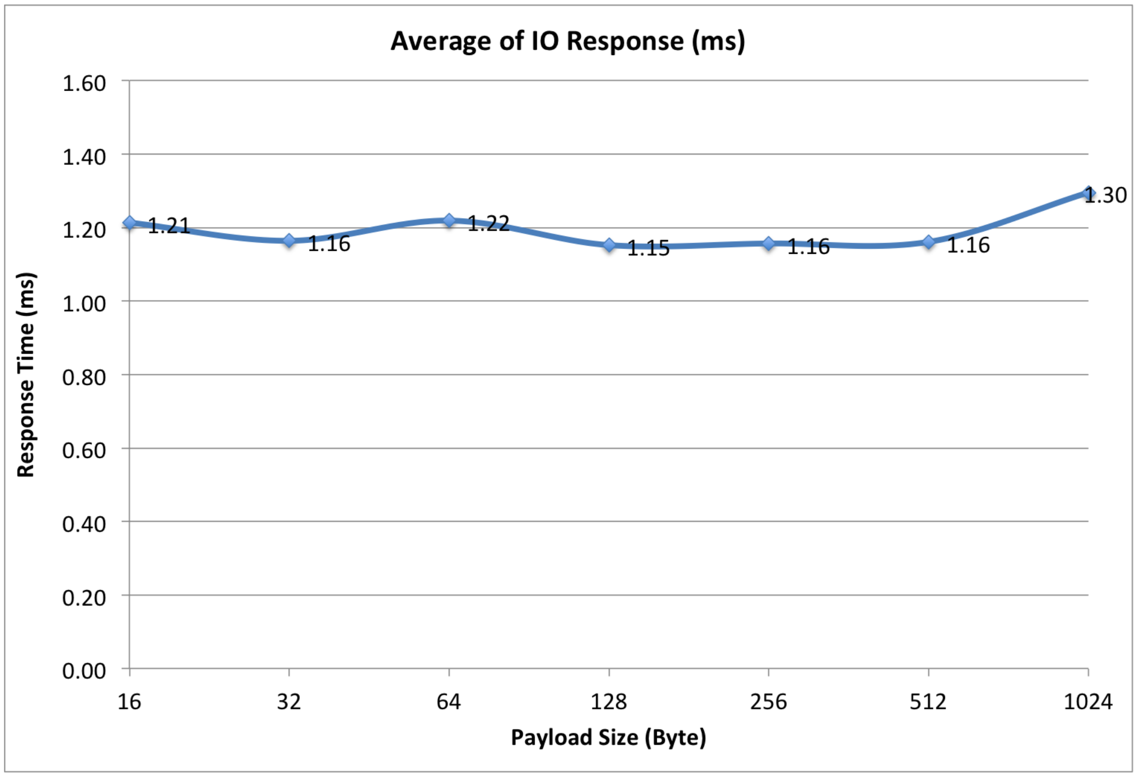

Figure 5.11: Public Network - Average IO Data Transfer Response Time

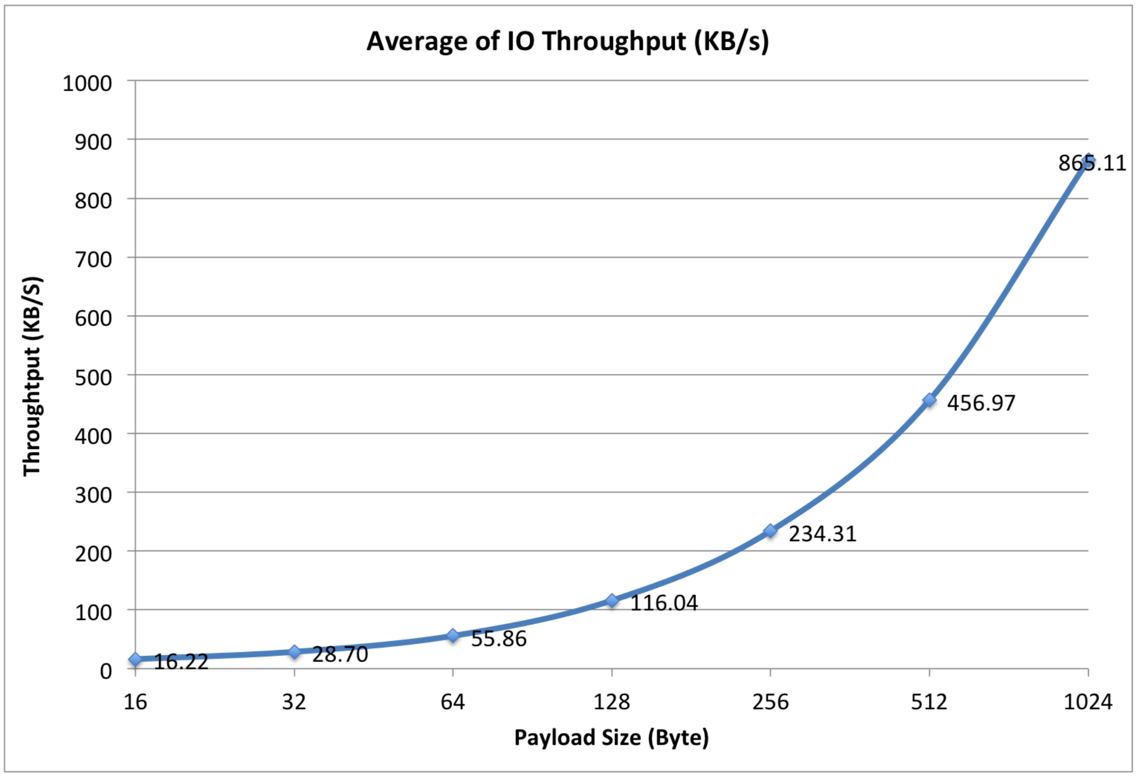

Figure 5.12: Public Network - Average IO Data Transfer Throughput

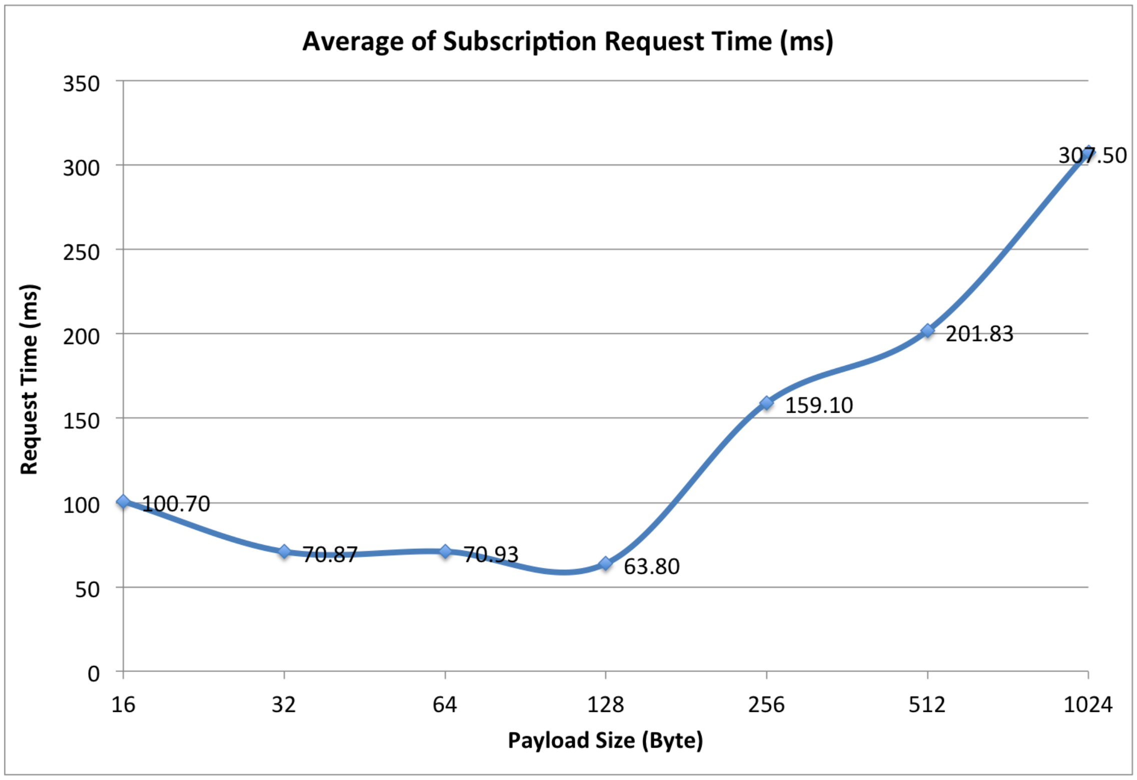

Figure 5.13: Public Network - Average Subscription Response Time

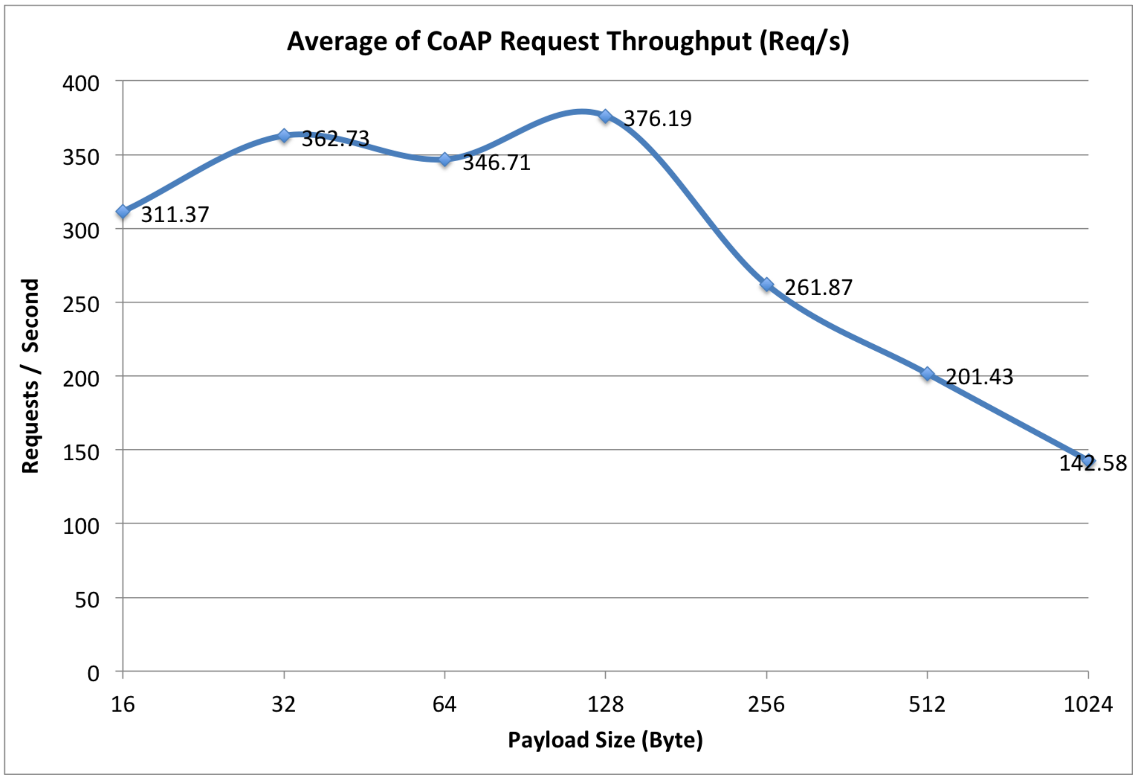

Figure 5.14: Public Network - Average Subscription Throughput

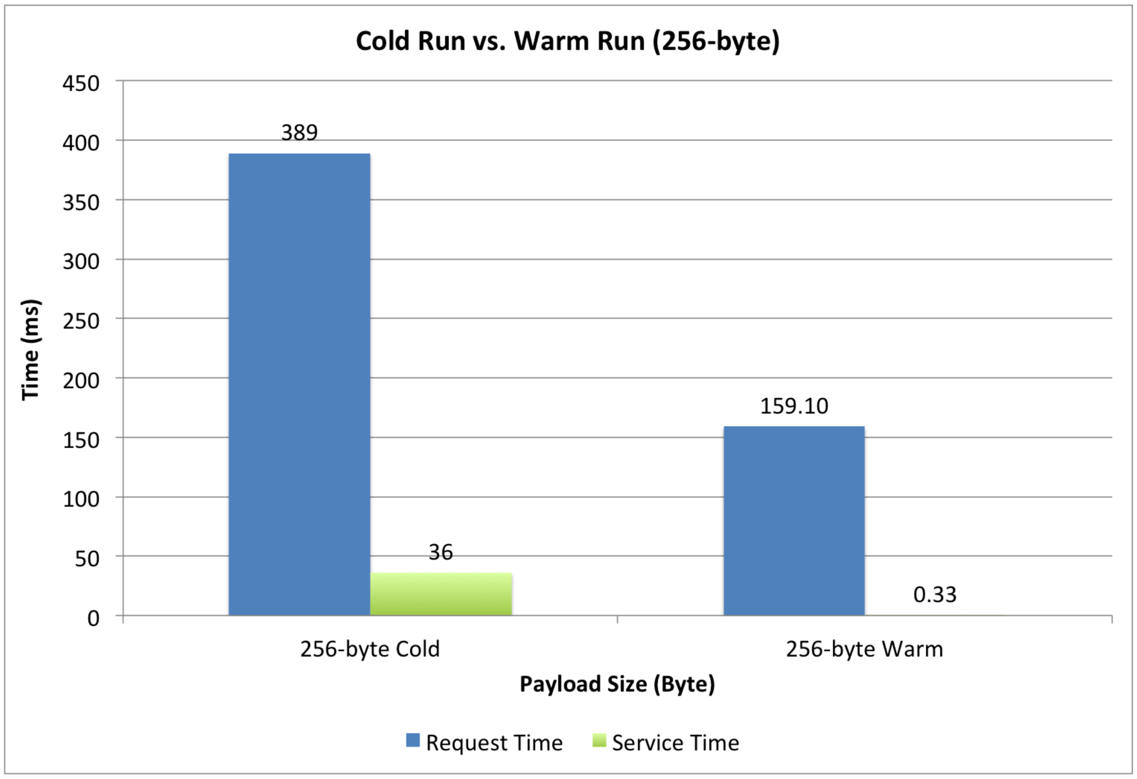

Figure 5.15: Cold Subscription versus Warm Subscription on Response Time

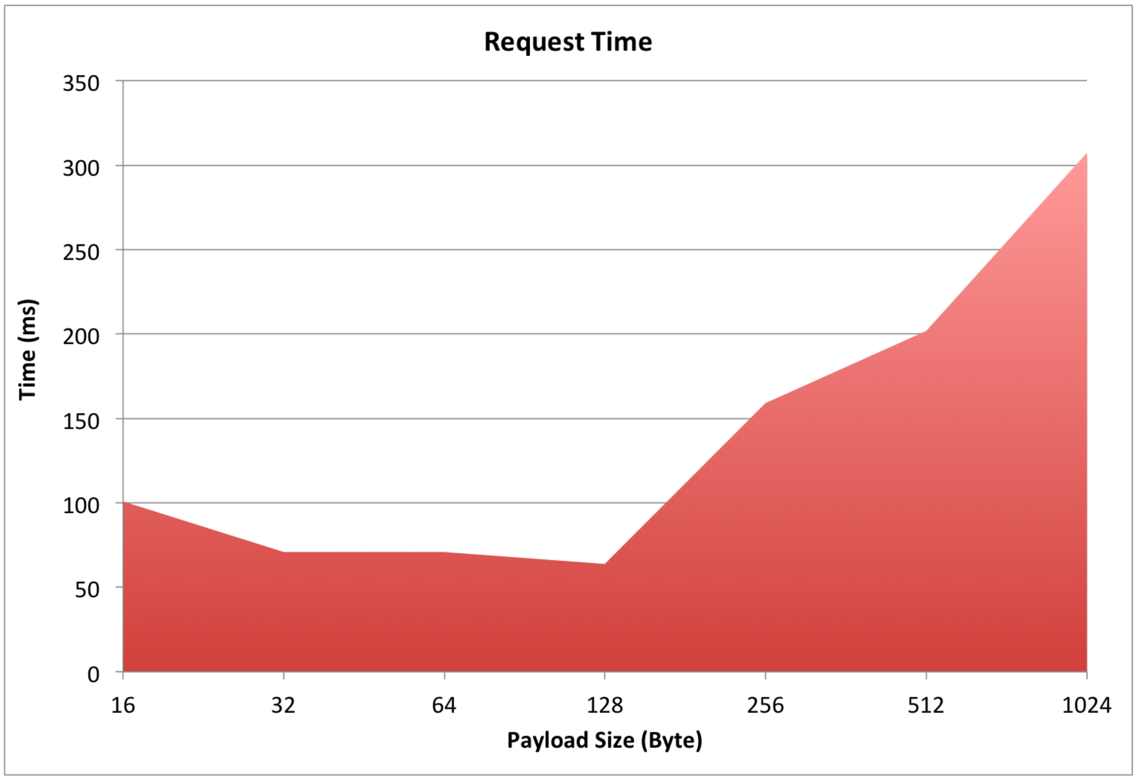

Figure 5.16: Subscription Response Time

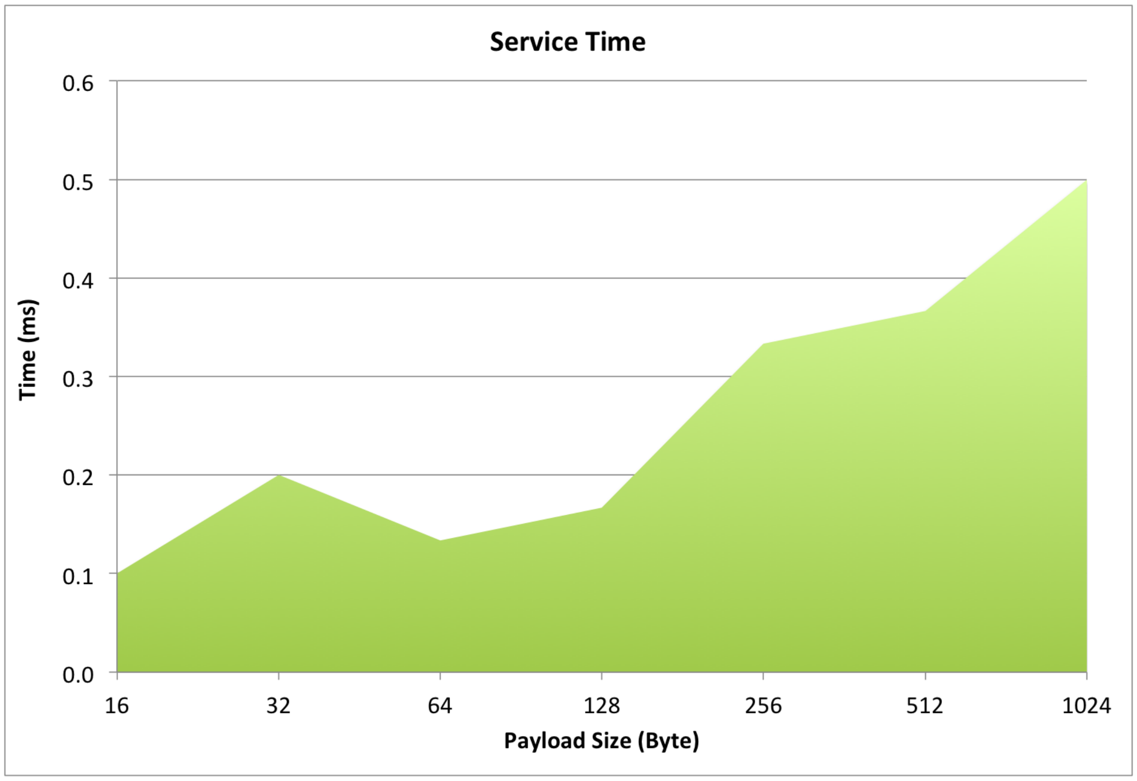

Figure 5.17: Subscription Service Time

story to the one that we saw in the subscription response time. The throughput increased a bit and this was expected because the response time was improved from the experiment on the dedicated network. In Figure 5.15, the cold and warm subscription comparison, we see the same story with a minor difference. The same result was on the service time. The service time in the cold subscription was 36ms and warm subscription was 0.33ms. The big service time gap was still there. The difference was in the subscription response time. This time we got 389ms for a cold subscription and 159ms for a warm subscription. This was the result that we expected since the service time got much shorter. However, in