FEASIBILITY STUDIES OF STATISTIC MULTIPLEXED COMPUTING

A Dissertation
Submitted to
the Temple University Graduate Board

In Partial Fulfillment
of the Requirements for the Degree
DOCTOR OF PHILOSOPHY

---

in Computer and Information Sciences

by
Yasin Celik
August 2018
Temple University

Examining Committee Members:
Professor Justin Yuan Shi, Computer and Information Sciences, Temple University, Advisory Chair
Professor Jie Wu, Computer and Information Sciences, Temple University
Professor Krishna Kant, Computer and Information Sciences, Temple University
Professor Chiu Tan, Computer and Information Sciences, Temple University
Professor Boleslaw K. Szymanski, Computer Science, Rensselaer Polytechnic Institute

2018

ABSTRACT

Feasibility Studies of Statistic Multiplexed Computing

By

Yasin Celik

In 2012, when Professor Shi introduced me to the concept of Statistic Multiplexed Computing (SMC), I was skeptical. It contradicted everything I have learned and heard about distributed and parallel computing. However, I did believe that unhandled failures in any application will negatively impact its scalability. For that, I agreed to take on the feasibility study of SMC for practical applications.

After six+ years research and experimentations, it became clear to me that the most widely believed misconception is "either performance or reliability" when upscaling a distributed application. This conception was the result of the direct use of hop-by-hop communication protocols in distributed application construction.

Terminology:

Hop-by-hop data protocol is a two-sided reliable lossless data communication protocol for transmitting data between a sender and a receiver. Either the sender or the receiver crash will cause data losses. Examples: MPI, RPC, RMI, OpenMP.

End-to-end data protocol is a single-sided reliable lossless data communication protocol for transmitting data between application programs. All runtime available processors, networks and storage will be automatically dispatched to the best effort support of the reliable communication regardless transient and permanent device failures. Examples: HDFS, Blockchain, Fabric and SMC.

Active end-to-end data protocol is a single-sided reliable lossless data communication protocol for transmitting data and automatically synchronizing application programs. Example: SMC (AnkaCom, AnkaStore (this dissertation)).

Unlike the hop-by-hop protocols, the use of end-to-end protocol forms an application-

dependent overlay network. An overlay network for distributed and parallel computing application, such as Blockchain, has been proven to defy the "common wisdom" for two important distributed computing challenges:

a) Extreme scale computing without single-point failures is practically feasible. Thus, all transaction or data losses can be eliminated.

b) Extreme scale synchronized transaction replication is practically feasible. Thus, the CAP conjecture and theorem become irrelevant.

Unlike passive overlay networks, such as the HDFS and Blockchain, this dissertation study proves that an active overlay network can deliver higher performance, higher reliability and security at the same time as the application up scales.

Although application-level security is not part of this dissertation, it is easy to see that application-level end-to-end protocols will fundamentally eliminate the "man-in-the-middle" attacks. This will nullify many well-known attacks. With the zero-single-point failure and zero impact synchronous replication features, SMC applications are naturally resistant to DDoS and ransomware attacks.

This dissertation explores practical implementations of the SMC concept for compute intensive (CI) and data intensive (DI) applications. This defense will disclose the details of CI and DI runtime implementations and results of inductive computational experiments. The computational environments include the NSF Chameleon bare-metal HPC cloud and Temple's TCloud cluster.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First and foremost I want to thank my advisor Dr. Justin Yuan Shi. I would like to express my sincere gratitude to my advisor for the continuous support of my Ph.D study and related research. It is due to your immense knowledge, valuable guidance and ever-friendly nature that I am able to complete my Ph.D study. Your unwavering enthusiasm and excitement in research will continue to be a mentor to me through my future career. It has been an honor and I have been truly fortunate to be able to work with you. Your motivational speeches and patience helped me a lot during tough times and I am very grateful for that. I am also thankful for the excellent example you have provided as a successful scientist and a professor.

I would also like to take this opportunity to thank my other committee members, Dr. Jie Wu, Dr. Krishna Kant, Dr. Chiu Tan and Dr. Boleslaw K. Szymanski to agree serving as my dissertation oral defense external reviewer.

I would also like to thank Aakash Pradeep and Moussa Taifi, former fellow CIS graduate students under Dr. Shi's supervision. They have been a source of friendships as well as good advice and collaboration. In addition, I would like to thank Alparslan Sari, Hasan Huseyin Eruslu, Mustafa Zengin for helpful comments and motivational discussions and for their friendship.

I would like to thank my family: my parents, my brothers and sisters and especially my dear wife, Halise Celik, for her positive attitude towards difficulties both in my life and throughout this study. Her love, sacrifices, patience and support helped me to finish this degree. I would also like to thank my daughters, Isra and Zeynep, for being nice kids.

LIST OF FIGURES

Page

Chapter 1

BACKGROUND AND MOTIVATIONS

1.1 Moore's Law

In 1965 realizing the potentials of shrinking circuit size in silicon, Gordon Moore predicted that the number of transistors in a single chip would double every two years for next decade in [4]. Shrinking circuit sizes brought faster and cheaper chips, his prediction was for 1965-1975. In fact it remained valid until 2010s. As the manufacturing processes are approaching the physical limits of circuit miniturisation, the Moor's Law is starting to slow down in the second decade of 21^{st} century. Although some vendors continue to be optimistic about Moore's Law that is evolving and will continue to be current with new materials [5], the needs for computer processing have long outstripped processor capability improvements. Almost every successful computer application requires more than a single computer. In 2012, a 21st century computer architecture inflection point was claimed using existing hardware and software technologies [6].

1.2 Terminologies

Program-to-program communication protocol decides a distributed or parallel application's architecture.

A hop-by-hop data protocol is a two-sided reliable lossless data communication protocol for transmitting data between a sender program and a receiver program. Either the sender or the receiver crash will cause data losses and application crash. For example, RPC, RMI, MPI and OpenMp [7, 8, 9, 10] rely on two-sided hop-by-hop communication.

An end-to-end data protocol is a single-sided reliable lossless data communication protocol for transmitting data between application programs. All runtime available processors, networks and storage will be automatically dispatched to the best effort support of the reliable communication regardless transient and permanent device failures. For example, HDFS, Blockchain, Fabric and SMC [11, 12, 13, 1] rely on end-to-end principles.

An active end-to-end data protocol is a single-sided reliable lossless data communication protocol for transmitting data and automatically synchronizing application programs. In this dissertation, we present two systems, namely AnkaStore and AnkaCom, to show feasibility of SMC which applies single-sided communication and end-to-end principles.

1.3 The Scalability Dilemma

Different communities interprets "application scalability" in different ways. It is interesting to observe that the simple application performance and reliability requirements got "moulded" into different shapes and forms for different communities. Application performance and application reliability are the two very basic requirements, which should not be relaxed in order to deliver desired results for any large scale computer applications. However,

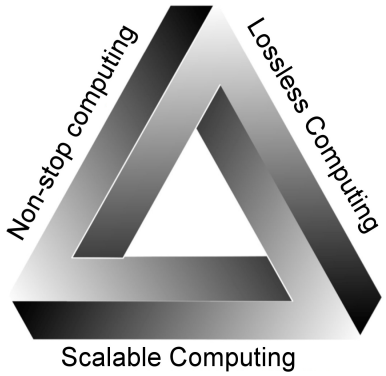

Figure 1.1: The impossible triangle [1].

presently it has been accepted that an application can only deliver either better performance or better reliability when the problem size grows and the underlying computing infrastructure expands. Consequently communities have focused on more performance when talking about the scalability term by relaxing reliability, which brought "performance-only scalability."

A closer look into this parallel application scalability difficulty reveals that direct use of hop-by-hop protocols in building a distributed application can result in a grid lock of impossibilities [14].

Figure 1.1 illustrates the impossible triangle formed by connecting three practically desirable but theoretically impossible requirements: "Non-stop computing", "Scalable Computing" and "Lossless Computing". The word "impossible" was derived directly from the theoretical impossibility studies using hop-by-hop protocols [15].

The early circuit-switching data communication networks endured similar scalability challenges if the word "computing" is replaced by "communication". Paul Baran proposed packet switching network (or statistic multiplexing) in 1960's [16] as a new data communication paradigm in order to deliver scalable data communication network. Although the packet switching network idea was not accepted when it was first proposed, the study of end-to-end caretaking system design [17] has made it the fundamental data communication standard for today's communicating systems. The internet would not be possible with circuit switching

methods. The packet switching network has proven the feasibility of the impossible triangle by eliminating all single-point failures enabling the ultimate solution using massively many unreliable components between any two communicating components [18].

However, direct use of the two-sided hop-by-hop protocol in building a distributed and parallel application forms an application-level overlay network that contains massively many single-point failures. Any component failure can crash the application. This violated the end-to-end system design principle. As the impossibility theory indicated, reliable failure detection is also impossible. Thus, unless the application is rebuilt using different protocols, the application scalability dilemma is not avoidable.

On the other hand, an application-level single-sided protocol can implement end-to-end caretaking. The Blockchain protocol is such an example [12].

Unlike hop-by-hop protocols, the use of end-to-end protocol forms an application-level overlay network that has zero single-point failures. The computer application can continue regardless processor and network failures. The Blockchain experiments have also proven transaction replication at scale is practical. It made the CAP conjecture and theorem [19] irrelevant.

However, the Blockchain protocol sacrificed end-to-end best-effort computing for decentralized authority by using cryptographic mining algorithm.

End-to-end distributed computing performance protocol has been a long standing illusive design objective until the discovery of active end-to-end Tuple Space protocols [20]. An active end-to-end single-sided protocol enables program-to-program communication and synchronization at the same time. Thus it enables automatic parallel component optimization and overhead amortization at the global scale – a benefit that is inaccessible to programs built using hop-by-hop protocols.

In this dissertation, parallel application scalability is defined as the application's ability to deliver incrementally better performance and reliability at the same time as we up scale the processing infrastructure. It is easy to see that, extreme scale computing is not possible for applications that do not scale.

1.4 Compute Intensive Application Challenges

Compute intensive applications aim to solve scientific problems for various disciplines including biological, chemical, astronomical, engineering, meteorological calculations and simulations. These applications can typically run for from minutes to days using from hundreds of servers to tens of thousands. As the size and complexity of problem increase, the need for hardware using to compute the problem increases as well. Increasing the size of hardware increases the probability of application crash proportionally, which result in the difficulty of the application sustainability. The application using check pointing as a fault tolerance mechanism to eliminate data loses in case of crashes spend more time on check pointing data and program state accordingly.

The fundamental flaw is the direct use of hop-by-hop communication paradigms. These paradigms depend on the assumption that when a sender sends a message to a receiver, the application logic always presume that both sender and receiver sides are 100% reliable. No hardware and software manufacturing processes can guarantee such assumption. Therefore, most of applications built on top of this logic periodically check point the program and data state in order to recover this states from last successful check point in case of crashes. The biggest challenge here is the time and energy spent on check point restart (CPR) process. As the size of problem and the processing infrastructure grows in size, the energy and time waste increases. At exascale, CPR can easily consume more time than the application itself.

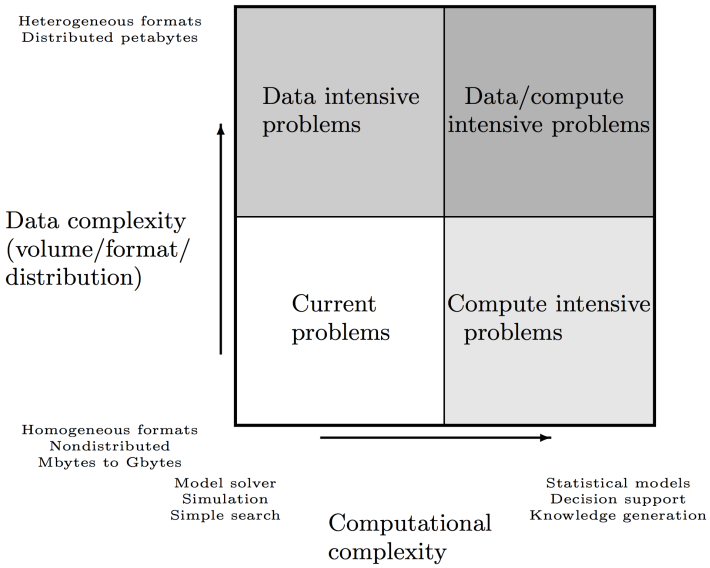

Figure 1.2: Emerging trends for data and compute intensive applications [2]

With the increasingly longer running times, large applications become increasingly more difficult to reproduce. Scientific computations lose credibility without reproducibility [21].

1.5 Data Intensive Application Challenges

Using direct hop-by-hop protocol, such as RPC (remote procedure call), for distributed data storage presents similar scalability problems. The application-level storage overlay network can be easily broken by any component failure. Data will be lost if unprotected.

Currently, there are two data replication protocols: synchronous and asynchronous. The synchronous data replication employs a two-phase commit (2PC) protocol. It requires immediate rollback if either the primary data source or the replication target cannot complete the replication request. The synchronous replication protocol has a severe performance penalty that is currently only used in low traffic applications.

Asynchronous data replication employs either an explicit or implicit replication queue between the primary data source and the replication target. Crash of the replication queue or

any of the servers can result in permanent data losses.

Expanding the data store requires more storage servers. The same scalability dilemma persists for storage systems built using hop-by-hop protocols.

Fig. 1.2 shows the transitions between data and compute intensive problems in terms of data and computational complexity. The recent shift paradigm in HPC community shows that there is a transition in HPC application architectures such MapReduce [22] to satisfy requirements for building new processing environments in order to deal with big data as well as the computation that requires big data. This new paradigm even brings more problems such as ones mentioned above.

1.6 OSI & Fallacy

Assuming "the network is reliable" was considered the first fallacy in distributed computing [23]. It is rather confusing to read the OSI documentation to find that hop-by-hop protocols are reliable, lossless data transmission protocol between any two points [24].

A closer look of the existing protocols, such as MPI (Message Passing Interface), RPC (Remote Procedure Call), OpenMP (share memory) and RMI (Remote Method Invocation), found that they are all hop-by-hop protocols from the same communication paradigm. While the data communication community is well served by the hop-by-hop protocols, direct use of the hop-by-hop protocols in distributed computing are fallacies since they fail application-level end-to-end protection by assuming the application-level overlay network is reliable [25].

Direct use of hop-by-hop protocols in computing applications leaves massively many potential end-point crashes in the application figure 1.3. Although the probability of each end-point crash is very small, the growth in application size (thus the processing infrastructure) is

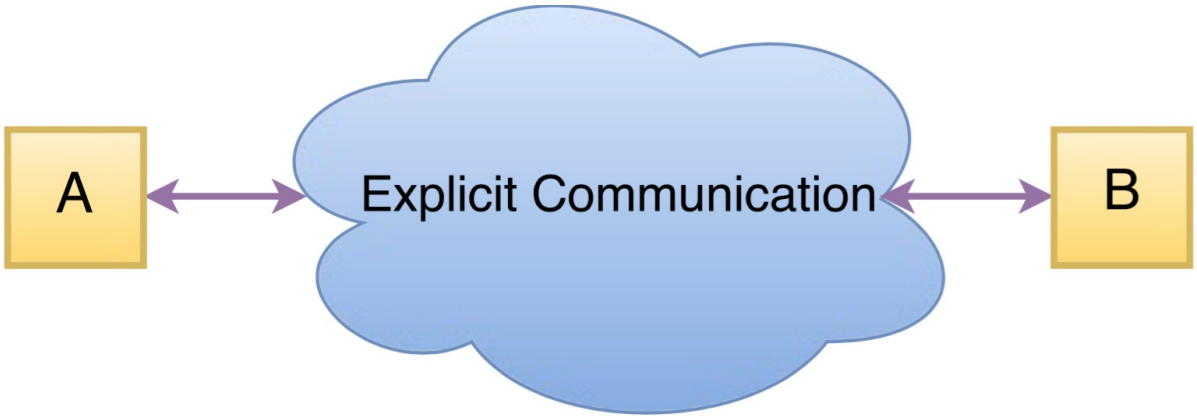

Figure 1.3: Hop-to-hop communication paradigm

guaranteed to increase the planned and unplanned application downtimes and data loss possibilities. It is reported in [26] that MTBF (Mean Time Between Failures) of cluster components is rapidly declining to sub hours for large distributed systems.

1.7 Impossibility & Possibility Theories

A reexamination of the 1993 proof of the impossibility of implementing reliable communication in the face of crashes [27] exposed the root cause of the alleged fallacy: the robust communication protocols are ineffective when either the sender or the receiver in the end-to-end communication could crash arbitrarily. No error detection and recovery methods could reverse this impossibility.

On the other hand, Spinelli gives a formal proof, in his Ph.D. thesis [18], that reliable communication is possible given unbounded resources. Except for practical IP-addressing issues, the Internet architecture has no scalability limits. This means that its architecture allows 100% reliable communication as long as it affords the minimal survivable resource set (MSRS) for every request. The strategically combined use of both packet-switching and circuit-switching protocols is the basic building block for the ultimate Internet architecture

we enjoy today.

The theory of Statistic Multiplexed Computing (SMC) employs both above results by leveraging an active end-to-end single sided protocol design.

1.8 CAP Theorem

As the size of "Big Data" started to grow in large amounts in 1990s, the extreme scale computing for this amount of data has become one of the most important necessities for several areas, such as intelligent business decission making, security, defense, social media and many more. Following ACID (Atomicity, Consistency, Isolation, Durability) properties when processing "big data" was one of the main obstacles that prevented HPC applications expansions. BASE-paradigm (Basically Available, Soft state, Eventual consistency) sematics proposal did not solve the scalability problem because as stated in 1.3, consistency cannot be relaxed in order to deliver "through scalability" for most of the distributed applications.

CAP theorem started appear in 1990s proposed by Eric Brewer [28], [29]. Eric Brewer proposed the CAP as a conjecture in his keynote speech at the Symposium on Principles of Distributed Computing [30]. CAP basically states that it is impossible to deliver *Consistency*, *Availability*, and *Partition tolerance* at the same time for distributed data stores when there is a failure in the system. In 2002, an informal proof for CAP theorem derived from CAP conjecture by Gilbert & Lynch [19].

Although many people believe that you can get "2 out of 3 (CAP)", it is now realized that this assumption is wrong. "2 out of 3" means that CA is also possible in case of network partitions. However, it is impossible to deliver C (Consistency) among replicas if two or more replicas cannot communicate with each other. Thus, CA is not possible.

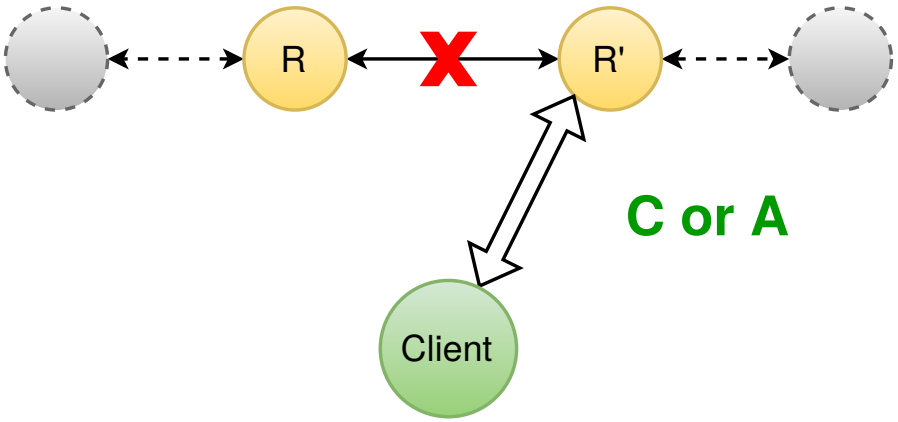

Figure 1.4: One of C or A has to be picked by application logic if a network in a distributed system partitions.

Only CP by forfeiting A and AP by forfeiting C are possible options for distributing systems. Figure 1.4 illustrates a network partition in a distributed system. In this case, the application logic can either continue with the available node using possibly stale data by relaxing consistency among data replicas, or stop the service by returning an error code making the system not available. Application logic has to be built according to service requirements. CA options is not possible if there is partition in the network since C implies that all the nodes are available and can communicate each other. Thus, it is impossible to deliver CA if there is a network partition among nodes in a distributed system that shares data.

The experiments of the Blockchain protocol have proven the practical feasibility of consistent transaction replication at scale. Although the CAP Theorem remains correct, they demonstrated that properly designed overlay networks can indeed make CAP irrelevant.

1.9 Single Point of Failure

Many distributed systems suffer from the single point failure (SPOF). The SPOF problem becomes more visible as a distributed system with one or more centralized components and grows in size. Since the probability of failure of any software or hardware component is

greater than zero, any system that has a SPOF is in danger of crashing any time because the failure of any of its parts prevents the entire system to operate. A HPC application can be considered as always available if and only if all of its components are replicated including data, network, software, or any other part of the system. a SPOF may also create bottlenecks for the system performance. A distributed system has to meet the requirement of no SPOF in its architectural design in order to meet scalability requirement.

As discussed earlier, it is impossible to eliminate all SPOF if the application is built using hop-by-hop protocols.

1.10 Overlay Network and Decentralized Architectures

The use of end-to-end protocol forms an application-dependent overlay network without any centralized controllers. A properly designed active overlay network for distributed and parallel computing application can meet three important distributed computing challenges:

a) Extreme scale computing without single-point failures is practically feasible. Thus, all transaction or data losses can be eliminated.

b) Extreme scale synchronized transaction replication is practically feasible. Thus, the CAP conjecture and theorem become irrelevant.

c) Incrementally better performance until the economic law of diminishing returns.

Unlike passive overlay networks, such as the HDFS ([11] and [31]) and Blockchain [12], this dissertation study proves that an active overlay network can deliver higher performance, higher reliability and security at the same time as the application up scales.

1.11 Structure of Dissertation

This dissertation is an implementation and experimentation study of the proposed SMC framework. The key focuses are a) design and implement a feasible runtime system and protocol library for feasibility studies, b) performance and reliability experimentation against existing benchmarks. The rest for this dissertation is organized in three parts. In the first part we discuss statistic multiplexed computing principle. We give feasibility studies in the second part which shows the proof of SMC concepts in data and compute intensive application domains.

- Part 1: In the first part, we discussed introduction, background, and motivations about this thesis.

- Part 2: This part is divided into three chapters. The first chapter explains statistic multiplexed computing architecture, background, motivation, assumptions, and basic requirements. And it concludes with a complete picture of SMC as a solution for distributed and parallel applications.

  The second chapter in this part discusses single-sided communication paradigm application for SCM principle. The single-sided paradigm is one of the necessities for extreme large scale HPC applications in order to deliver expected performance and scalability. It concludes with computational experiment results to prove that single-sided paradigm is feasible for HPC applications and does not degrade the performance although it has extra overhead costs.

  The third chapter discusses Unidirectional Virtual Ring (UVR) concepts in depth by giving the main ideas behind the underlying components. We also describe implementation details of each component including $NodeManager$, $Tuple$, $TupleSpace$, ring forming and maintenance. In addition, we also examine failure scenarios and consequently actions taken to maintain correct and complete the single UVR image

structure.

- Part 3: This part gives two feasibility studies of SMC architecture. The first study discusses AnkaStore, an implementation of SMC feasibility for data intensive computing applications. AnkaStore is also a proof of SMC concepts which shows reliable data store is possible given minimal survival resource set. AnkaStore is complete decoupled, decentralized, fault tolerant, and zero single point of failure system.

  The second application is AnkaCom for compute intensive applications. AnkaCom implements parallel and distributed Tuple Addressable Network that provides reliable communication service for compute intensive applications. AnkaCom is complete decoupled, decentralized, fault tolerant, and zero single point of failure system that provides reliable tuple space.

Chapter 2

STATISTIC MULTIPLEXED COMPUTING ARCHITECTURE

2.1 Background

Extreme scale HPC applications can be unsteady when failures are not handled correctly. Transient and permanent failures increase proportionally as the size of the processing infrastructure expands. The consequences of unhandled failures may result is incomprehensible crashes causing time, money, and human lives. In 2012, it is reported that the primary factor amongst all factors is the wide-spread use of the virtual circuit (VC) concept which relies on "reliable communication" [32]. The assumed VC reliability is not supported by existing communication protocols. Therefore, the use of VC leaves vast number of unhandled application states (timeout) that are directly responsible for the elusive crashes and data losses.

Justin Y. Shi propsed a Statistic Multiplexed Computing (SMC) [33, 34, 35, 36, 37, 38] framework to tackle with the unhandled failure problems. SMC, in summary, is general-

izing the packet switching concept for general purpose distributed and parallel computing applications. SMC has three principles:

(a) *a common unit of application data transmission or* $< key, value >$ *tuple,*

(b) *a store-and-forward application tuple switching architecture,*

(c) *end-to-end retransmission discipline.*

Although (a) is currently common in large scale distributed and parallel systems, such as Cassandra [39], Hadoop and Spark [40], (b) and (c) are still challenging open issues. Theoretical and experimental studies proved that SMC applications promises delivering incrementally better performance and reliability at the same time [41, 42, 43, 44]. Extreme scale HPC application built with careful engineering on top of SMC principle have scalability limits given unlimited resource.

2.2 Fallacies

The Open System Interconnection (OSI) reference model has been a standard for network communication systems. Figure 2.1 illustrates OSI reference model overview. OSI model partitions a communication system into seven layers: Application, Presentation, Session, Transport, Network, Data Link, and Physical. Each layer in OSI model has different responsibilities. Each layer feeds the upper or lower layer for inbound and outgoing communications.

The two-sided point-to-point communication TCP/IP protocol has became widely accepted as the standard building blocks for all distributed applications. The OSI Standards states "Transport Layer: Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing." After OSI model became the

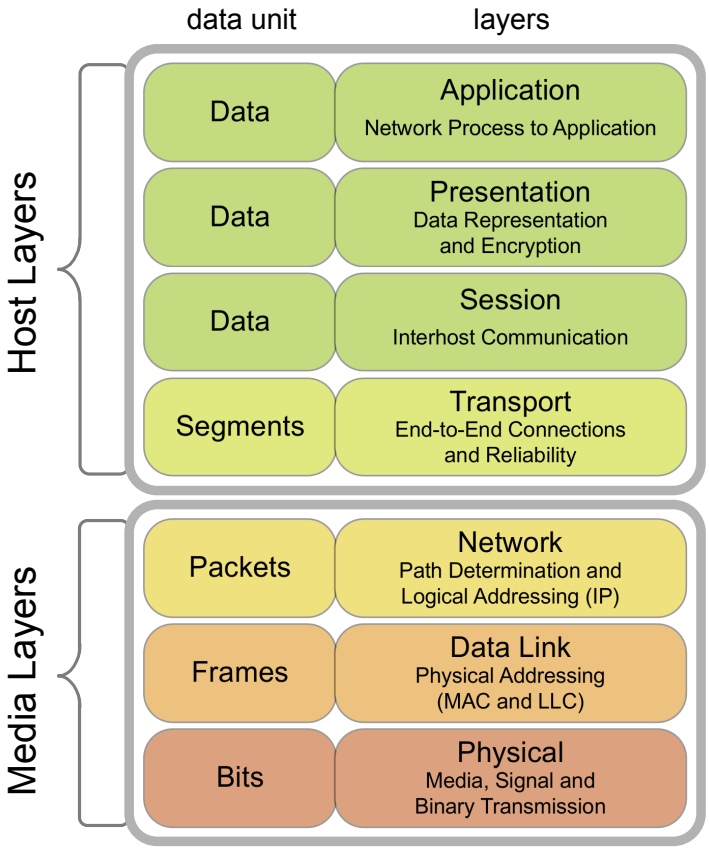

Figure 2.1: The OSI Model [wikipedia].

standard for communication systems, Peter Deutsch and his colleagues proposed the fallacies of distributed computing [25] in which the top fallacy was "The Network is Reliable." The problem arises when distributed applications depend on point-to-point communication paradigm while any end point can arbitrarly crash at any time. The fundamental flaw of point-to-point communication protocols is their inability to eliminate arbitrary data losses. Arbitrary data loses makes network unreliable, and they are not exception. Although the probability of data loss is small, it is the biggest treat to distributed application reliability as the application and data grows in size. Reliable communication systems are key to distributed application reliability. It is impossible to develop reliable distributed applications before fixing unreliable communication network. This vulnerability cannot be rectified unless the applications are re-engineered [24].

2.3 Assumptions

- The first assumptions made in this study is that the network is unreliable. Failures in distributed and parallel computing applications are mostly consequences of arbitrary data losses. These arbitrary losses mostly arise from network failures which is based on point-to-point communication paradigm.

- The second assumption is that storage is not reliable. A storage component can fail anytime during a computation. Every human made component has a lifetime which implies that a component failure is a norm rather than exception, and it cannot be predicted when a failure will happen. Therefore, storage failures are assumed.

- The third assumption is computing components are not reliable. A CPU or any associated part running on a task can fail at any time.

- The last assumption is the minimal survival resource set (MSRS). For a given path from $A \rightarrow B$ in a computing application, there must be MSRS that the computation can be carried out. MSRS can be any part involving in the computation of $A \rightarrow B$, such as network components, CPU, and storage devices.

2.4 Statistic Multiplexed Computing Architecture

Fekete et. al. states in [27] that it is impossible to implement perfect data communication between two end points if the probability of an end-point failure is greater than zero. In addition, reliable data communication is possible even with faulty networks using statistic multiplexing (store-and-forward) network of data packets [16] as long as both sender and receiver are reliable at the time of need. Although, the probability of either sender or receiver failure is very small, the problem arises when the application and data grow in size. As the number of components involving in a single computation increases, the probability of

failure of any parts in the system increases proportionally. For some large HPC applications, hundreds of thousands of heterogeneous computing, network, and storage devices are used for a single computation. Increasing the number and heterogeneity of participating components decreases the Mean Time Between Failure (MTBF). It is observed that the MTBF is steadily decreasing in recent years from weeks to sub hours [26].

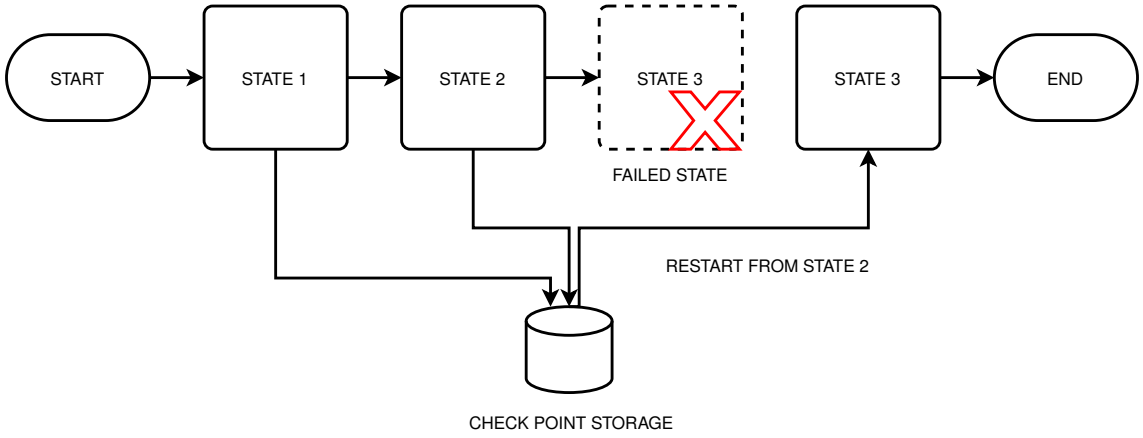

Figure 2.2: Check-point restart.

Redundancy has been the main solution for fault tolerance to overcome with system failures [44]. Spatial or temporal are two types of redundancy approaches currently being used as fault tolerance mechanism for HPC applications. Temporal redundancy (Figure 2.2) deals with re-running an application from a previously check-pointed state. Spatial redundancy (Figure 2.3) duplicates computation tasks to overcome failures. The former method periodically stores data and program states in a permanent storage, in case, if a program fails, the system restarts the program from where the last successfully stated is stored. This mechanism called check-point restart (CPR). The latter one replicates tasks multiple times in case of one task fails, the whole program does not fail by relying on the failed tasks' duplicates.

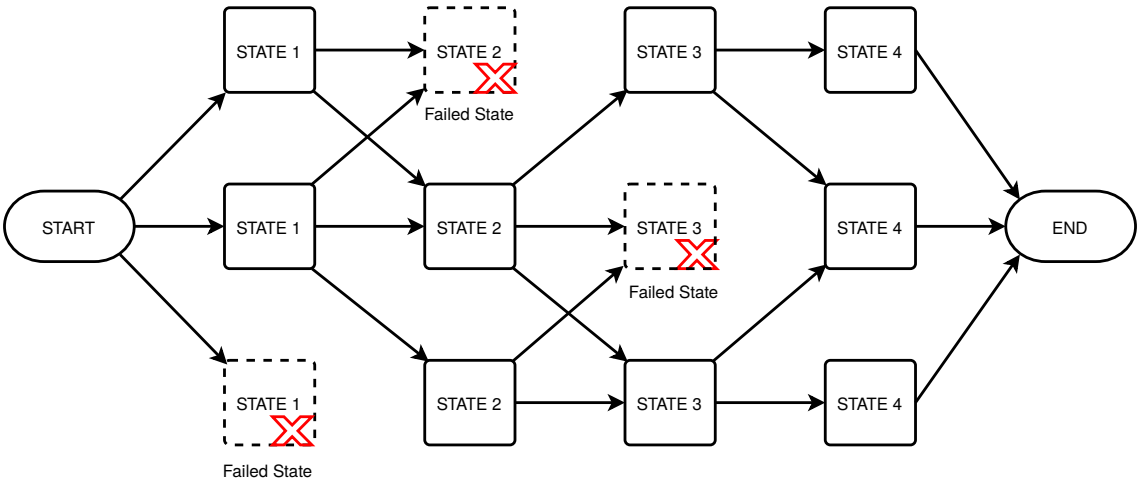

Figure 2.3: Spatial task redundancy.

The proposed SMC architecture was inspired by the Internet architecture for it has demonstrated its ability to deliver 100% reliable data packet services in faulty networks and to scale with incrementally better performance and reliability.

2.5 Tuple Switching Network

Although several ring architectures have been designed and deployed in many applications [45, 46, 47], the Tuple Switching Network designed by Shi is best suited for implementing SMC principle. Many look up services, such as Chord, use Distributed Hash Tables (DHT) when searching a desired data piece. The time complexity for a look up operation is $O(logn)$ in worst case when using DHT with ring. However, system infrastructure expansion and shrinking require a lot of data migration. For instance, when a new node joins or leaves Chord ring, a resdistribution of key partition and data migration may be required among nodes.

The Tuple Switching Network (TSN) (Figure 2.4) was designed to implement the principles of SMC for practical distributed and parallel processing. There are two design goals:

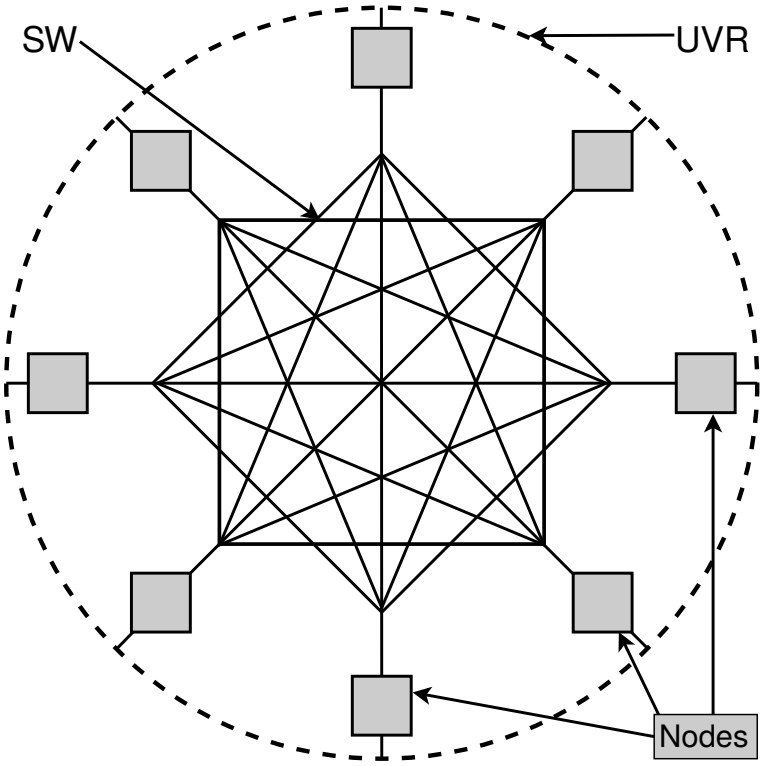

Figure 2.4: Tuple Switching Network using Unidirectional Virtual Ring.

a) Ease application programming by delivering single system image [48].

b) Deliver unlimited scalability by not assuming reliable (computing, communication and storage) components [1].