the previous comparison (Figure 5.6) on the dedicated network we saw an opposite result. As explained above, the comparison was based on the one-time event and the previous test does not get faster with a network delay. In the comparison of Figure 5.16 and 5.17, the request time was still far more than the service time. The results were similar to the previous comparison but had a faster request time. The maximum response time at a 1024 bytes payload was around 300ms as compared to more than 500ms before (Figure 5.7.)

Dedicated Network vs. Public Network

In all of the same sets of experiments conducted on the dedicated and public network, we can see that there was no significant behavior difference between the two types of networks. From an I/O performance perspective, both networks were able to respond in less than 1.5ms and the maximum throughput reached around 800KB/s at maximum 1024 bytes payload size. From a subscription perspective, all the results showed a 128 bytes payload size was a turning point. That is to say, the bigger payload size requires a much longer response time and lowers the throughput. No matter the network, we were able to achieve a minimum of 100 subscription requests per second, and a maximum of more than 330 requests per second. The only big performance difference between the two networks was in the public network. In the public network we got a slightly higher performance in terms of response time and throughput. It is suspected that the major factor that contributed to these difference was the router. The router used in the dedicated network was a regular home router. Although the exact model used in the Starbucks network is unknown, it should be a high-end router because it serves many people. High-end routers always have a more powerful processor, more internal memory, and more efficient scheduling algorithms in order to give a quicker response time to serve more online users. The different routers might be the reason why we saw a better result in the public network experiments. However, we did not see a big difference between the two, meaning that the performance was more sensitive to the device and its OS rather than network infrastructure. This sensitivity to the device and its OS rather than network infrastructure is why any maximum or boundary values were not emphasized, but instead focused on the level that the device could reach.

In summary, wireless network I/O performance using CoAP on a dedicated and a public network were examined. The prototype system can achieve quick I/O data response time of under 1.5ms regardless of payload size. The maximum throughput is around 800KB/s at the maximum payload size of 1024 bytes. The prototype system also achieves at least 330 subscriptions per second at its maximum performance. A 128 bytes payload size is the turning point on longer subscription response time and lower throughput. An application that is sensitive to response time needs to choose a payload size that is less than 256 bytes. Bigger payload size ($\geq$256 bytes) has a dramatically higher variance on response time, so a longer response time is sometimes expected.

Secured Communication

Security also plays a crucial role in networks. Compared to a traditional cloud computing environment, where system administrators manage and guard the server infrastructure 24x7, the IoT environment is more vulnerable because of its type of communication, operation locations, and management. In general, communication traffic is easily exposed to 3rd parties. Although a wired network makes its traffic harder to expose due to physical accesses, a wireless network is vulnerable to this threat. Exposing a wireless network's traffic does not even require physical access. If a 3rd party user sits close enough to a wireless network, he/she can pick up its signals and start listening. Security issues are critical to user privacy and sensitive data in a network environment, particularly for resource sharing. An easy and effective way to reduce such a threat is to encrypt all communications between both ends of the communication process. It is important for us to understand how CoAP performs in encrypted communications.