The TSN framework combines the tuple space API (application programming interface) ([49]), data parallel semantics with a statistic multiplexed tuple space engine. In particular, the tuple space API delivers the ease of programming using the popular $< key, value >$ pair convention with implied data parallelism ([33]) for automatic task scheduling and load balancing. The statistic multiplexing engine eliminates all single-point failures for SMC applications. Unlike the traditional explicit-parallel commands, such as $< send >/< receive >$ and $< read >/< write >$, the $< key, value >$ pair tuple space API supports three implicit parallel operators: *read*, *put* and *get* [49] and [33]. This simple API allows the SMC runtime to inject the statistic multiplexing effects for both computing and communication compo-

nents. Specifically, a unidirectional virtual ring (UVR) was designed to statistic multiplexing user-defined tuples in order to leverage potentially massive number of network switches and processor nodes. The use of UVR breaks the traditional dependency on virtual circuits allowing the elimination of all single-point failures. This also means that SMC applications are capable of tolerating massive permanent and transient software/hardware failures while delivering unlimited scalability.

2.5.1 Store and Forward Principle

The store and forward principle is borrowed from packet switching network in which packets are stored in an intermediate station for integration check and forwarded to a next station until it reaches its destination. A packet left from a sender may visit several intermediate stations before arriving its destination. The nodes in SMC principle act like intermediate switches in between senders and receivers. Every node between a source and destination ensures reliable packet delivery until the packet arrives its destination given that there is minimal survivable resource set. Packet delivery protocol does not rely on single path from source to destination.

2.5.2 Timeout-retransmission Discipline

Timeout-retransmission mechanism is one of the most critical part in order to have complete reliable packet delivery network. Timeout-retransmission discipline eliminates failures. While a tuple is traveling from a sender to receiver, any part can fail during delivery. Application level timeout-retransmission mechanism ensures that an ACK is received for every tuple transfer. If an ACK is not received for a tuple transfer, the sender retries the same tuple assuming that either the receiver or a part on the path has failed. In either case, the tuple will be delivered using different path or to a different destination. A receiver eliminates

duplicate tuples guarantees that a tuple is processed only one time.

2.5.3 Data Unit Definition

Data unit definition is critical for SMC completeness. A well defined unit of data per application ensures that SMC framework and application components, such as master and workers, agree on fundamental data exchange protocol. The proposed SMC paradigm uses a high level application programming interface (API) in the familiar $< key, value >$ tuple semantic. A unique key is generated per tuple using UUID method and associated with a value which caries the actual data. A value can be as simple as a string or a more complex object encapsulating several data pieces in different format. For instance, a value for matrix multiplication can be an array of numbers which represents a row or rows of a matrix, or it can be an object that contains a person information such names, phone numbers, addresses, and so on. The value is defined by application specifics. A value is represented as a byte array by SMC framework.

2.5.4 Gateway

Gateway is coordination mechanism in the SCM architecture. It is responsible for synchronous data replication that may encounter race conditions. As indicated in [50], the gateway has three crucial functions in the SMC architecture:

a) Dynamic serialization for eliminating race conditions. Requests received at gateway are processed in the first-in-first-served (FIFS) fashion. If two requests want to change same data object, the latter request is postponed until the first one completes. This mechanism prevents data inconsistencies.

b) Timeout-retransmission discipline without compromising the data change ACID properties. Gateway tracks all the requests it is processing. If gateway does not receive ACK for any requests after some certain retries and time threshold, the corresponding receiver is marked as *transient failure* for a certain time. This time window is configurable according to user specific policies. Finally, the *transient failed* node is marked as *permanent failure* and removed from UVR by issuing a *DECOMMISSION* tuple. Every gateway has a *duplicate checker* mechanism to avoid processing same tuple multiple times.

c) Non-stop Resynchronization for reconciliation of inconsistent datasets in case of node failures. If a node $JOIN$ the ring after a failure, it is treated as a new fresh node. It's data is not reliable anymore. The reconciliation mechanism resynchronize the newly joined node with the close neighbors.

The gateway ensures that ACID properties are enforced for all data changes while performing synchronous parallel data replication whenever there are no update conflicts. The non-stop data resynchronization algorithm is a new form of 2-phase commit protocol that does not force service shutdown when replicas return inconsistent states. Instead, an arbitrary replica is selected to service clients while forcing others to resynchronize. This new form 2-phase commit protocol does not impose performance penalty. The cost is a brief (< 60 seconds) worst case downtime per resynchronization incident, independent of dataset sizes. The non-stop resynchronization algorithm is not yet implemented in current prototypes.

2.6 SMC and Scalability

Extreme HPC application can promise unlimited scalability if the underlying framework avoids program to hardware binding architecture. To show the feasibility of SMC application

scalability, we compare explicit and implicit parallel application paradigms. If a task is explicitly assigned to a node as seen in Figure 2.5, the system expect the computation result of the task from the same node. If the node assigned to a task cannot return a result in case of transient or permanent failures, the whole job may fail unless the master program logic handles such cases.

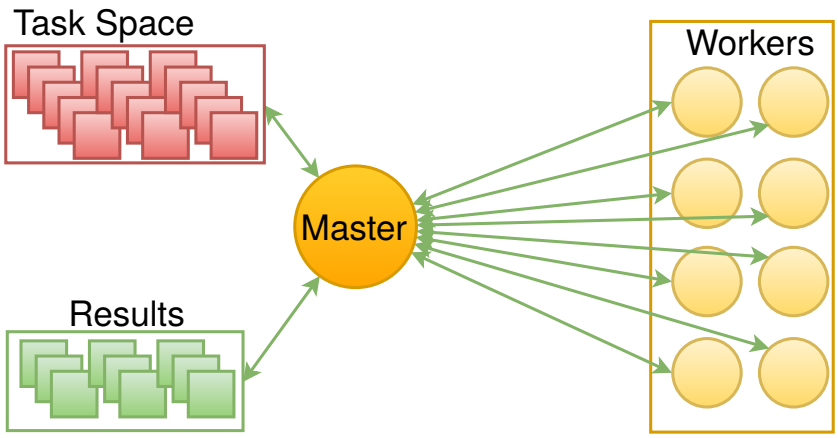

Figure 2.5: Explicit Parallelism Architecture

The probability of node failure increases as the number of explicit workers in a parallel application grows. Explicit parallelism is a major barrier to HPC applications.

SMC architecture, on the other hand, is designed to utilize implicit parallelism to avoid this barrier. Figure 2.6 illustrates implicit parallelism architecture. A task is not explicitly assigned to a node. Instead, a distributed tuple space serves as task cloud for workers to consume. A task can be processed by any worker. The tuple space is distributed and replicated on top of UVR. All the tasks in the tuple space ultimately will be consumed by some workers unless all workers die. Worker failures does not crash a whole job as long as there are some live workers to process unprocessed tasks. Tuple space can be considered as a communication network among HPC application components such as master and workers. Tuple space serves for both unprocessed tasks and produced results.

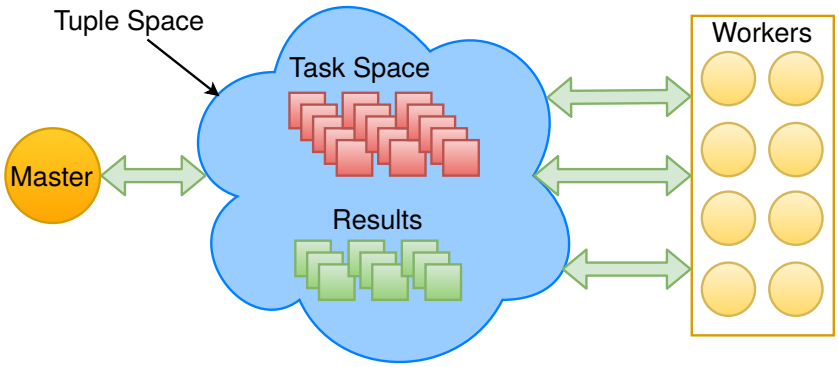

Figure 2.6: Implicit Parallelism Architecture

HPC applications built according to explicit parallelism paradigms tend suffer from "program-harware binding" as the underlying infrastructure, such as network, storage and computing devices, expands, which prevents a system to scale up. For "program-harware binding" systems, the probability of failure increases since the probability of whole job failure is the sum of probabilities of individual component failures. SMC has ability to deliver incrementally better performance and reliability due to "program-hardware soft-binding." A whole job does not fail if some of individual components fail. A job will ultimately complete as long as there exists a minimal survival resource set. Expanding computing infrastructure does not increase the probability of job failure. As the the infrastructure expands, the probability of live minimal survival resource set increases, thus better performance and reliability is delivered. However, the rule of "diminishing of returns" still applies for fixed problem size. Adding more computing nodes will not have effect on the performance once the optimum degree of parallelism is reached, where the performance will be plateau.

Chapter 3

SINGLE-SIDED COMMUNICATION

3.1 Background

One of the main challenges in exploiting extreme scale distributed computing is having reliable interprocess data exchange mechanisms. Hop-by-hop communication paradigms, in other words two-sided, has been the dominating data exchange model in HPC community. MPI (Message Passing Interface), RPC (Remote Procedure Call) and RMI (Remote Method Invocation) high level distributed programming models have mainly used two-sided communication paradigms. MPI-2 introduced one-sided communication model in 1996 [7], in which the interprocess communication is performed through a "shared memory." However, this model can also be considered as two-sided since the process which calls $MPI\_Put()$ and $MPI\_Get()$ operations required the other end to be alive. Any node crashes can still cause system failure since the calls are dependent on two sides.

As stated in section 2.6, "program-hardware" binding is one of the main obstacle to unlimited

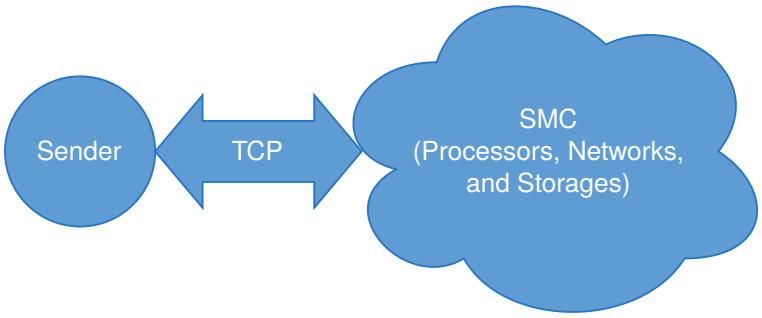

Figure 3.1: Single-Sided SMC Paradigm

scalability. All the above mentioned communication paradigms use two-sided communication model, which requires both ends to be alive during the course of communication. A practical solution is to unbind software from hardware, which forces application developer to avoid the assumption of reliable network and the receiver side. Although two-sided communication model has served the HPC community while building systems for small scale problems, it is now considered in [25] that the reliable network assumption is a fallacy. The issue becomes more visible as the size of the problem and computing infrastructure grows.

In this study, we use one-sided communication paradigm different than that of shared memory model. The communication between two sides is conducted through an intermediate layer, a distributed tuple space, which does not require the receiver side to be alive at the time of send. Rather than sending a message to an explicit receiver, the messages are put using an implicit $put(message)$ call in an reliable distributed tuple space to be consumed by a receiver. A receiver will then use an implicit $get(message)$ call to receive the message from the same tuple space.

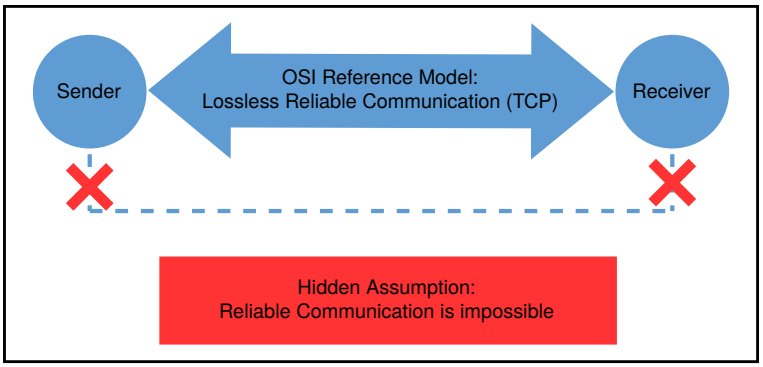

Figure 3.2: OSI Standards and Its Hidden Assumption

The root cause of reliable network fallacy comes from the hidden assumptions in OSI reference model (Figure 3.2). Although TCP is lossless communication protocol, its theoretical reliability is guaranteed at the packet level, given unbounded resources. The fallacy is assuming both ends, sender and receiver, will be alive during the communication process. The proof made by Fehete et. al. in [27] shows that it is impossible to implement reliable communication when the probability of either sender and receiver failure is greater than zero. Thus, node failures are expected not exceptions. On the other hand, it is proved in early 1980's [18] that reliable communication is possible using faulty network. In this study, we combine these two proofs to deliver reliable communication, computation, and storage using faulty hardware.

The SMC principle requires every component involved in a distributed and parallel computing architecture statistically multiplexed in order to promise "best effort" application level communication, computing, and storage.

3.2 Assumptions, Definitions, and Tests

The exponential growth of "big data" size [51, 52, 53] obligates companies, government institutions and academia to explore new ways of processing infrastructures and software systems. As explained in section 2.6, the scalability term in this study is defined as a system ability to deliver better performance and reliability as the size of the problem and computing infrastructure grow. Justin Shi et al. sketches three tests in [54] to determine whether a system is architecturally scalable or not. The same report also proposes Scalable Computing Service (SCS), which is defined as a computing service that allows unlimited infrastructure expansion without sacrificing performance, reliability and service quality degradation. The theoretical unlimited infrastructure term can be translated as "minimal survivable resource set" in practical applications.

A distributed computing system should pass the following tests in order to deliver scalable service:

(a) **Share-nothing Test:** Since there exists no manufacturing process that can guarantee 100% reliable components, no processors or networks can be assumed reliable. Not passing this test breaks the scalable service quality definition and End-to-End principle.

(b) **Sub-linear Cost Test:** The infrastructure management overhead must be no more than sub-linear timing factor with regard to the infrastructure size. A typical good system design should result in logarithmic complexity for broadcast and lookup.

(c) **Reproducible Node Test:** Each node in the architecture must be able to reproduce identical **semantically acceptable results** given identical inputs. These include both deterministic and non-determinstic programs. Failing this test will break the service quality definition.

According to definition of SCS, a system must pass tests (a) and (b). In addition to tests (a)

and (b), a system must also pass test (c) in order to guarantee reproducible SCS. Although all current computing system running reasonable

All modern computers running legitimate software can pass the *Reproducible Node Test*, it is challenging to pass the *Share-nothing* and *Constant Cost Tests*.

The architecture of the following systems is determined to pass the Share-nothing Test: a) **dataflow machines** [55], b) **Tuple Space machines** [56], and c) services by **Content Addressable Networks** [57] because of have two common features: single-sided paradigm, relying on content addressable network.

Statistic Multiplexed Computing (SMC) [33, 34, 35, 36, 37, 38] combines all three to deliver seamless scalable distributed system architecture. Specifically, "best effort computing" service can become possible by building a "Service Content Addressable Network" using the Tuple Space semantics and dataflow principle.

3.3 Single-Sided Statistic Multiplexing

As we explained in section 2.5, UVR (Fig. 2.4) is a SMC principle compatible structure in which a group of standalone nodes with any number of cores, storage devices, and network interfaces forms a ring by leveraging a collection of network switches. UVR is an implementation of Service Content Addressable Network, (aka Tuple Switching Network (TSN) [20]) without posing single point failures regardless the number of networks and processors in the infrastructure.

The single-sided communication paradigm employed by TSN avoids explicit hop-by-hop data exchanges. Applications using TSN as intermediate communication platform can efficiently exchange messages using implicit *put*() and *get*() operations. Therefore, an application does

not depend on specific hardware. All the components of an application, including master and worker applications, are parts of UVR architecture. All the parts are responsible for safely data delivery by implementing application level timeout-retransmissioin discipline.

However, there is a performance trade-off when using Single-sided communication paradigm since the data is indirectly exchanged among nodes. SMC principle leverages Chord's [45] ring-hoping algorithm for node traversals. The worst case time complexity for a lookup operation in Chord algorithm costs $O(logn)$, where n is the number of nodes in the UVR.

Consistent Hashing [58] $O(logn)$ time complexity may be reduced to $O(1)$ in some scenarios using client side caching [43]. It is up to developer how the consistent hashing is utilized.

3.4 Optimized Parallel Performance

The degree of parallelism determines the task size for a parallel application. It is difficult to find a task size that fits all problems for several reasons:

- Processing instances used in a computing infrastructure are generally consist of commodity hardware, therefore, instances are heterogeneous. Identical homogeneous hardware may not even deliver same computing power.
- It is difficult to exploit the aggregate bandwidth of a shared network infrastructure.
- The total number of instructions to compute tasks may not be same.

The ideal scenario is to find an optimal task granularity for load balancing when distributing a job. A perfect task granularity for a problem can deliver *time equilibrium* [59, 60] , where the communication and computation are overlapped most. *Time equilibrium* is to achieve optimal task granularity, where the compute instances start and finish processing tasks at

about the same time. The system overall performance is always bounded to the slowest machine or a machine which receives a task last, and finish it last. Distributed and parallel computing systems using coarse-grained parallelism, in other words larger granularity, tend to be imbalanced since some nodes may be idle while some others may have already received some tasks, or some nodes may finish their tasks way before others and sitting idle for long times. Therefore, fine-grained granularity may achieve higher concurrency.

It is important to note that there is always a trade-off between the communication overhead of finer granularity and imbalanced start and finish time of coarse-grained granularity. However, a theoretical optimal granularity size exists for all compute intensive problems, that successes the best balanced parallel performance for any homogeneous and heterogeneous environments.

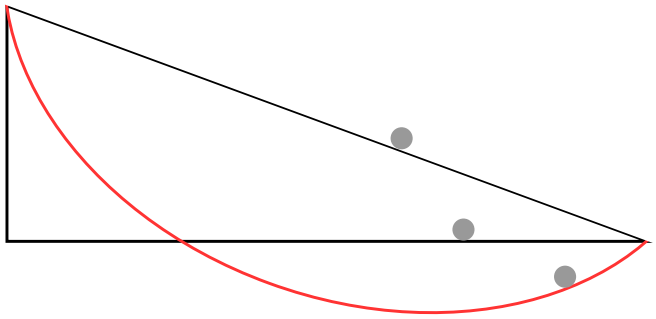

Figure 3.3: The solution to the brachistochrone problem is not a straight line or some combination thereof but a cycloid.

Assuming the best task size a homogeneous environment is to partition a problem into equal sized tasks may present imbalance computation. At first glance, one will think that the shortest path is the fastest path in Brachistochrone problem in physics and mathematics [61] illustrated in figure 3.3. However, it is proved that the fastest path is the red line in figure 3.3, where the optimal acceleration for a ball is succeeded with gravitational force. Finding ideal granularity for a problem is similar to Brachistochrone problem in this sense. Additionally,

this study shows that there exists multiple optimal granularity sizes for a problem citeshi92, ankacom. Brachistochrone problem has similar feature, called "Tautochrone", where a ball will reach the end point at the same time no matter where it starts.

Tuning the granularity for application using single-sided paradigm does not require reprogramming. Hadoop [62] and Spark [40] using single-sided computing paradigm have proved that they can achieve better performance and reliability than that of similar parallel programming paradigms for data intensive applications. Hop-by-hop programming paradigms requires reprogramming when changing task granularity unless some type of pool is not utilized [63].

The remaining of this chapter compares computation performance of two SMC principle prototypes against MPI. Synergy is an earlier SMC prototype (implemented in C) using a centralized Tuple Space to simulate the TSN [33]. AnkaCom is a Java implementation of the distributed TSN [42].

The computation platforms include the NSF Chameleon bare-metal cluster [64] at TACC (Texas Advanced Computing Center) and the owlsnest.hpc.temple.edu traditional bare metal cluster [65] at Temple University. The benchmark application is a naive dense square matrix multiplication application. The SMC prototypes are Synergy 3.0+ and AnkaCom 1.0 and OpenMPI 1.4.4 and 1.10.0.

3.5 Experiment Design

The main purpose of these experiments is to determine whether SMC prototypes with single-sided paradigm comparable with MPI, which uses hop-by-hop paradigm. We use dense matrix multiplication application because of its simplicity and high computing and communication requirements. The order of nested (i, j, k) loops is optimized for best cache hits as

shown in:

```
for (int i = 0; i < mA; i++)
   for (int k = 0; k < nB; k++)
      for (int j = 0; j < nA; j++)
         C[i][j] += (A[i][k] * B[k][j]);
return C;
```

where mA is matrix A's row count, nB is matrix B's column count, and nA is matrix A's column count.

We run these experiments on traditional bare metal HPC clusters "Owlsnest" [65] at Temple University and the newer NSF "Chameleon" bare metal cluster [64] at TACC (Texas Advanced Computing Center):

a) *Owlsnest cluster has native multiple IB (Infiniband) support. The Chameleon cluster only has single IB support.*

b) *The Chameleon and Owlenest bare metal clusters have same number of cores (48) and big memory (256 GB for Chameleon) per node and (543GB for Owlsnest).*

The Synergy's simulated UVR implementation can only sustain to a small number of cores before the single-threaded daemon fails to function properly, which is also reliability test for the concept of UVR. On the other hand, the new UVR implementation, AnkaCom, is a real distributed tuple space that utilizes SMC principle and single-single sided communication and computation paradigms can sustain any number of nodes and cores. AnkaCom is implemented in Java that introduces more overhead than C-MPI implementation.

In these experiments, we examine various configurations in the following ways:

- *Single-Node Single-Core. This group of tests examines the runtime's ability to leverage multiple hardware components in parallel by overlapping single-core computing, communication and disk activities.*

- *Single-Node Multiple-Core. This group of tests examines the parallel runtime system's ability to use up to 48 cores concurrently in addition to other communication and storage components.*

- *Multiple-Node Multiple-Core. This group of tests examines the parallel runtime system's ability to harness multiple multicore nodes effectively. In this study, we used twenty 12 core nodes on Owlsnest with dual IB support.*

3.6 Computational Results

3.6.1 Single-Node Single-Core Results

The first experiment is conducted on a single core to find a baseline performance of a sequential matrix multiplication C-program. Fig. 3.4 shows the results in GLOPS on Owlenest running matrix of sizes 1000 - 6000. In all reported tests, the C(gcc) and mpicc compilation option is –O3. The Java compiler does not have an optimization option.

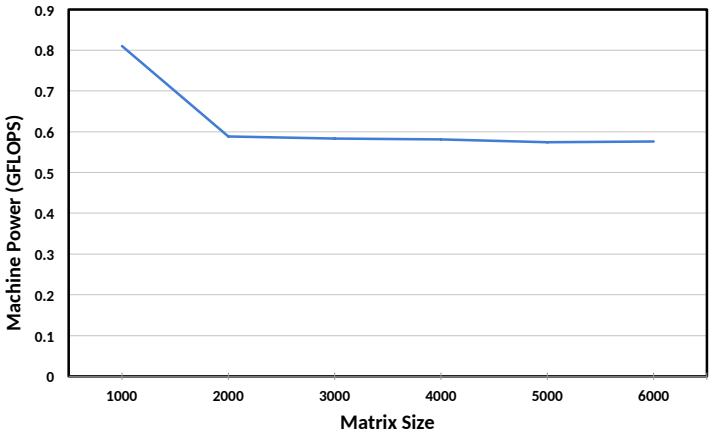

Figure 3.4: Single-Node Single-Core (Sequential)

The significance of Fig. 3.4 curve is that it depicts the typical "Cache, Memory, Swap and Die" CMSD single core performance envelope. This envelope enables quantitative application scalability analysis [60].

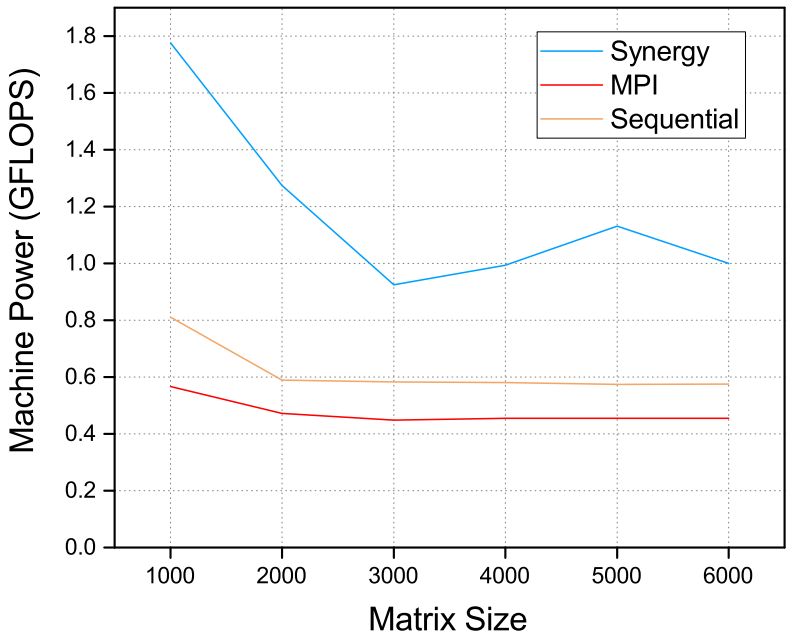

Figure 3.5: Single-Node Single-Core (Parallel)

Fig. 3.5 reports the single-core single worker tests for both MPI and Synergy against the sequential program (in yellow). MPI (Open MPI 1.4.4) was not able to compete against the sequential code in all multiple matrix sizes (1000-6000) consistently. The Chameleon results are very similar.

3.6.2 Single-Node Multiple-Core Results

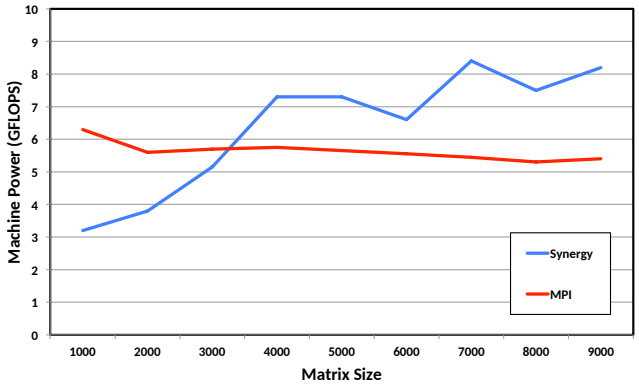

Figure 3.6: Single-Node Multiple-Core (Broader Range Tests)

Fig. 3.6 reports a broader range of tests for different matrix sizes without granularity tuning ($G = 20$ fixed). Synergy still outperforms MPI when the matrix size exceeds 3,200. These results are consistent with Fig. 3.5: the bigger the problem sizes, the better the Synergy performance.

3.6.3 Multiple-Node Multiple-Core Results

Figure 3.7 reports performances using AnkaCom 1.0 against MPI on owlsnest. This firgure also shows the scalability in terms of performance for both AnkaCom and MPI. The base test starts from 2 nodes, each with 6 workers, and the number of nodes increased by one for every test. The granularity is tuned for AnkaCom. Each AnkaCom run is optimized by

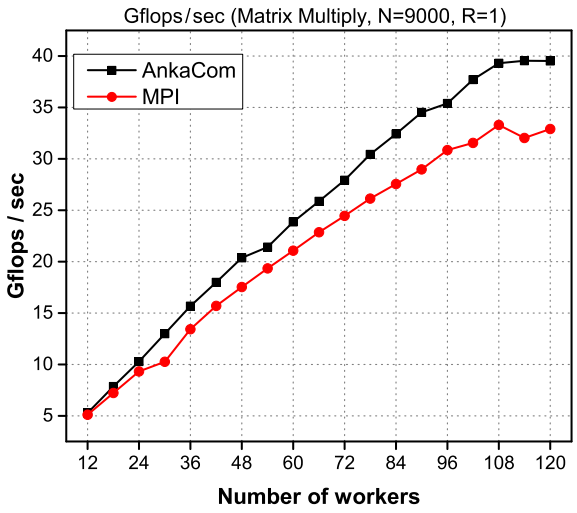

Figure 3.7: Multi-Node Multi-Core Tests, AnkaCom against MPI

picking the best performing granularity through multiple runs. The recorded MPI run is the best of three runs with identical task size N/P.

Fig. 3.7 shows consistently better AnkaCom performances over MPI. The performance differences grew bigger as the number of cores increased. The MPI performance started to trend down at P=108 while AnkaCom kept strong for yet another 12 more cores until the end where N/P became merely 75 (9000/120). It would be interesting to notice the Intel XEON clock cycle is 2.2GHZ. The 75 rows (9000 columns) of dot-products was the break-even point for sustained performance scalability. After this point, both MPI and AnkaCom's communication overheads took over. This was a demonstration of the economic law of "diminishing return".

3.7 Summary

Hop-by-hop computing and communication paradigms have served for small scale distributed applications. However, they are barrier to extreme scale data processing systems. As the size

of "big data" exponentially grows, the need for more reliable and scalable data processing systems increase accordingly. Single-sided communication and computation paradigm along with SMC principle has potential to deliver the scalability and reliability in order to process "big data" in time manner.

The programming paradigm for HPC applications should shift from IP-address-centric to data-centric techniques. Content addressable network is the form of data-centric distributed data store [43, 42] in which a computation is not binded to specific hardware, rather it is relying on data availability. As long as the desired data to be processed is available somewhere in the data store, a worker will process the data ultimately.

Chapter 4

UNIDIRECTIONAL VIRTUAL RING

In this chapter, we discuss Unidirectional Virtual Ring (UVR) concepts in depth by giving the main ideas behind the underlying components. We also describe implementation details of each component including $NodeManager$, $Tuple$, $TupleSpace$, ring forming and maintenance. In addition, we also examine failure scenarios and consequently actions taken to maintain correct and complete the single UVR image structure.

4.1 Introduction

UVR was designed to implement the principles of SMC for practical distributed and parallel processing. There are two design goals: a) ease of application programming and b) allows for unlimited scalability by not assuming reliable (computing and communication) components ([1]). Specifically, UVR was designed to statistic multiplexing user-defined tuples for leveraging multiple network switches and processor nodes. The use of UVR breaks the traditional

dependency on virtual circuits allowing the elimination of all single-point failures for each SMC application. Consequently, SMC applications are capable of tolerating massive transient software and hardware failures. UVR allows distributed and parallel applications run reliably on top of unreliable components. Thus SMC applications do not have theoretical scalability limits.

4.2 UVR Main Concepts and Implementation

The UVR has four main components: $NodeManager$ daemon orchestrates the node's actions based on tuple events; $Tuple$ a unit of transmission, which consists of a header and a payload; $TupleSpace$ a distributed raliable communication channel which provides and manages a set of $TupleBuckets$; $Timeout - Retransmission$ principle with $Store - and - Forward$ mechanism, which delivers reliable message delivery; and $Failure - Detection - and - Handling$ mechanism, which is responsible for detecting component failures and handling accordingly.

4.2.1 Node Manager

$NodeManager$ is responsible for maintaining the global UVR structure. Every node in UVR runs one $NodeManager$ daemon. The UVR implementation borrows some technical concepts from the Chord system [45] and first implementation in [3].

4.2.2 Tuple Space

UVR maintains a distributed tuple space for reliable communication among nodes. Tuple is the unit of transmission in the tuple space, that allows message exchanges. A tuple can be

thought as a TCP packet. A tuple has a header and a payload.

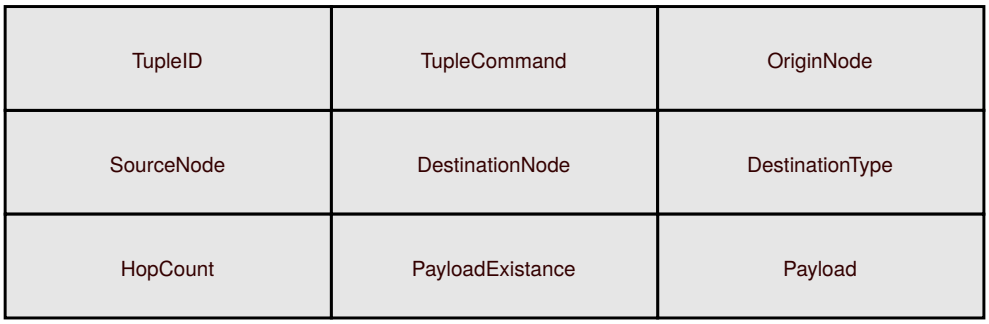

Figure 4.1: Tuple Structure [3].

Figure 4.1 depicts a logical representation of a tuple. A tuple's header caries information that is needed from time it leaves from a source until arrives a destination.

- $TupleID$ is a UUID generated at the creation of each tuple and identifies the uniqueness of the tuple in tuple space throughout its life time cycle.
- $TuleCommand$ represents the purpose of the tuple such as $JOIN$, $DECOMMISSION$ for ring maintenance or any application specific commands.
- $OriginNode$, $SourceNode$ and $DestinationNode$ fields identify the hosts information involved in a communication channel.
- $DestinationType$ caries information on which the Node Manager decides what routing algorithm to apply e.g.
  - *direct* is set if the destination is known upfront. In this case, the tuple space will deliver the tuple only to the specified destination using the shortest path.
  - *serial* is set if a tuple is to visit all the nodes in the UVR serially. A tuple is always delivered to the immediate successor until it reaches the tuple initiator.
  - $flood$ is set if a tuple needs to be delivered to all peers in the UVR in shortest time.

  - *lookup* is set if the destination is unknown. Tuple will not hit all the nodes by jumping ahead given some hints.

- *HopCount* field sets the number of max hop count a tuple can travel in the ring. The number in this field is decreased by 1 for every arrival to a node until it becomes 0, when the tuple is terminated.

- *PayloadExistance* is a boolean value to show whether this tuple is carrying a payload.

- *Payload* caries the actual data in piggyback model. *Payload* is of type binary in which almost any data could be carried, such as an application message, a text file, a picture, etc.

Each Node Manager in the UVR maintains a set for *TupleBucket* to store and forward tuples. Up on a tuple arrival, it is put in the appropriate *TupleBucket* depending on *TupleCommand*. Tuples are processed in the first come first served ordered inside each *TupleBucket* unless some other polices are specified.

4.3 UVR Maintenance Protocol

Single image ring maintenance is critical in order to deliver correctness and completeness for mission critical distributed applications. Since SMC makes no assumption for any component reliability at any time, the ring maintenance protocol is designed to deliver single image in case of node joins and leaves at any time i.e. in case of failures. Each node in the UVR is strongly connected with its neighbors via logical connections. Each node dynamically maintains a *FingerTable* which stores its position in the UVR and relationship with its neighbors. Figure 4.2 depicts a sample *FingerTable*. A node has an immediate successor and a list of subsequent successors; and an immediate predecessor and a list of predecessors.

This *FingerTable* is periodically exchanged among peers via heartbeats and in case of any node arrivals and departures. *FingerTable* is updated in the event of any changes.

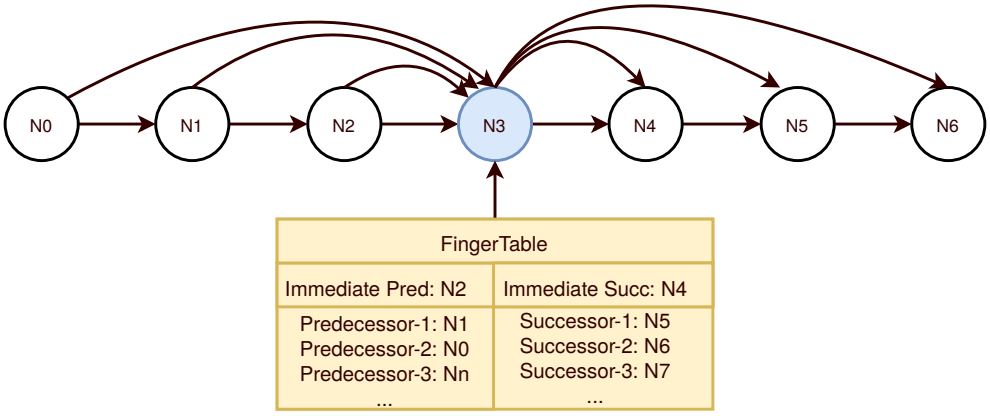

Figure 4.2: Finger Table

The following steps show a node join without failure.