Except for the above performance tests, I/O throughput data on various payload sizes along with different encryption have also been measured. AES (Advanced Encryption Standard) was chosen as the encryption

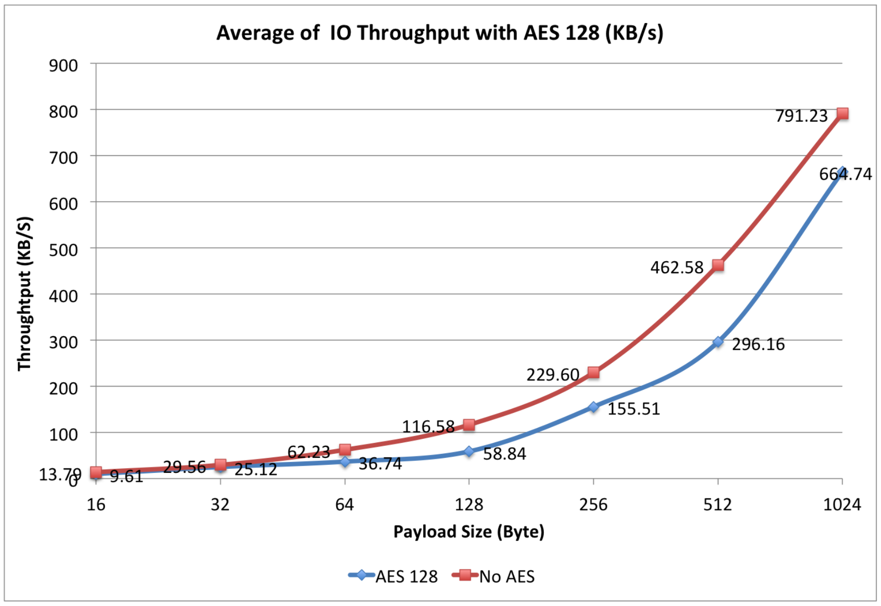

Figure 5.18: IO Throughput Comparison Between AES and non-AES

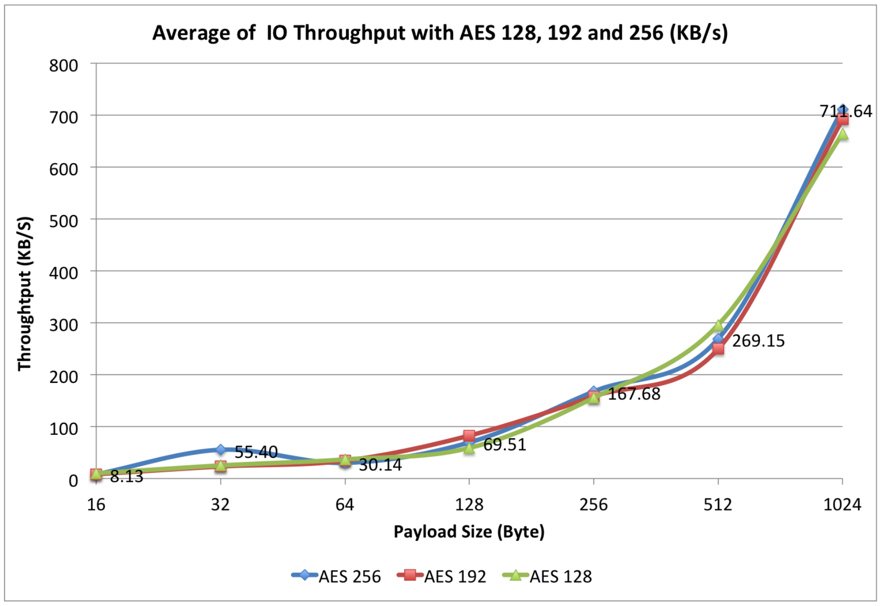

Figure 5.19: AES IO Throughput Comparison on Different Key Size

algorithm in all tests. The AES standard allows different key sizes for different purposes. A larger key size requires a longer time to encrypt and decrypt. This longer decryption time requires a 3rd party to spend more time in order to expose data. However, we also need more processor power to encrypt and decrypt. For a desktop computer system, more processor power is not a problem because it has a relatively powerful processor and unlimited power supply. This is a different story for a mobile device and both its processor power and battery life are limited. A part of the security tests was to identify a balance point between security levels and energy levels consumed. The exact same tests were run as the previous regular performance tests, except their communications were encrypted by AES. The tests with different AES key sizes from 128 bits to 256 bits were run in order to examine different CoAP performance behavior. Figure 5.18 shows that comparison of IO throughput on different payload sizes with and without AES 128-bit encryption. As the payload size grew, we saw the encrypted IO throughput become lower than the regular throughput. The largest gap appeared on the 512 bytes payload test. The regular throughput reached 462KB/s and the encrypted one was 296KB/s. So, in all cases we expected that any difference should be less than 170KB/s. If a smaller payload size was chosen, a closer throughput between the encrypted one and the regular one was expected. In Figure 5.19, IO throughput on the encrypted payloads by AES 128, 192 and 256 bits were measured. According to the diagram, there was not a big difference between the various key size tests. In fact, the key size tests almost grow in the same fashion. Typically we expected that there would be an IO throughput drop when a larger AES key size was applied. The reason why we did not see an I/O throughput drop happen in the tests was that CoAP payload size was relatively small (max. payload was 1024 bytes.) Thus, an encryption with a larger key size would not spend much more time to encrypt and decrypt as compared to a smaller key size.

Based on the results, we saw that there was IO throughput degradation when communications were encrypted by AES, especially on a larger payload size. However, the difference was less than 170KB/s. Meanwhile, there was no significant performance difference between the different AES key size tests because of CoAP small payload size. If an application was sensitive to data security, AES 256-bit was a better choice because the 256-bit key size made our communication more secure and importantly it has almost the same performance as the tests with the smaller key sizes. If we want a higher IO throughput for effective data, we need to choose a bigger payload size like 512 or 1024 bytes. The AES encryption had less of a throughput impact on a larger size payload in term of percentage of the encrypted communication throughput. However, if energy consumption was an important factor for an application, and AES 128-bit was a better choice because it protected our data at some levels, and produced the same performance as the larger key size encryption yet consumed less power energy in the long run.

5.6 Summary

The prototype system and its architecture were discussed in this chapter. Feasibility and effectiveness of the system has been examined via a group of experiments conducted on both the dedicated as well as the public network. This evaluation examined performance with regards to network I/O, resource subscription, and secured communication. Some of the key system performance, which includes transfer rate, response time, number of subscription and their variability, were examined. A summary of the results are shown in Table 5.2.

Experiment	Results
I/O Performance	1) Quick I/O data response time under 1.5ms regardless of payload size 2) The maximum throughput was around 800KB/s at the maximum payload size of 1024 bytes 3) For response sensitive applications, the payload size should be less than 256 bytes 4) A bigger payload size (≥256 bytes) has dramatically high variability on response time
Resource Subscription	1) The system can achieve at least 330 subscriptions per second at maximum 2) 128 bytes of payload size was the turning point on longer subscription response time and lower throughput
Secured Communications	1) I/O throughput degrades when encrypted; Larger payload size leads to larger degradation 2) No significant performance difference between different AES key size 3) For security sensitive applications, AES 256-bit is a better choice 4) For transferring larger effective data, a bigger payload size such as 512 or 1024 bytes is better 5) For energy consumption sensitive applications, AES 128-bit is a better choice

Table 5.2: Summary of Experimental Result

Chapter 6

Summary

Computers are indispensable in most aspects of our daily lives. Since they were invented, there have been many different generations of computing models that have been improved for better computation in terms of efficiency, utility, and application. The first generation of computing started the era of human automation computing. Due to various technology limitations at that time, all computational resources were completely centralized in local machines. Because of the appearance of networks, second generation computing significantly improved data availability and portability so that computing resources could be efficiently shared among the networks. The network applications allocate, utilize and collaborate wider distributed network resources in order to provide a single and compound service for users to consume. The service-oriented third generation of computing provides functionality by breaking down applications into services, on-demand computing through utility and cloud infrastructures and ubiquitous accesses from wide-spread geographical networks. Services, as main computing resources, are far spread from local to worldwide. Services loosely couple applications and servers which make services to easily scale up and with higher availability.

As the computational resources become more available, fault-tolerant, scalable, better performance, particularly spatial distributed, the complexity of locating, utilizing and optimizing computational resources becomes even more challenging. How applications can dynamically utilize and optimize unique/duplicate/competitive resources at runtime in the best efficient and effective way without code changes as well as providing high available, scalable, secure and easy development services becomes a critical question. Current computing models miss the capability of dynamically utilizing and redistributing I/O in distributed systems in a flexible way. They are also very hard to extend, group and connect functionality at runtime without changing codes. And current programming models are too complicated to develop and deploy new applications and evolve along quickly with time. Because of these challenges and constantly changing environments, resources and demands, a new flexible and dynamic computing model is needed to resolve these problems.

By solving such problems, this work contributes to the following items below:

1. Propose a new computational model, called Domain Computing

2. Build a prototype of the domain computing

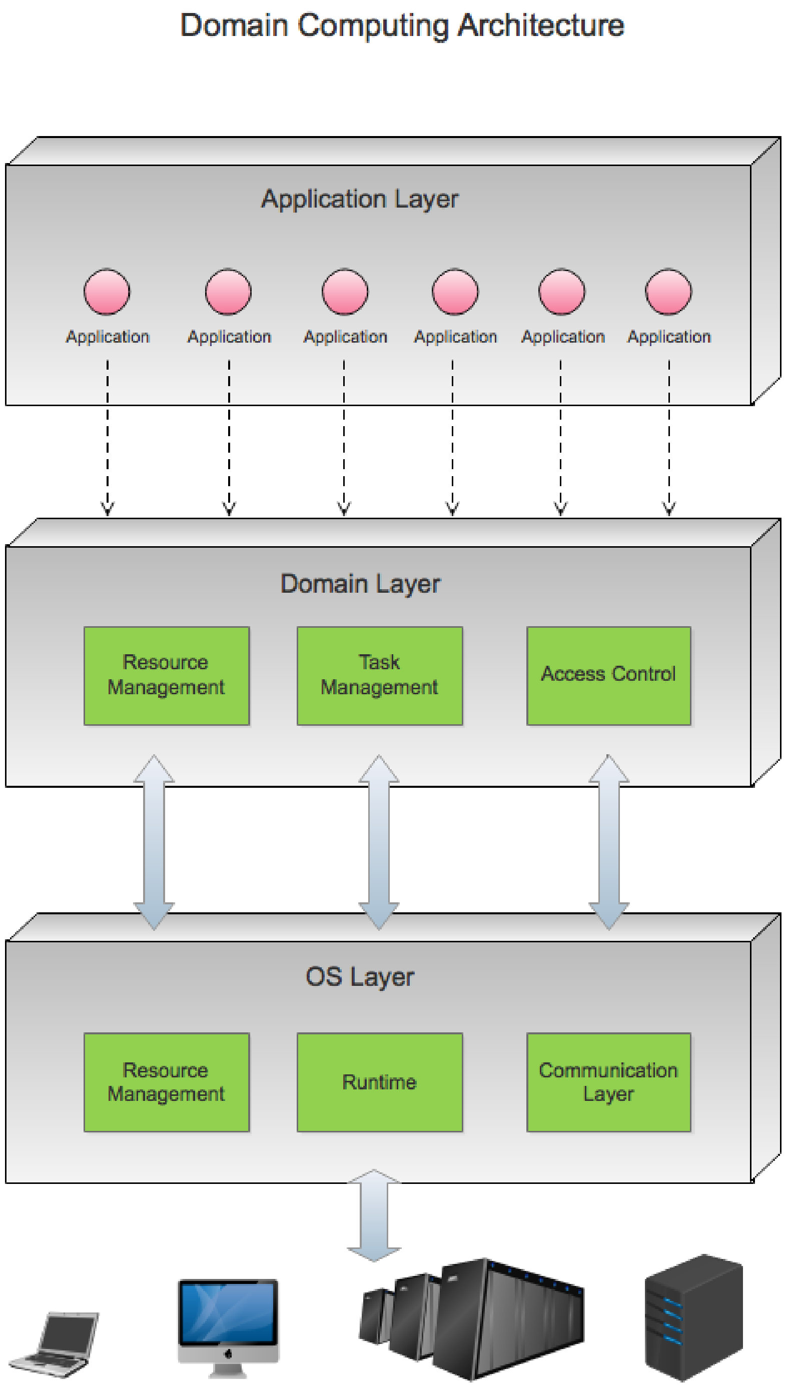

Figure 6.1: Domain Computing Architecture

3. Examine its feasibility and effectiveness of the conceived concept via some key performance including network throughput, response time, their variability, resource publishing and subscription and secured communications

Furthermore, this work has been published in several papers and journals Xue et. al [67] [68] [66].