1. **JOIN:** A node joining the ring needs to know an access point to initiate the joining process. Similar to peer-to-peer systems, we assume that a *seedList* is provided from which a node is selected as the contact point. The *seedList* can be provided either as a static file or a dynamically maintained service running on *http* server or any configuration service such as *Zookeeper* [66]. Once the joining node has an access point, it sends a $JOIN$ tuple. A node is placed in the ring according to node order policy. The node placement policy is application specific. The current implementation places nodes based on $RingID$ which is calculated from on node's $DataCenterID, RackID, IPAddress, port$.

2. **JOIN_OK:** The node receiving $JOIN$ tuple does not necessarily responds back to the initiator. If it is the prospective successor to the initiator, it sends back a $JOIN\_OK$ tuple to approve the joining node's request. Otherwise, the $JOIN$ tuple is injected in to the tuple space to find the correct place for the initiator to join. The tuple will serially travel the ring until it finds the prospective successor, which responds back to

the initiator with the $JOIN\_OK$ tuple. The prospective successor to the joining node marks the joining node as $seccondaryPredecessor$ in its $FingerTable$.

3. **JOIN_COMMIT:** The joining node sends $JOIN\_COMMIT$ tuple upon receiving $JOIN\_OK$ message from the UVR. The prospective successor to the initiator node becomes the immediate successor node to the joining and forwards $JOIN\_COMMIT$ tuple to the ring so that every node has relationship to the new node updates their $FingerTable$ accordingly.

Nodes can join or leave the ring at any time for any reason. The above steps explains the process of node joining. A $DECOMMISSION$ tuple is published into the tuple space to declare a node's leave so that peers in the ring updates their $FingerTable$ accordingly.

4.3.1 Failure Scenarios

The steps explained in the previous section assume non-failure environment during joining process. However, any failure can happen during this process. In this section, we discuss potential failure scenarios while a node is joining the ring.

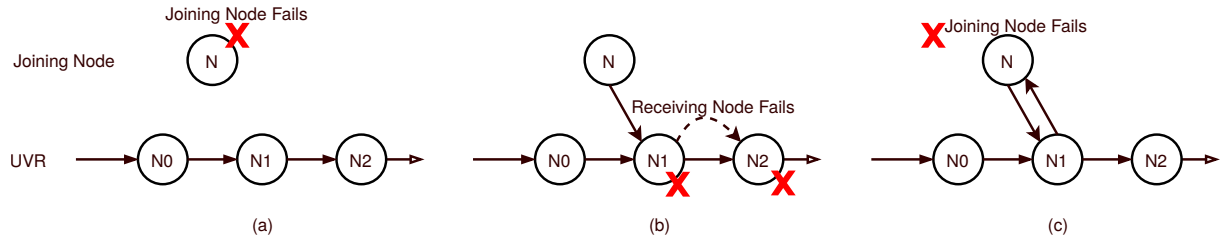

Figure 4.3: Failure scenarios: (a) the joining node fails before sending $JOIN$ tuple; (b) the receiving node(s) failure after $JOIN$ tuple is delivered; (c) the joining node fails after successfully sending $JOIN$ tuple.

Figure 4.3 shows the following failure scenarios:

(a) The joining node fails before sending $JOIN$ tuple, which means the tuple does not leave from the node. There is no effect to the ring in this scenario.

(b) Any node in the path to the prospective successor to the joining node fails after receiving the $JOIN$ tuple before sending $JOIN\_OK$. In this scenario, the joining node waits for $JOIN\_OK$ tuple for a given time limit and timeouts if a response is not received. The joining node then retries the ring from different access points in the *seedList*. This process continues until the $JOIN\_OK$ tuple is received from any node in the ring.

(c) The joining node is marked as the prospective successor by a node in the ring after receiving $JOIN$ and is sent the $JOIN\_OK$ tuple. After this point, the node in the ring expects *heartbeat* from the joining node since it is now the temporary successor. If no heartbeat is received from the prospective successor for a certain time, the *tempSucccessor* field in the $FingerTable$ is unmarked and a $DECOMMISSION$ tuple is sent to the ring. At this point no change happens in the UVR ring.

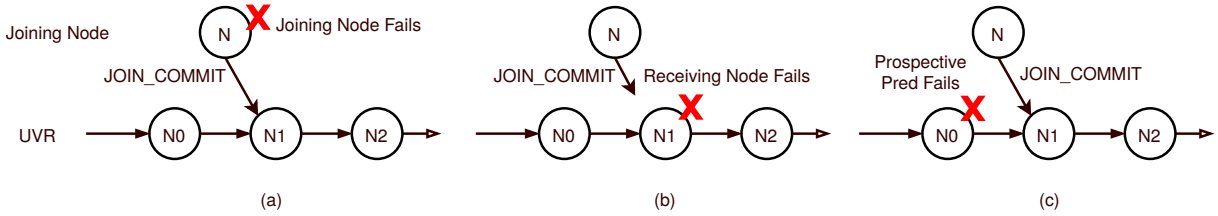

Figure 4.4: Failure scenarios: (a) the joining node failure after sending $JOIN\_COMMIT$ tuple; (b) the receiving node failure before receiving $JOIN\_COMMIT$ tuple; (c) the prospective predecessor node for the joining node fails before receiving $JOIN\_COMMIT$ tuple.

Figure 4.4 shows the following failure scenarios:

(a) The joining node fails after successfully sending $JOIN\_COMMIT$ tuple to its successor. At this point, according to figure 4.4 (a), $N1$ expects *hearbeats* from new node N.

If $N1$ does not receive any signals from N, the regular $DECOMMISSION$ process is initiated and $N1$ reconnects with $N0$, it is previous immediate predecessor.

(b) The prospective successor $N1$ to the joining node N fails before receiving $JOIN\_COMMIT$. Since the joining node had received $N1$'s $FingerTable$ at the time of receiving $JOIN\_OK$ tuple, it has information about $N1$'s immediate successor and successor list. At this point, the joining node will try to join the ring by connecting to $N2$, the immediate successor of $N1$.

(c) The prospective predecessor to the joining node fails before receiving $JOIN\_COMMIT$ tuple. At this point, the joining node N has become a part of the ring, namely the immediate predecessor of $N1$. The new node N will expect *heartbeats* from $N0$. If it does not receive any signal, the regular $DECOMMISSION$ process is initiated to remove $N0$.

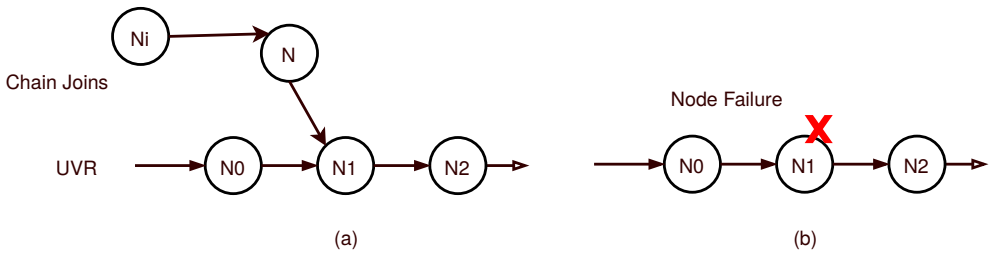

Figure 4.5: Failure scenarios: (a) The chain joining happens when a node wants to join the ring by contacting a node, which is in the process of joining. (b) a regular node failure.

Figure 4.4 shows the following failure scenarios:

(a) The chain joining happens when a node wants to join the ring by contacting a node, which is in the process of joining. This potential issue is solved by not accepting any inquires at the time of joining process. A joining node simply ignores $JOIN$ requests made by other nodes.

(b) A node could be declared "fail" by multiple mechanisms. The first mechanism consists of *hearbeat* sender and receiver daemons. If a node is not able to send a *heartbeat* tuple after trying several times along with timeouts, it assumes that the node it is trying to send *heartbeat* is failed. On the other side, every node expect *heartbeat*s from their immediate predecessors. If a *heartbeat* is not received in a certain time period, the predecessor is assumed to be failed. The second mechanism triggered if a node is not able to deliver a tuple to a peer, the receiving peer is assumed to be failed. On top of the UVR failure detection and handling mechanisms, application specific mechanism can be introduced to deliver application level reliability.

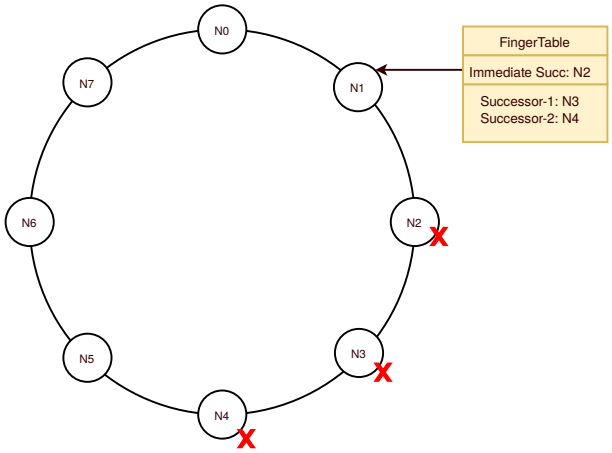

Figure 4.6: Failure scenarios: Cascading failures.

Cascading Failures: As stated before, each node in the UVR ring maintains a logical $FingerTable$, which stores a limited list of successor and predecessor nodes. The limit is configurable and is currently set to $m = logn$, where n is the total number of nodes in the ring. This number can vary per application. It should be set to $m = n/logn$ in order to deliver $logn$ lookup time. The cascading failure happens when a node's immediate successor and all other subsequent successors in its $FingerTable$ fail simultaneously. Figure 4.6 visualizes this scenario where the other edge of the ring, namely $N5$, is unknown to the node $N1$, which is trying to heal the ring. The probability of this scenario is very low if the order of the

nodes in the ring is randomly distributed where no more than $m-1$ nodes in the ring are depending on same single "unit" such as same rack, switch, power supply, datacenter etc. For example, probability of having a cascading failure starting from $N0$ to Nm is given by:

$$p_0 p_1 ... p_{m-1}(1-p_m)(1-p_{m+1})...(1-p_{N-1}) = \prod_{i=0}^{m-1} p_i \prod_{j=m}^{N-1}(1-p_j) \tag{4.1}$$

where p_i denotes the probability of failure for node i, m denotes the number of nodes in a $FingerTable$. The probability of a set of m consecutive nodes failure can be calculated with equation 4.1. However, since there are N number of set of m consecutive nodes in a ring, the probability of any set of m consecutive nodes failure can be calculated with equation 4.2.

$$P = \sum_{k=0}^{N-1} \prod_{i=k}^{m+k-1} p_i \prod_{j=m+k}^{N+k} (1-p_j) \tag{4.2}$$

Equation 4.2 calculates P, the total probability of ring partition without external healing service. Here, when index exceeds N, we consider $p_{N+k} = p_k$. As stated in section 4.3, an external service is used for ring forming and maintenance. Even in this low probability P, where all the successors fail, a node will try an access point from the *seedList* and finds the other edge of the broken ring and heals the ring.

Chapter 5

ANKASTORE: A SINGLE-SIDED STATISTIC MULTIPLEXED TRANSACTIONAL DISTRIBUTED STORAGE SYSTEM

5.1 Introduction

Data management has been one of the most challenging tasks for many communities including science, industry, and government. Although client/server model has been widely used for small scale of data management, it is now the main barrier when application and data grow in size. To tackle with this challenge, academia and industry have been proposing new designs apart from traditional client/server model. Complete decentralizing and decoupling are two main architectural requirements in order to achieve scalable and reliable distributed data stores. The purpose of distributed data stores is not only to deliver reliable

and scalable service solely for data-centric requests, but also provide the ability to run high performance applications on top of data. MapReduce [22], for instance, one of the best known computation framework that runs on top of Hadoop Distributed File System (HDFS) [11].

Data storage is the ultimate performance and reliability bottleneck of all mission critical infrastructures. The rapidly growing digitized data demands robust data storage and processing infrastructures that are reliable (lossless), non-stop servicing and without a scalability cap for all applications. Inspired by the statistic multiplexed computing (SMC) concept and architecture, we report report the preliminary results of a prototype implementation of a statistic multiplexed transactional storage engine called AnkaStore. AnkaStore is part of a proof of concept implementation of the statistic multiplexed computing architecture for general purpose distributed and parallel data processing applications. It can be seamlessly integrated with the SMC computing engine to deliver highly reliable extreme scale storage and data-intensive computing solutions. This paper reports replicated file read/write performance, reliability and service quality using the fully distributed AnkaStore prototype and HDFS (Hadoop File System).

As the circuit packaging technologies approaching the physical limits, the need for clustered computing became evident. The rapid growth in digitized data and the scalability dilemma of the existing clustered computing and storage architectures motivated the research on statistic multiplexed computing (SMC) architectures. In this research, assuming unlimited resources, we seek the scalability of a clustered service architecture as the ability to expand the cluster indefinitely without service reliability, quality and performance degradations. That is, the cluster service architecture should allow the application to grow indefinitely while delivering constant service turnaround time by expanding the processing infrastructure indefinitely. The Internet is such a service architecture for data communication applications.

These goals are motivated by the growing practical demands. They also seem theoreti-

cally feasible if one applies the principle of statistic multiplexing correctly to computing, or Statistic Multiplexed Computing (SMC).

For compute intensive clusters, studies have shown that it is possible to completely decouple program/data from processors and networks. Consistently better performance and reliability against the legacy message passing system were recorded ([44],[42]). For clustered storage, the idea is also to completely decouple data from its storage devices. Thus expanding the storage size will not suffer rapidly degrading service quality, reliability and performance.

However, unlike compute intensive applications, applying the SMC principle to extreme scale persistent data storage requires solving the CAP (Consistency, Availability and Partition tolerance) challenges. Otherwise, the clustered storage will continue to suffer the scalability dilemma.

The fundamental difference of the SMC cluster from other computing/storage cluster is its departure from the de facto point-to-point communication paradigms, such as RPC (Remote Procedure Call) and message passing and share-memory conventions. A single-sided application content addressable service network is proposed as the only means for program-program communication. This design was necessary in order to circumvent the well-known point-to-point communication impossibility. As a result, the SMC cluster can promise zero single point failures for the target service independent of the infrastructure scales. It is highly desirable that the scalable clustered storage architecture follow the same framework.

This section is organized as follows. The "Background and Related Work" Section introduces the possibility and impossibility theories, definitions of scalability and three scalability tests. It then explains the solution to the "split brain" CAP challenge and its application to transactional data storages via a Transaction Addressable Network (TRAN). Brief introduction of HDFS (Hadoop File System) is also included. Section II describes the SMC architecture assumptions and its unique UVR (Unidirectional Virtual Ring) structure. Sec-

tion III describes the UVR-based storage cluster architecture that is capable of supporting unlimited data intensive computing needs. Section IV describes the details of the AnkaStore prototype architecture. Section V reports the counter-intuitive computation results of replicated bulk read and write tests against HDFS. Section VI contains the summary and acknowledgements.

We present the design and a prototype implementation of Anka Store in this paper. Anka store is proof concept of statistic multiplexed computing architecture for distributed applications. Anka store promises scalable performance and high reliable framework for extreme scale storage and computing applications

5.2 Background and Related Work

5.2.1 Possibility and Impossibility Theories

The first revelation in the study of scalable computing is the impossibility of implementing reliable (point-to-point) communication in the face of (end point) crashes [27]. This proof revealed that direct use of the OSI (Open System Interconnection) reference mode l[67] in distributed computing can be harmful. In fact, the direct use of OSI reference model was cited in 1997 as the first "fallacy in distributed computing" [23]. The scalability dilemma was predicted.

The second revelation is that reliable computing can become possible if one applies the statistic multiplexing principle correctly. It was proved for the OSI reference model in early 1980's that reliable 100% lossless communication is deliverable using faulty components as long as the end-points are reliable [18]. The fundamental principle of the reliable communication service rests on three assumptions: complete decoupling of data packets from

transmission devices, 100% end-point reliability and the infrastructure affords the "minimal survivable resource set" that is capable of statistic multiplexed processing at the time of need. In literature, this is often cited as "the best effort communication". For data communication applications, such as streaming media and VOIP (Voice over IP) applications, these best effort services are sufficient for unlimited application scalability. For mission critical computing applications, such as database and data storage or high performance computing, however, assuming the 100% reliability of the end-points is the root cause of the scalability dilemma.

Careful reading of the reliable communication proof also revealed that leaving any extreme scale service in the hands of the sender (or client) is completely reasonable in practice and in theory.

In other words, single-sided statistic multiplexed computing can potentially guarantee 100% reliable service regardless the scale as long as program/data are completely decoupled from the physical hardware; and the infrastructure affords the minimal survivable resource set to complete any request with the best effort while keeping the sender connected.

The same principle is applicable to persistent data storage. As long as the data can be completely decoupled from its physical storage devices.

However, complete decoupling of persistent data from its storage requires solving the CAP puzzle [68]. Consistent data replication at scale for persistent storage has been an open challenge in distributed computing.

5.2.2 Solving the Split Brain Puzzle

The CAP conjecture concerns the feasibility of non-stop data service in extreme scale under faulty component conditions[30]. Data consistency, service availability and tolerance to

network partition are all desired service properties. It was proved in 2002 that under the assumptions of point-to-point communication paradigm and the atomic data replication conditions, C, A, and P may not be achieved at the same time [19]. In existing clustered storage systems, data consistency may be relaxed (eventual consistency) in order to deliver higher service availability.

The problem is that once data consistency is relaxed, the quality of data service deteriorates as the cluster expands (the A=B test). The scalability dilemma persists.

Using the single-sided SMC paradigm brings a new possibility: a transaction addressable network (TRAN). The idea is to build an application dependent service network that is application content addressable. The use of TRAN can circumvent the impossibility theory completely while fully leverage the statistic multiplexing principle. In other words, as long as synchronous data replication can satisfy the A=B test and and the constant cost test, with share-nothing guarantee, the new storage system should be able to free to scale in performance and in reliability without losing service quality. To see this, one can set R be the replication factor and P the number of independent data partitions. Given A=B and constant cost guarantees, increasing R will enhance service reliability and quality, increasing P will distribute the data accessing loads. It follows that as long as $P >> R$, one can expand the persistent storage indefinitely without negative performance, reliability and service quality consequences.

For solving the "split brain" puzzle in CAP, one only needs to build a TRAN switch or gateway as illustrated in Figure 5.1.

The TRAN gateway functions as an application content switch, synchronous data replicator and global conflict resolution center by statistic multiplexing multiple physical storage nodes with identical contents. The TRAN is incomplete without the application-side retransmission discipline. This discipline is identical to the TCP/IP packet retransmission discipline but

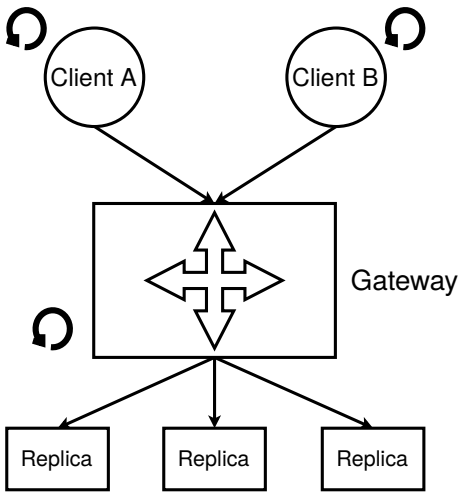

Figure 5.1: Transaction Addressable Network.

at the level of application tuples (or transactions). Since the gateway is stateless, it can be easily multiplexed without the loss of any transactions within the TRAN.

In Figure 5.1, the gateway interfaces the clients and the multitude of storage nodes. The horizontal arrows represent potential integration with other gateways (see next section). This design must also be checked for the "Constant Cost" test. Otherwise, the same dilemma remains.

For TRAN, the "Constant Cost Test" has two parts: a) Explicit A=B cost test, and b) Implicit A=B cost test.

The explicit A=B cost includes replication and conflict resolution overheads. The theoretical minimal is the sum of concurrent update conflict resolution overheads as argued in Jim Gray's "The Danger of Replication" paper [69]. Assuming $P >> R$ and infinite raw storage and network supplies, this cost can be confined to a constant independent of data growth.

The implicit A=B cost covers the overheads for maintaining A=B in production. Since A and B are equally likely to develop data corruption and software/hardware service needs, non-stop data resynchronization is necessary. Without it the scalability dilemma remains.

The gateway synchronous replication protocol shifts the traditional two-phase commit protocol (2PC) into two loosely coupled dependent phases: a) dynamic serialization (conflict resolution) and b) Non-stop service data resynchronization. The dynamic serialisation algorithm includes the discipline to instantly cut-off servers whose execution status do not agree with others (data inconsistency found). Since data consistency only needs to be maintained for all connected users, the strategic gateway position in TRAN plays critical roles in both tests: a) it maintains the constant cost for conflict resolution and can render the maximal replication performance subject only to the rule of diminishing return (total data transmission volume against total available network bandwidth); b) it is responsible for the non-stop data resynchronization using the "Mobious Strip" principle (more in next section). Since the non-stop data resynchronization algorithm can guarantee continuous data service with the worst-case 60 second service downtime, independent of data size [36], the "Constant Cost Test" can be satisfied.

Although CAP Theorem remains true, using TRAN ensures a persistent storage service to expand indefinitely without performance, quality and reliability degradations. As a consequence, the maximal deliverable performance is only governed by the rule of diminishing return based on the balance of pure computing/communication necessities. The clients can still choose to use out-of-sync data at their own risk. The out-of-sync nodes will eventually become consistent. In this context, the CAP concerns are moot.

5.2.3 Hadoop File System

Hadoop is a distributed data processing system designed for very large distributed data intensive applications. HDFS (Hadoop File System) is the bottom layer of clustered data storage service (Figure 5.2). Fault tolerance was a critical goal in Hadoop design. All data files are automatically replicated according to a user-defined replication factor R. To

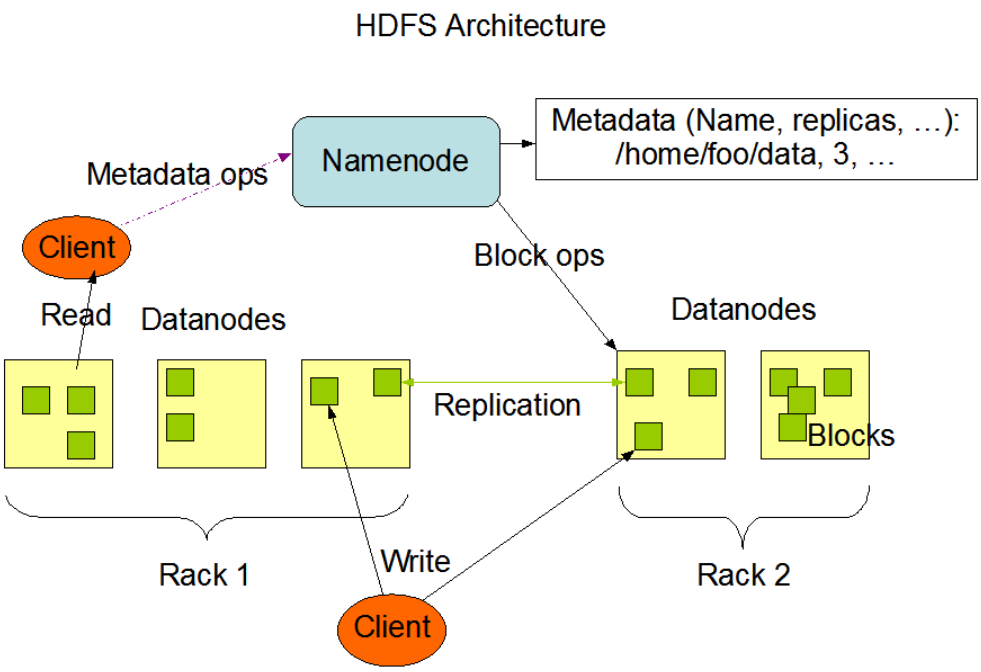

Figure 5.2: Hadoop File System.

accelerate the massive data movements, pipeline replication is used in order to maximize the efficiency of all components in parallel. Spark is an in-memory computing engine that runs on top of HDFS (Figure 5.3). Spark has demonstrated consistently faster performances compared to the Hadoop native map-reduce engine [40].

Although Hadoop's API (Application Program Interface) is single-sided $< key, value >$, the Hadoop implementation followed the traditional point-to-point communication paradigm. The consequence is the single Namenode architecture[11]. The Namenode and Hadoop cluster manager (Figure 5.3) fail the "Share-nothing Test". Consequently, Hadoop's single namespace/namenode architecture imposes a 100M file design limitation [31]. However, HDFS was performance optimized and widely deployed in industry and academia for large scale data processing.

HDFS is an ideal target of this comparison study. This paper reports raw data read and write tests using a prototype SMC storage system named AnkaStore. Reliability and service

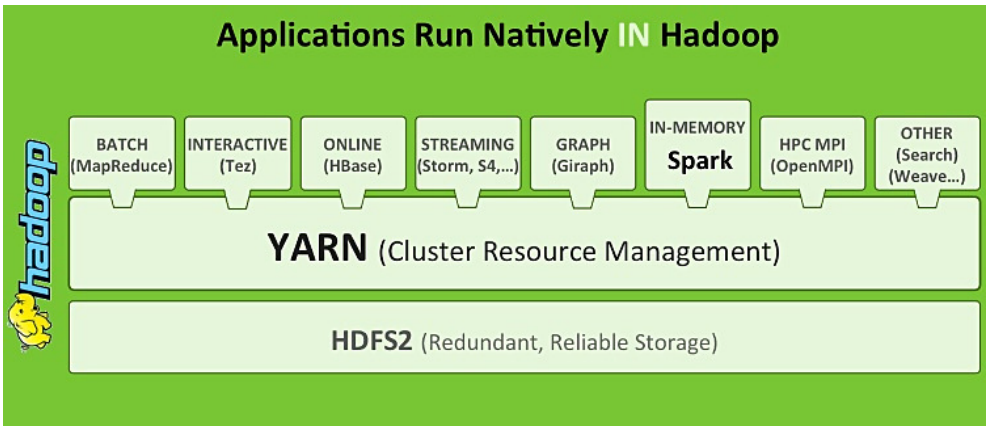

Figure 5.3: Hadoop and Spark.

quality comparisons are also included.

5.3 Statistical Multiplexed Computing Architectures

The SMC framework departs from the common cluster computing paradigms in its use of single-sided application-dependent content addressable network. The API (Application Program Interface) is also $< key, value >$ but with the dataflow (Tuple Space [41]) semantics in order to meet the content addressability needs.

The SMC single-sided content addressable networks (SMC-CAN) have the following assumptions:

- The application client is reliable and connected.
- All processors, networks and storages can fail arbitrarily (excluding the client machine).
- For each application, 100% reliability must be delivered if the infrastructure affords the minimal survivable resource set.

The SMC content addressable network is an overlay network that is only complete when the

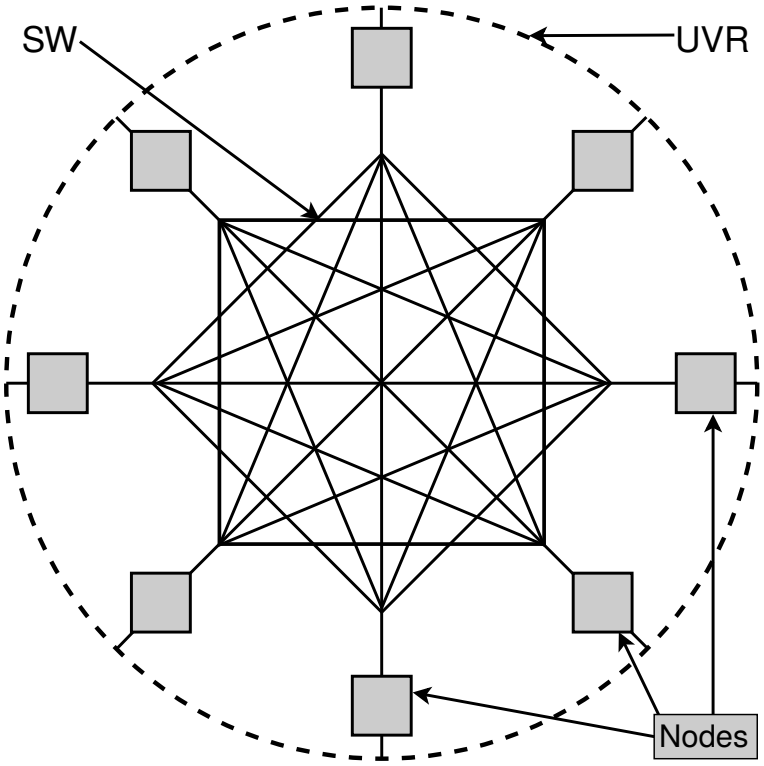

Figure 5.4: Unidirectional Virtual Ring.

clients are complete with respect to the application-level SMC protocol. Once the application is SMC-enabled, in many cases, its minimal survivable resource set can be reduced to a single processor with a local storage with the minimal network support.

This paradigm shift has direct impacts on all mission critical applications. Since useful non-mission critical applications will eventually become mission critical, the proposed SMC framework has very broad impacts.

The physical construction of the SMC architecture is shown in Figure 5.4.

The UVR (Unidirectional Virtual Ring) [37] implements the application content addressable network using the $< key, value >$ API and dataflow machine semantics. Each node is a stand-alone computer with any number of cores, local memory, network interfaces and storages. Each node runs the SMC protocol in addition to the TCP/IP protocol stack. All nodes have access to either a centralized storage of parallel programs or locally replicated

application programs for computation. Like existing content addressable networks, such as CAN [57], Chord[45], and Tapestry [46], UVR is also an overlay network. Unlike the existing efforts that focused on object search and routing algorithms, each UVR is only responsible for application dependent services. It can dissolve when service is no longer needed. UVR focuses more on efficient data processing employing all available resources. Specifically, the UVR is only used to constrain the scope of resource search – implementing the "single system image". Since the SMC framework permits 100% application reliability under minimal survivable resource set, thus the minimal UVR requirement is to ensure the client can reach the minimal survivable resource set. Actual data exchanges are conducted via multiple direct physical networks, not UVR.

The SMC content addressable network is incomplete without the client-side SMC protocol library. The SMC framework literally implements a new dynamic application-dependent service layer on top of the existing TCP/IP protocols. This single-sided application dependent network design completely circumvents the impossibility of point-to-point communication paradigms[27].

Within the SMC content addressable network, the following are the minimal implementation requirements:

- $< key, value >$ pairs with extended headers are the units for statistic multiplexing and application management,
- The application programs and the node managers form the dynamic content addressable network
- The application programs must include $< key, valu >$ timeout-retransmission discipline.

5.3.1 Single-Sided SMC Storage Cluster

To implement a single-sided SMC storage cluster, the above mentioned UVR needs to include data storage support at each node. In other words, each node needs to have its own storage content addressable network. These local storage content addressable networks form the UVR-level storage content addressable network via the high level SMC protocols.

Figure 5.5 illustrates the UVR-based storage content addressable network architecture (AnkaStore).

Given the descriptions of the replication gateway in Section II, this storage content addressable network can pass the "A=B Test" and the "Constant Cost Test" easily.

5.4 AnkaStore

AnkaStore is a distributed file system that can be seamlessly integrated with the distributed and parallel computing engine AnkaCom [42].

Like typical fault tolerant content addressable networks, the AnkaStore overlay network supports dynamic node leave, join and hot spot resolution operations. Unlike the compute intensive SMC networks, AnkaStore network will persist. In this regard, AnkaStore is similar in purpose to the OceanStore project[70].

The primary goal of the AnkaStore implementation is to demonstrate the feasibility of the SMC-based single-sided storage cluster for delivering better performance, reliability, and service quality as the infrastructure expands. AnkaStore has four components.

- NodeManager is responsible for forming and maintaining the UVR. It runs the SMC protocol on top of TCP/IP protocols.

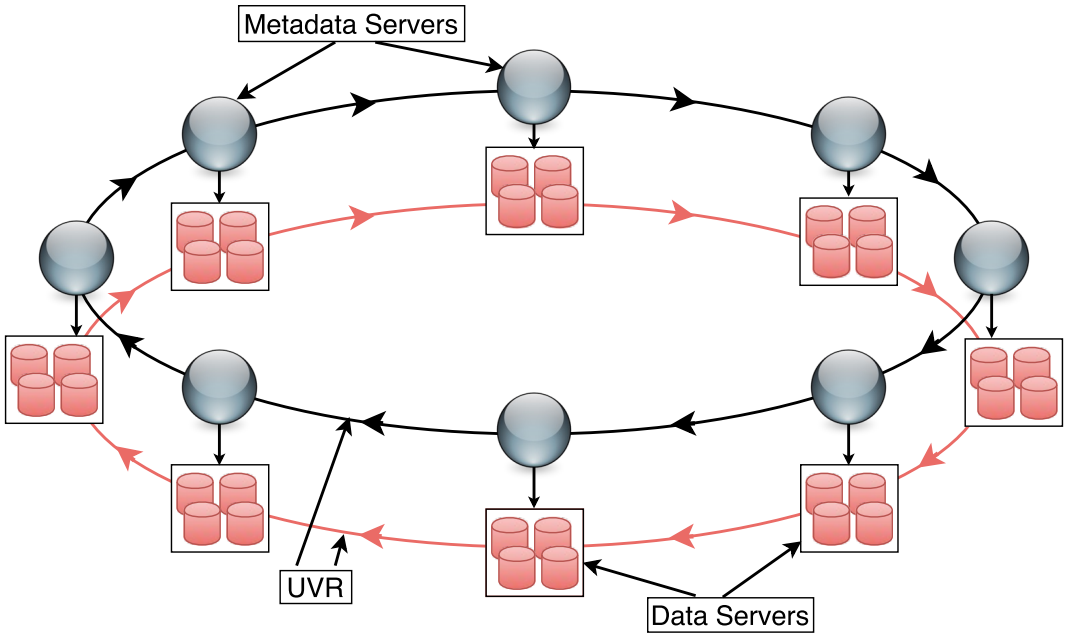

Figure 5.5: Storage Content Addressable Network.

- Gateway is responsible for the local and global storage content addressable networks. It implements user metadata management, request coordination, data replication, conflict resolution/request serialization;
- Storage engine is responsible for file block management and replication;
- Failure detection and handling. These utilities are for on-site detection and warning. The application-side SMC retransmission discipline is ultimately responsible for the completion of the user requests.

5.4.1 Unidirectional Storage Ring

Figure 5.5 shows a logical representation of AnkaStore virtual ring. Nodes in the UVR has peer-to-peer relation with each other. Every node in the ring maintains m number of predecessor and successor nodes, where m is log of n nodes.

A node may join or leave the ring any time. A $JOIN$ tuple is issued to inform other nodes

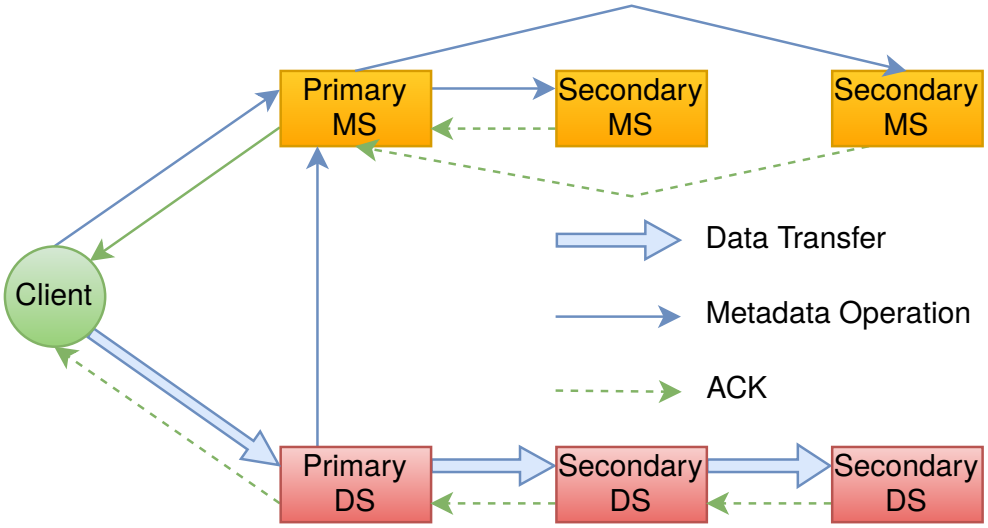

Figure 5.6: System architecture: A Client with metadata operation (MS) and data store (DS) requests.

in the UVR for new arrival nodes. A DECOMMISSION tuple will be sent to replace a failed node with the next alive node in order to maintain a valid UVR structure. Transient and permanent software/hardware failures can cause arbitrary number of nodes dropped off from UVR. New or recovered nodes can also join UVR at arbitrary time. The current AnkaStore implementation forms the UVR based ordered RingID using the Jenkinson hashing algorithm [71]:

$$RingID = H(DataCenterID, RackID, IPAddress, port)$$

Although non-essential, each NodeManager also contains self-diagnosis and recovery functions. It sends heartbeats to its predecessor node. If a node does not respond in some specified time, it's marked as failed [3].