Domain computing proposes a new way to manage computational resources and applications (Figure 6.1). It dynamically manages resources within logic entities, domains, and without bound to physical machines so that application functionality can be extended at runtime. Domain computing introduces domains as a replacement of the traditional computer in order to run applications and link different computational resources that are distributed over networks into domains so that users can significantly improve and optimize the resource utilization at a global level. By negotiating with different layers, domain computing dynamically links different resources, shares resources and cooperates with domains at runtime so that applications can quickly adapt to changing and dynamic environments and gain better performance. Additionally, domain computing presents a new way to develop applications that are resource stateless based.

An initial prototype system was built for evaluation. Some of the prototype's key system performance including network throughput, response time and their variance , were examined. The system can achieve a quick I/O data response time under 1.5ms, regardless payload size. The maximum throughput is around 800KB/s at the maximum payload size 1024 bytes. The application sensitive to response time needs to choose a payload size that is less than 256 bytes. Bigger payload size ($\geq$256 bytes) was found to have a dramatically high variance on response time, so a longer response time was sometimes expected. The IoT brings millions of resources together via heterogeneous networks and it is the future computing environment. Resource publishing and subscription as one of the fundamental mechanisms in IoT was also examined. The system can achieve at least 330 subscriptions per second at a maximum. However, a 128 bytes payload size is a turning point on a longer subscription response time and lower throughput.

Security always plays a crucial role in networks for users' privacy and their sensitive data. Because network communication is easily exploited, encrypted communication is necessary in many applications. There is IO throughput degradation when communications were encrypted with AES, especially on a larger payload size. However, the difference was less than 170KB/s and there was no significant performance difference between the different AES key size tests because of a CoAP's small payload size. If an application is sensitive to data security, AES 256-bit is a better choice because the 256-bit key size makes our communication more secure and, more importantly, it has almost the same performance as the tests with the smaller key sizes. If we want a higher IO throughput for the effective data, we need to choose a bigger payload size like 512 or 1024 bytes. The AES encryption has a less of an impact on throughput on a larger size payload in terms of percentage of the encrypted communication throughput. However, if energy consumption is an important factor for an application, AES 128-bit is a better choice because it protects our data in some levels and produces the same performance as the larger key size encryption but consumes less power energy in the long run.

Chapter 7
Future Work

The domain computing system architecture and its main components have been discussed. Multiple layers of the components work together and enable it to be able to manage and optimize resources within logic entities, domains dynamically, and without binding to physical machines so that application functionality can be extended at runtime. Domain computing also offers a new way to develop applications that are resource stateless based. Various aspects of performance have also been studied. However, the domain computing system architecture still is in a very early phase and there still is a lot of improvement and evaluation needed in order to release its full and potential functionality. In this chapter, some future works are discussed.

Domain computing involves changes on both system and network architectures. Although various network performance tests have been done, there are some features that have not been implemented in the prototype system. The security feature is one of them. The access control component is responsible for maintaining permissions for resources within a domain, and it provides the capability to control how a resource can be accessed. The following items in the security part need to be implemented.

- Ownership Management: maintain the ownership of each registered resource. A resource belongs to the domain that physically owns it, and that domain has ownership of the resource.
- Permission Management: maintain a permission list for each resource. As a service, a resource can be shared with other domains. Each resource maintains a list to check who and how the resource is accessed. A shared domain registers the resource as its resource with permissions and authentication information.
- Level Control: maintain a full list of control level for different accesses. Accesses can be managed based on different purposes like reading, writing, reading/writing, duplicating, and granting.
- Domain-Based Unit: unlike user-based control, the basic unit of an access is based on the domain. Since applications run on domains, different users have their domains instead of sharing the same domain. The permissions are granted for those domains. Domain-Based access control removes the physical bounds between users and systems. It allows users create as many domains as needed so that users can build complex relationships between their domains for isolation purposes.

- OS runtime: the access control also requires that OS layer identifies and registers the local resources to domains so that the resources can be used and shared with other domains. Tasks like permission grant and authentication are also done in the OS layer. When a domain sends a request to ask for access to a resource, it can accept and deny the request. Meanwhile when the domain has the permission and queries data, it checks if the request is valid with the proper credential.

Except for the security features, other features need to be implemented. In the OS layer, standard protocols, like runtime rerouting and resource permission request/grant, need to be developed.

- Rerouting Protocol: the domain computing relies on the underlying OS platforms and the platforms can be heterogeneous among Windows, Linux, and Unix. The rerouting protocol is used to reconnect data flows as required among resources at runtime. The design of the protocol should take into consideration the existing communication mechanisms and binary formats on various platforms so that it allows switching of the resources on any platform.

- Resource Sharing Protocol: a communication protocol for resource availability and permission negotiation. How to share a resource between domains needs to be designed. A security module should also be developed so that the system can have the ability to control resource access under proper permissions. How to authenticate a request, and access it in an efficient and effective way, needs to be examined and designed.

- Resource Interface: due to the heterogeneous platforms underneath, a common resource interface should be designed properly so that all the platforms can develop resources independently and hook them up at runtime, which significantly improves the resources and their code utilization. For example, in the resource management component, a standard resource interface including input and output format should be clearly defined.

Some system optimization and performance re-evaluation are needed as well.

- Optimization: as domain computing is a network system, local compilation and global optimization need to continue studying and improving based growing different applications.

- Performance: certainly, its overall system performance should be re-evaluated when any of the new features is added into the system. In addition, some performance characteristics like the overhead of linking resources in the local machine and also in remote will be interesting to study.

References

[1] Mohammad Aazam and Eui-Nam Huh. Fog computing and smart gateway based communication for cloud of things. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 464–470. IEEE, 2014.

[2] Gregory D Abowd and Elizabeth D Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):29–58, 2000.

[3] Richard M Adler. Distributed coordination models for client/server computing. *Computer*, 28(4):14–22, 1995.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. 2009.

[6] Edward A Ashcroft and William W Wadge. Lucid, the dataflow programming language. *APIC Studies in Data Processing, Academic Press*, 9, 1985.

[7] Kevin Ashton. That internet of things thing. *RFiD Journal*, 22(7):97–114, 2009.

[8] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[9] Thomas J Bergin. 50 years of army computing from eniac to msrc. Technical report, DTIC Document, 2000.

[10] Fran Berman, Geoffrey Fox, and Anthony JG Hey. *Grid computing: making the global infrastructure a reality*, volume 2. John Wiley and sons, 2003.

[11] Alex Berson. *Client/server architecture.* McGraw-Hill, Inc., 1996.

[12] Norbert Bieberstein. *Service-oriented architecture compass: business value, planning, and enterprise roadmap.* FT Press, 2006.

[13] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[14] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[15] James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Market-oriented grids and utility computing: The state-of-the-art and future directions. *Journal of Grid Computing*, 6(3):255–276, 2008.

[16] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1507–1542, 2002.

[17] Rajkumar Buyya and Amir Vahid Dastjerdi. *Internet of Things: Principles and Paradigms.* Elsevier, 2016.

[18] Rajkumar Buyya et al. High performance cluster computing: Architectures and systems (volume 1). *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999.

[19] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[20] Paul E Ceruzzi. A history of modern computing (history of computing). *Cambridge, Mass., London*, 21999, 2003.

[21] David Chappell. *Enterprise service bus.* " O'Reilly Media, Inc.", 2004.

[22] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *The knowledge engineering review*, 18(03):197–207, 2003.

[23] Cisco Visual Networking Index. Global mobile data traffic forecast update, 2013-2018. *White paper*, 2014.

[24] Om P Damani, P Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. One-ip: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN systems*, 29(8):1019–1027, 1997.

[25] Niccolò De Caro, Walter Colitti, Kris Steenhaut, Giuseppe Mangino, and Gianluca Reali. Comparison of two lightweight protocols for smartphone-based sensing. In *Communications and Vehicular Technology in the Benelux (SCVT), 2013 IEEE 20th Symposium on*, pages 1–6. IEEE, 2013.

[26] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[27] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[28] Thomas Erl. *Service-oriented architecture: a field guide to integrating XML and web services.* Prentice Hall PTR, 2004.

[29] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1, 2011.

[30] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, 2013.

[31] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure.* Elsevier, 2003.

[32] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. Ieee, 2008.

[33] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[34] Brian Hayes. Cloud computing. *Communications of the ACM*, 51(7), 2008.

[35] IDC. Forecasts worldwide public it cloud services spending to reach nearly $108 billion by 2017 as focus shifts from savings to innovation. *Press release*, 3, 2013.

[36] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[37] Hermann Kopetz. Internet of things. In *Real-time systems*, pages 307–323. Springer, 2011.

[38] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: service-oriented architecture best practices.* Prentice Hall Professional, 2005.

[39] Kalle Lyytinen and Youngjin Yoo. Ubiquitous computing. *Communications of the ACM*, 45(12):63–96, 2002.

[40] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.

[41] M Douglas McIlroy. A research unix reader: annotated excerpts from the programmers manual. 1971.

[42] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[43] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[44] Eric Newcomer and Greg Lomow. *Understanding SOA with Web services.* Addison-Wesley, 2005.

[45] Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302. ACM, 2007.

[46] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[47] Han Qi and Abdullah Gani. Research on mobile cloud computing: Review, trend and perspectives. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, pages 195–202. ieee, 2012.

[48] Michael A Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1):32, 2004.

[49] Peter Rodgers. Service-oriented-development on netkernel. *Web Services Edge East, Boston, USA*, 2005.

[50] Jeanne W Ross and George Westerman. Preparing for utility computing: The role of it architecture and relationship management. *IBM systems journal*, 43(1):5–19, 2004.

[51] Debashis Saha and Amitava Mukherjee. Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31, 2003.

[52] Mahadev Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7. ACM, 1996.

[53] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2001.

[54] Roy W Schulte. Enterprise service bus usage scenarios and product categories. *Gartner research [online]*, 2007.

[55] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (coap), 2014.

[56] Ivan Stojmenovic and Sheng Wen. The fog computing paradigm: Scenarios and security issues. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pages 1–8. IEEE, 2014.

[57] Liba Svobodova. Client/server model of distributed processing. In *Kommunikation in Verteilten Systemen I*, pages 485–498. Springer, 1985.

[58] Lu Tan and Neng Wang. Future internet: The internet of things. In *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 5, pages V5–376. IEEE, 2010.

[59] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[60] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36. ACM, 2012.

[61] Feng Wang, Liang Hu, Jin Zhou, and Kuo Zhao. A survey from the perspective of evolutionary process in the internet of things. *International Journal of Distributed Sensor Networks*, 2015:9, 2015.

[62] Rolf H Weber and Romana Weber. *Internet of Things*, volume 12. Springer, 2010.

[63] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[64] Mark Weiser, Rich Gold, and John Seely Brown. The origins of ubiquitous computing research at parc in the late 1980s. *IBM systems journal*, 38(4):693, 1999.

[65] Michael R Williams. *A history of computing technology*. IEEE Computer Society Press, 1997.

[66] Yi Xue and Ralph Deters. Resource sharing in mobile cloud-computing with coap. *Procedia Computer Science*, 63:96–103, 2015.

[67] Yi Xue and Ralph Deters. Towards horizontally scalable apps. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–9, 2016.

[68] Yi Xue, Richard K Lomotey, and Ralph Deters. Enabling sensor data exchanges in unstable mobile architectures. In *2015 IEEE International Conference on Mobile Services*, pages 391–398. IEEE, 2015.

[69] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

[70] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[71] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[72] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.