5.4.2 Node Manager

All nodes in the UVR runs a NodeManager, which is responsible for forming and maintaining global UVR structure. Anka store implements two communication channels for tuple space managements. NodeManager has its own communication channel to interact with other NodeManagers. Having a dedicated communication channel among NodeManagers reduces tuple management complexity. A node communicate with other nodes by issuing and injecting a $< key,\ value >$ tuple into the global tuple space. A tuple finds its destination though global tuple space. The tuple switching algorithm delivers a tuple to its destination in $O(logn)$ in worst case. SMC principle rely on multiple switches in parallel in order to implement tuple switching network.

5.4.3 Gateway

Each Anka node also runs a gateway daemon for metadata operations. All clients interact with Anka through Gateways. As discussed, each Gateway performs:

- dynamic serialization for eliminating race conditions
- timeout-retransmission discipline
- non-stop resynchronization for reconciliation of inconsistent datasets

Figure 5.7 shows an overview of Gateway. Gateway is responsible for synchronous data replication. Requests received at a primary MS are processed in parallel. A conflict resolution mechanism ensures that metadata inconsistencies do not happen in case if two write operations are received for the same metadata object by serializing requests based on their arrival time. A latter received request is released after a former request is completed. A write operation is synchronously replicated over replicas using gateway mechanism to maintain

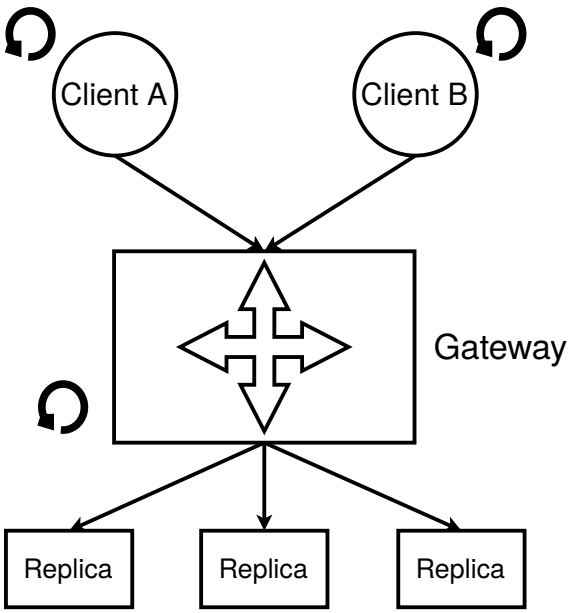

Figure 5.7: Gateway for TRAN.

namespace consistency. In this context, unlike HDFS, AnkaStore is a transactional storage cluster that allows concurrent file updates. The "primary" gateway for a user is the local gateway that is also responsible for local read operations. Having synchronous data replication and serving read operations on behalf of a user ensures that metadata inconsistency does not happen. A client should never face stale (inconsistent) data.

Due to the understanding of the impossibility theory [27], the timeout-retransmission discipline is used throughout the SMC framework in order to deliver reliable services while MTBF (Mean Time Between Failures) of cluster components is rapidly declining to sub hours for large distributed systems [26]. Unlike the 2-phase commit replication protocol, the gateway does not rollback any transactions. It follows the identical retransmission discipline borrowed from TCP[72]:

$$RTT_{new} = (1 - \alpha) * RTT_{current} + \alpha * RTT_{old}$$

A replica is not immediately considered as failed in case of timeout. If a node timeouts for three retries, it is marked as a transient failure, and all the update requests received from

that point are accumulated in a cache. The accumulated updates are sent to the node if the becomes alive again. However, the transient failure node is marked as permanent failure if there is no response after a certain time. A $DECOMISSION$ tuple is issued to inform other nodes in the UVR to remove the failed node from the ring. The next node to the failed node is temporally assigned as the primary node for the users to whom the failed node was assigned as primary node. The time window for transient and permanent failure is configurable to meet application specific requirements.

The data on the failed node is not usable for mission critical applications. It may still be used for non-mission critical purposes. A non-stop resynchronization algorithm can recover the data to mission critical status on the failed node once it recovers. The resynchronization algorithm (by the Gateway) first takes a snapshot of its neighbor's data and transaction log who have either primary or secondary data and use it to build the new node. It then continually scans the source node for new update logs. There are two cases. a) Empty update log is found. This means that the source and target data are identical. The new node can join the cluster for service. b) The scan continued for a fixed number of rounds and still no empty log can be found. The Gateway will pause data service. This pause allows the replication threads to complete data copying and recover the new node for service. The Gateway pause is triggered by receiving a larger log compared to an earlier log. The replication queue is automatically throttled to avoid overflowing. The worst-case pause duration can be confined to a small constant, such as 60 seconds. The non-stop resynchronization process is transparent to the users.

5.4.4 Namespace Management

Every node in the ring is responsible for storing and managing metadata and actual data. A node in the ring is the primary MS (Metadata Server) for a group of users and secondary

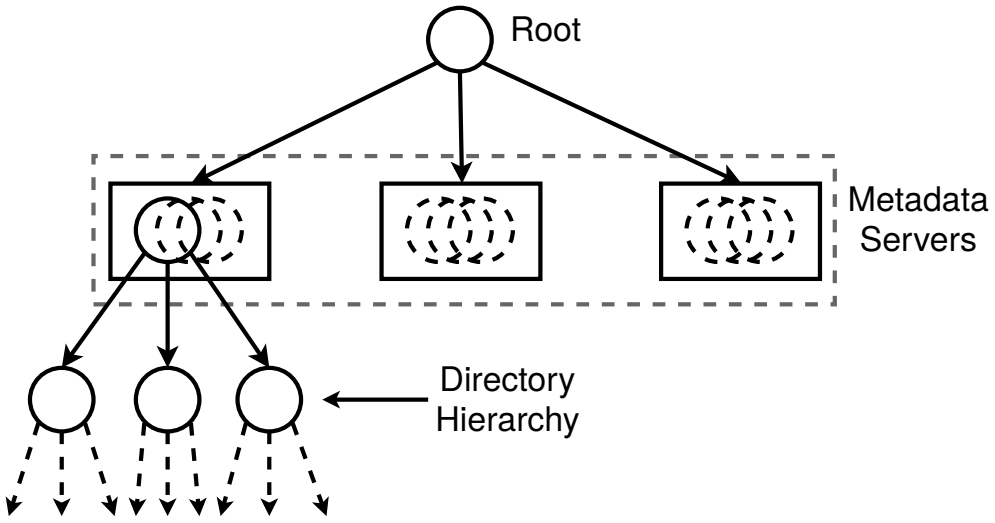

Figure 5.8: Metadata distribution.

MS for some other group of users controlled by the replication factor. For replication factor 3, for instance, a node has a set of primary users and two sets of secondary user sets for two of its predecessor nodes. Only a primary MS handles both read and write operations for its primary users. Secondary MSs forward client requests to user's primary MS. Figure 5.8 shows the global namespace tree. File inodes stored in the namespace tree maintain information similar to UNIX file system with the addition of block maps which encapsulates block ids and locations along with replicas.

Following is a logical representation of a file in JSON format. It shows all the necessary metadata information belong to a file including block replicas location.

```
{
   "o":"UserA",
   "n":"B",
   "h":"UserA/B",
   "c":1531334464852,
   "l":0,
   "m":1531334464852,
   "s":"",
```

```
"u":{
},
"f":{
   "10G.txt":{
      "i":"bc9f3adc-8ca9-4713-b157-d2fa4606aa88",
      "n":"10G.txt",
      "o":"UserA",
      "p":"UserA/B",
      "h":"UserA/B/10G.txt",
      "f":10737418240,
      "c":1531362159300,
      "a":1531362159300,
      "m":1531362159300,
      "s":null,
      "b":42,
      "l":{
         "bc9f3adc-8ca9-4713-b157-d2fa4606aa88_1":[
            "10.52.0.101:50044",
            "10.52.0.130:50044",
            "10.52.0.85:50044"
         ],
         "bc9f3adc-8ca9-4713-b157-d2fa4606aa88_2":[
            "10.52.0.130:50044",
            "10.52.0.85:50044",
            "10.52.0.29:50044"
         ],
         "bc9f3adc-8ca9-4713-b157-d2fa4606aa88_3":[
            "10.52.0.36:50044",
            "10.52.0.129:50044",
```

```
            "10.52.0.126:50044"
         ],
         "bc9f3adc-8ca9-4713-b157-d2fa4606aa88_4":[
            "10.52.0.29:50044",
            "10.52.0.200:50044",
            "10.52.0.36:50044"
         ],
         "bc9f3adc-8ca9-4713-b157-d2fa4606aa88_5":[
            "10.52.0.200:50044",
            "10.52.0.36:50044",
            "10.52.0.129:50044"
         ]
      }
   }
}
```


Metadata Primary Node Selection Protocol

A primary node selection protocol is followed when a new user is added to the system. When a super user wants to add a new user, a request is created with ADD_USER command with necessary user information, and sent to an arbitrarily selected node in the ring, say node A. The node in the ring received the request injects a tuple into metadata tuple space (MTS). This tuple visits every node in the ring serially to collect some metrics about Anka servers including $\{available\ space, available\ memory, op/min, CPU\ usage, network\ load\}$. After node A receives the tuple back with collected data, it selects "the best available node" out of best three nodes from the received node list based on their ranking which is calculated

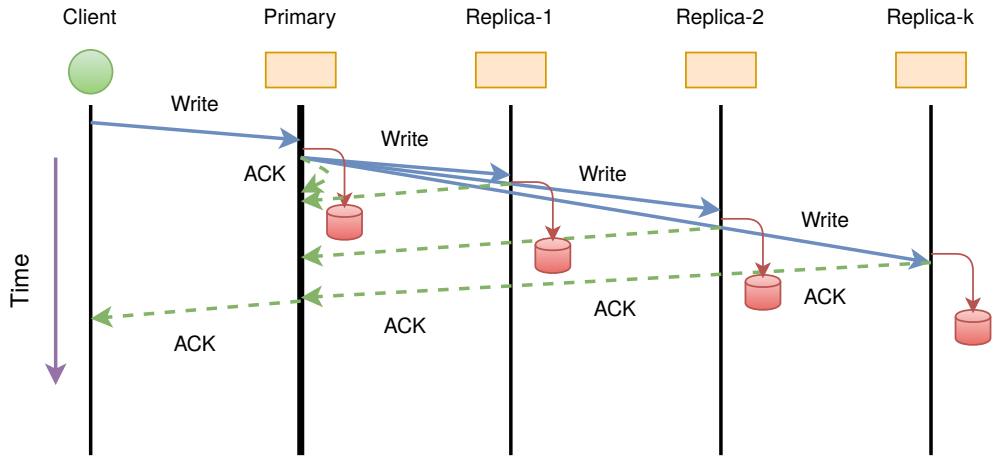

Figure 5.9: Parallel replication.

using following function:

$$f_i(S_i, M_i, O_i, P_i, N_i),$$

where for $node_i$, S stands for *available space*; M for *memory usage*; O for *operation/minute*, P for *CPU usage*, N for *network load*. $\{S,\ M\}$ has positive effect and $\{O,\ P,\ N\}$ has negative effect on ranking. This protocol guarantees at most one, at least one, and only one primary node is assigned to a user. In case, if two users are added with same user id at the same time, the protocol fails both requests.

Replication and Consistency Model

Metadata is replicated over multiple MSs for reliability and fault tolerance. Gateway is responsible for synchronously replicating metadata. Although file replication seems a well researched subject, it is not clear at all if concurrent data replication could out-perform pipelined parallel replication. Figure 5.9 shows the logic of concurrent parallel replication for metadata replication. When a gateway receives a request for updating or adding new metadata object, it first checks whether it is the primary MSs for the owner of request.

Otherwise, it forwards the request to the ring. The primary MS creates a replica map per request, in which there is a mapping for primary MS and per secondary MSs to track the status of replication. A thread is spawned in Gateway to replicate and track the status of the request. As the thread receives ACKs from replicas, it updates the status map. Finally, the thread marks the request status $SUCCESS$ upon receiving ACKs from all replicas, and sends an ACK message to the request owner. Requests aiming to update same object are serialized based on their arrival time, and processed in first in first served manner.

Metadata Management

Metadata updates are synchronously replicated and stored on multiple MS'. The Metadata Servers store inode information in JSON format. Each level of hierarchical directory structure is stored in a separate JSON file. Storing each level in separate file reduces the management complexity and helps fast access if the metadata is later reloaded. A separate version is stored upon new update, and the oldest one is removed. The number of version depends of application needs and can be configured.

The MS keeps a copy of namespace for each connected user in-memory cache for fast access. A level of metadata hierarchy for a user is streamed to the permanent storage as a new version upon an update request. Anka only loads first three levels of the metadata tree. Deeper levels are only loaded upon request.

5.4.5 Data Management

For user data storage, AnkaStore is a block addressable network. The unit of multiplexing is a file block. The block size is configurable per file, and the default size is 128 megabytes. Each file block is replicated on multiple nodes running local file systems.

Block Distribution and Replication

File blocks are distributed and replicated on multiple nodes. Effective block distribution is crucial to effectively utilize the network bandwidth across the system and load balancing for storage devices. We use "best available node" algorithm to select candidates for block placement. This algorithm collects two metrics about nodes when a client wants to add a new file. Network Load and Storage Device Availability are two inputs to the block placement algorithm for calculating ranking for each node and sorts them based on their rankings. A list of "best available nodes" is provided to a client for block placement. The current block placement strategy ensures that the system avoids data load imbalances (hot spots). Network load imbalance is a much harder research challenge since users behavior changes dynamically.

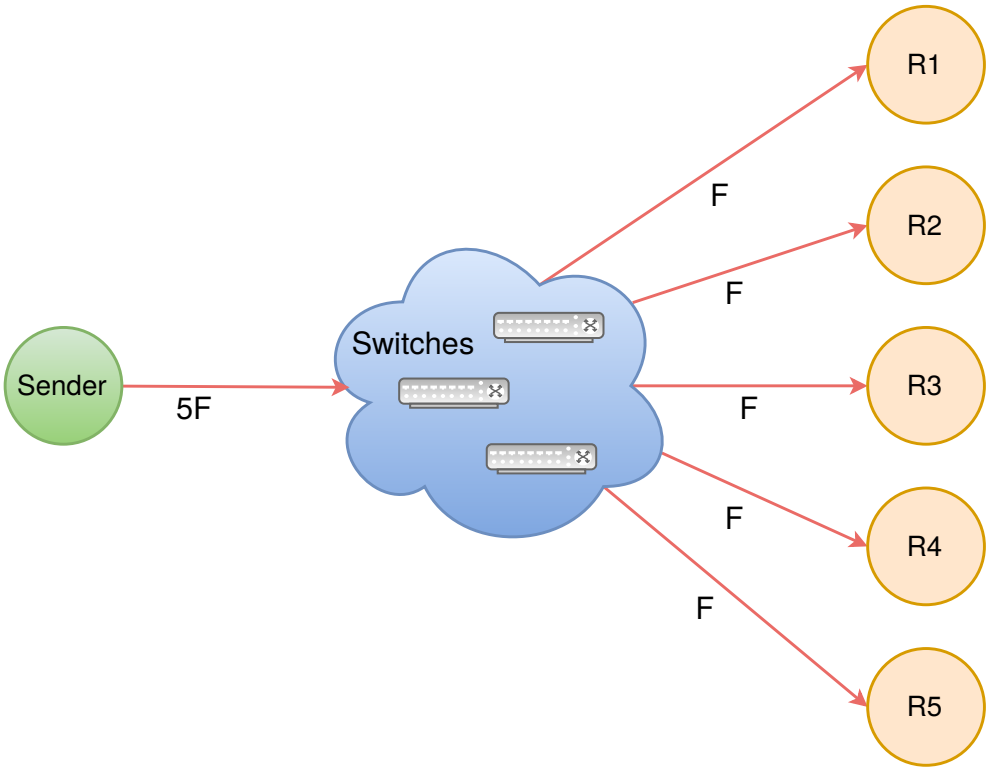

Figure 5.10: Parallel replication.

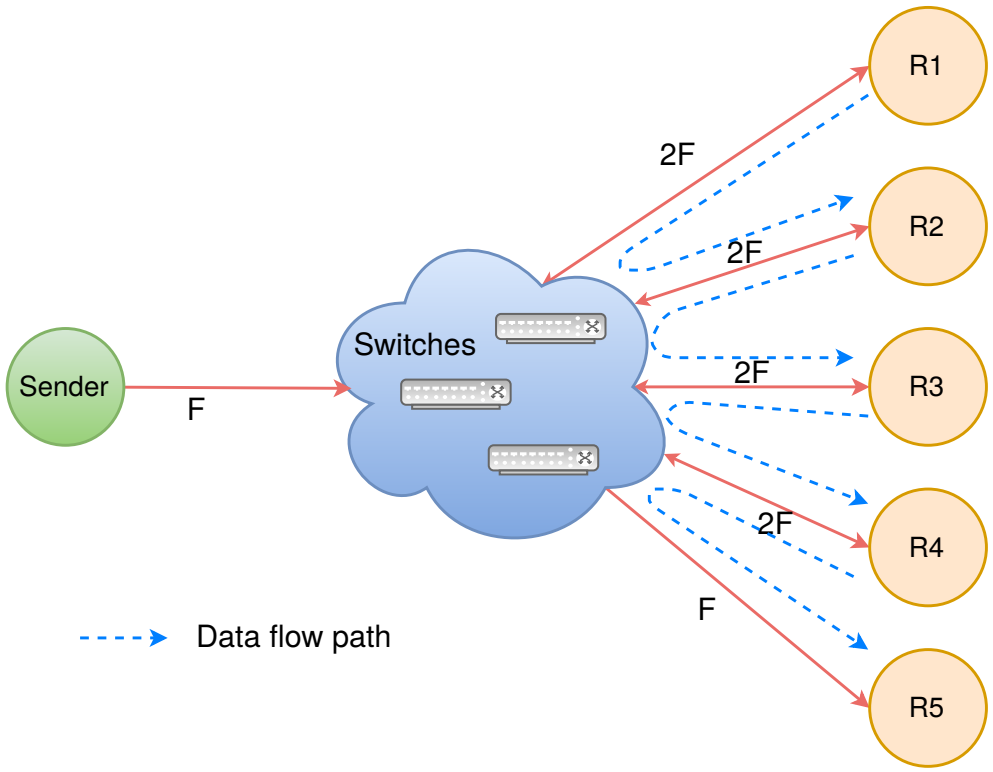

Figure 5.11: Pipeline replication.

Anka store uses pipeline replication method for block replication. Figure 5.12 shows the pipeline mechanism. A Client API sends a *copyFromLocal* request to initiate the process of uploading a file from a user machine to AnkaStore. The primary responsible gateway receives the request and collects "best available node" list. A candidate node list is constructed among the collected nodes and provided to the client API. The primary gateway creates an inode object for the corresponding file and replicate it on secondary nodes. After client receives the list of candidate nodes for the file, it constructs one pipeline per block. Number of pipelines depends on the file size, block size, and the number of nodes in the ring. The ultimate goal for AnkaStore when spreading blocks is to find best distribution so that a file is distributed as much as it can. The degree of block distribution determines the degree of parallelism when the file is needed for a computation.

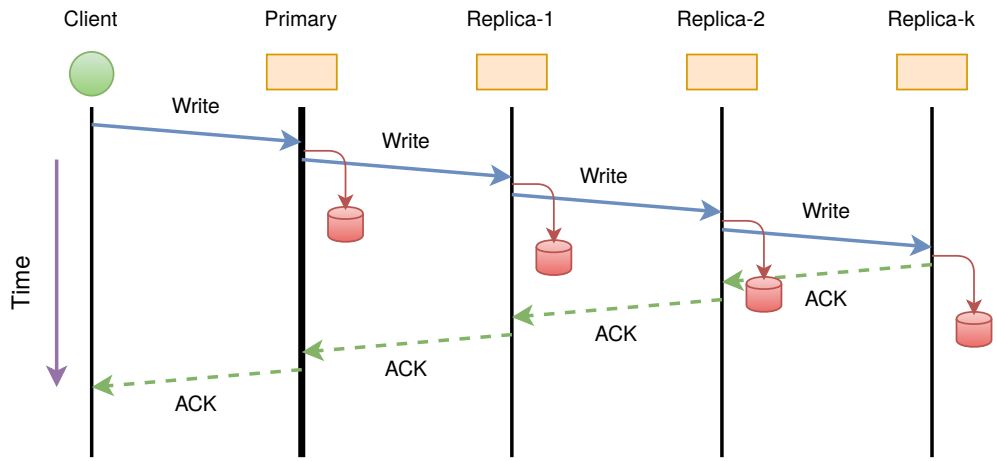

Figure 5.12: Pipeline replication detailed.

Client API sends a block as small packets that allow the system overlap the data transfers. The typical size of a packet is 64KB. The client sends packets to the primary DS in the pipeline. As the primary DS receives packets from client, it forwards packets to the next node in the pipeline. Once a packet reaches the last node in the pipeline, it sends ACK message to its immediate predecessor in the pipe, and this process continues until the ACK reaches the primary DS. The primary DS then sends ACK message to the client. This process happens at the packet level for all the blocks. If the primary DS receives the ACK for the last packet of a block, it informs the primary MS for the successful arrival of a block by sending a message, which contains file id, block id, and information for replicas.

Failure Detection and Handling

The SMC principle assumes that any component involving in AnkaStore system such as switches, CPUs, storage devices, or any other part can randomly fail at any time. Through the application-level retransmission discipline, in AnkaStore, component failures are treated as norm rather than exceptions.

Nodes have self-diagnosis functions through regular heartbeats. (Note that SMC application

does not require consistently available UVR. Heartbeats are only for UVR maintenance) Every node in the UVR periodically sends heartbeats to its successor to declare its aliveness. If a node does not receive any heartbeat from its predecessor for a certain time and specified number of retries, the predecessor is assumed failed. A $DECOMMISSION$ tuple is injected into the nodeManager tuple space to declare the failure. The failed nodes' predecessor and successor nodes who have either data for the failed node or have data on the failed node takes necessary actions to handle the failure. The heartbeat mechanism is also used for updating the finger table. The finger table contains information on UVR maintenance. Any change in the finger table of a node triggers neighboring nodes to update their finger tables as well.

As discussed, the AnkaStore content addressable network does not require strict UVR consistency. The system will continue to operate as long as the minimal survivable resource set is available for the respective user. Thus, even under catastrophic failures, the system will continue to deliver the best effort results until the very last resource.

5.5 Performance and Scalability Evaluations

Since HDFS and AnkaStore have similar design objectives, HDFS is an ideal target for performance, reliability, and service quality comparison studies. Both HDFS and AnkaStore are programmed in Java.

5.5.1 Experiment Setup

The experiments were conducted on the NSF (National Science Foundation) Chameleon bare-metal cluster at TACC (Texas Advanced Computing Center) [64]. 4 baremetal nodes are reserved for the tests shown in figures 5.13 and 5.14, and 8 baremetal nodes are reserved for the test shown in figures 5.15, 5.16, and 5.18. Each node has two CPUs with 10 cores,

67 GB memory, and 400GB SSD drive. One of the nodes for each experiment setup is used as client.

For HDFS, all four nodes are data nodes and the Namenode co-exists with one of the data nodes.

For Ankastore, all nodes except client run AnkaStore daemons.

The operating system is Centos 7, Java version is 1.8.

For AnkaStore, the minimal survivable resource set is 1 node. For HDFS, the minimal survivable resource set is 1-2 depending on the location of the namenode.

5.5.2 Service Quality and Reliability Evaluations

The automatic replication factor is set to 3 in all tests.

The HDFS was also designed for fault tolerance. Except for the Namenode, any data node crash can be automatically mitigated without causing the application to crash. HDFS is also optimized using RPC (remote procedure call) and pipeline parallel replication techniques. Thus the client-observable delay in large file movements can be minimized.

However, HDFS fails the "Share-nothing Test" due to the single namespace/namenode architecture. Synchronous replication helps in passing the A=B Test. At the time of this writing, it is difficult to evaluate the Constant Cost Test for HDFS. We will let the performance data to reveal some insights. HDFS was not designed to be transactional storage. Concurrent file updates are not supported.

AnkaStore is a pure peer-to-peer storage system. It passes the "Share-nothing Test" easily. AnkaStore default replication mode is synchronous. Therefore, it passes the "A=B Test".

The use of the replication Gateway allows it potentially pass "Constant Cost Test".

AnkaStore also optimizes large file replication using parallel pipelining. In theory, it should deliver comparable performance compared to HDFS.

For reliability tests, the six nodes are individually killed. HDFS's Namenode crash stops the application immediately. Performances after killing a data node in HDFS is comparable to the five-node AnkaStore test. The system continues operation until 3 consecutive nodes are killed. AnkaStore can still continue even if 3 consecutive nodes are killed for the other users who have their metadata on the alive nodes.

5.5.3 Performance Studies

A reliable large scale persistent storage system must implement automatic data replication. It is, however, not immediately obvious how to best exploit all network and computing resources for the automatic data replication task. For example, direct parallel data replication (Figure 5.9) from a single client to multiple data nodes performs badly after the file size passes a certain point. Investigation found that this level is tied to the maximal available network bandwidth. In comparison, pipelined replication (Figure 5.12) would be able to saturate the aggregate of available network bandwidths after the data size grows to a point.

Therefore, both HDFS and AnkaStore use synchronous pipeline parallel replication. Unlike HDFS's simple data recovery, AnakaStore implements transactional synchronous replication using the SMC protocol. Figures 5.14 and 5.13 report the AnksStore read/write performances against HDFS.

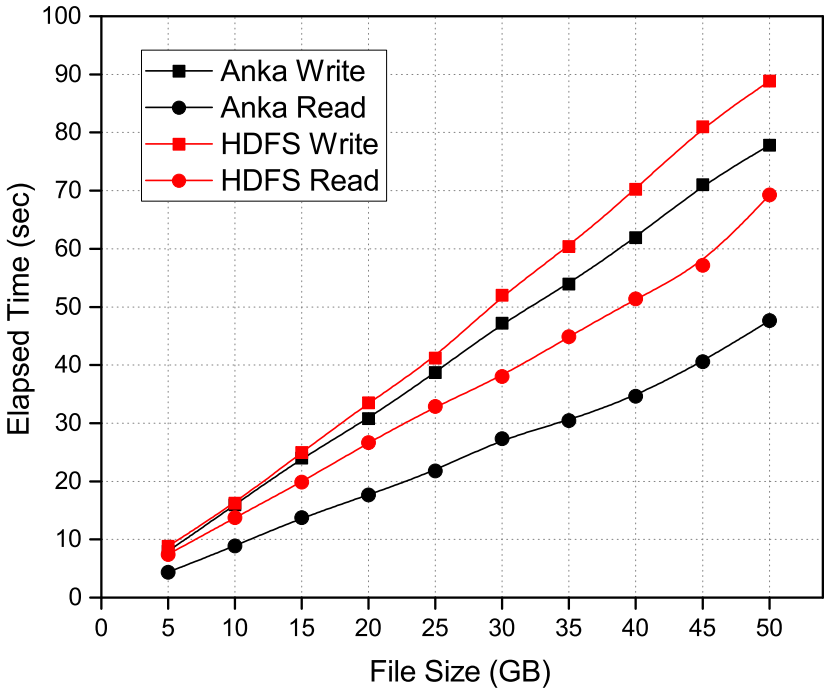

Figure 5.13: AnkaStore vs. HDFS (read/write times).

Figure 5.13 shows the write and read performance test results for both AnkaStore and HDFS. The tests in this experiment clearly show that AnkaStore write and read elapsed times are comparable with HDFS. Read operations deliver better latency results for both AnkaStore and HDFS because the read operation utilize all the nodes which have blocks for the desired file. A client reads a file from multiple sources. However, a client is the only source for writing a file to the systems.

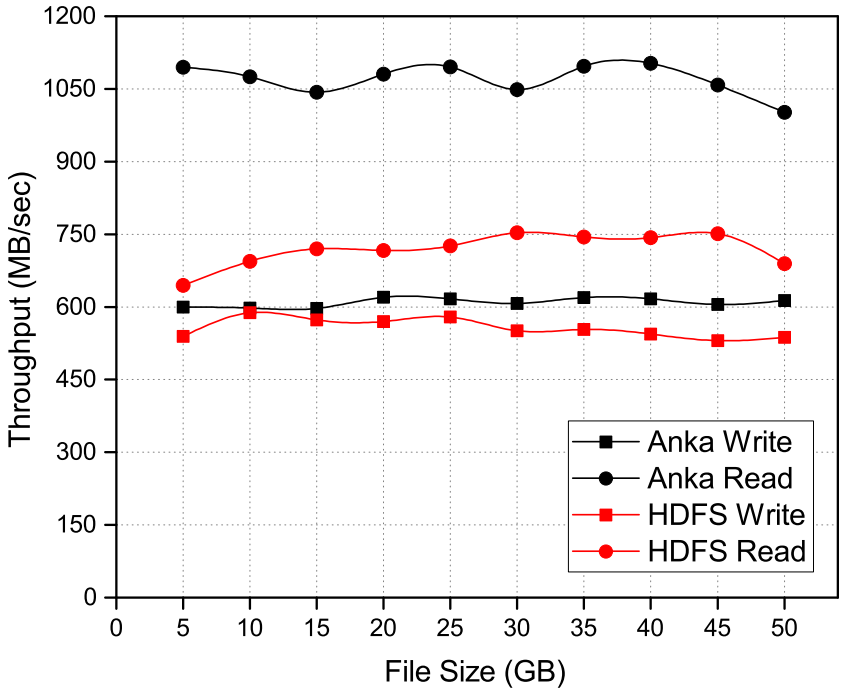

Figure 5.14: AnkaStore vs. HDFS (Throughtput in MB/Sec.)

Figure 5.14 shows MB/sec performance for AnkaStore and HDFS for both write and read operations. The experiments show that write operation for AnkaStore and write and read operations for HDFS deliver a throughput between 550 - 750 MB/sec. Read operation for AnkaStore can go over 1 GB/sec.

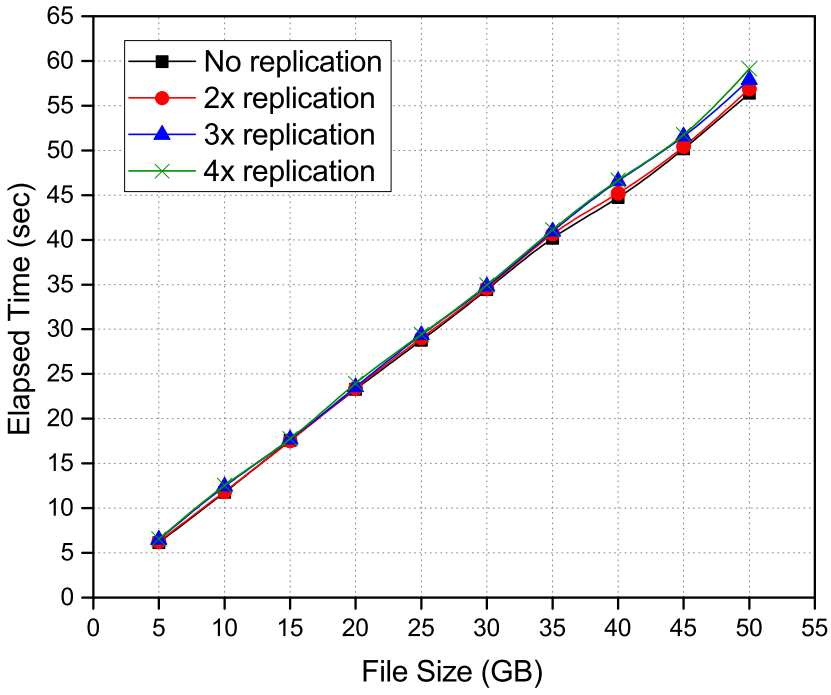

Figure 5.15: Replication Test

Figure 5.15 reports AnkaStore performance for replication factor $R = [1 - 4]$. The performance results suggest that increasing the number of replica for data from 1 to 4 does not have high impact on the system overall performance. Parallel replication schema cannot deliver the performance shown in this figure. Using pipeline replication schema can only deliver consistent performance for certain number of replicas by overlapping packet delivery.

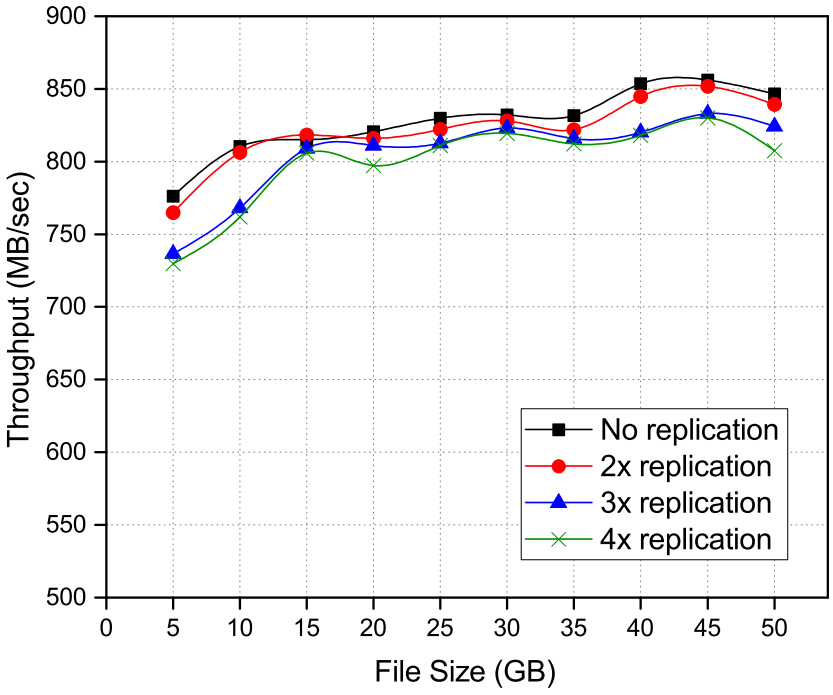

Figure 5.16: AnkaStore Throughput Scalability.

Figure 5.16 shows the throughput for multiple replica configurations when writing a file. As this figure suggests, changing replication factor between 1-4 does not significantly change the system overall performance.

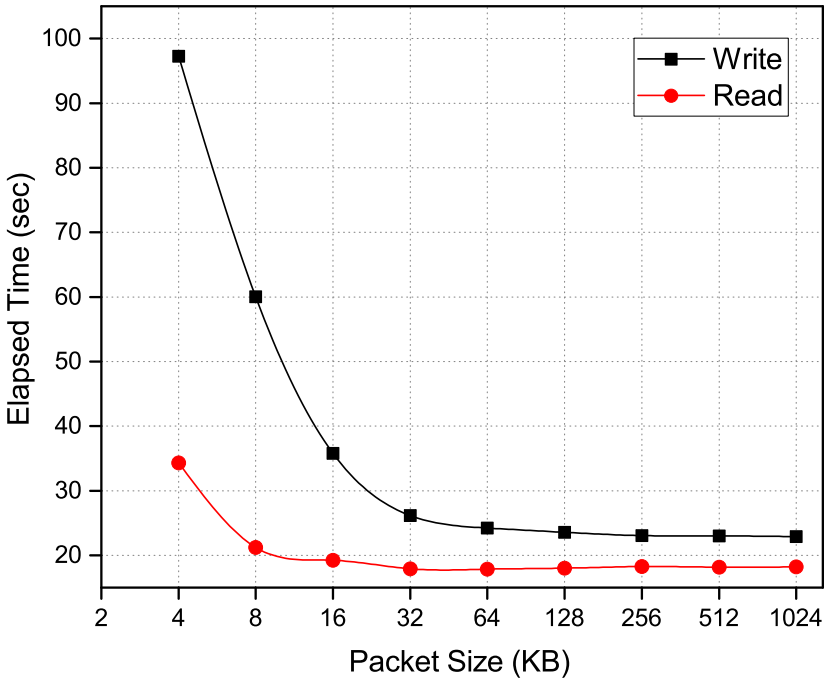

Figure 5.17: The impact of different application level packet size on system performance.

Figure 5.17 reports performance for different packet sizes for both write and read operations. The maximum TCP packet size is 64KB. Like TCP packets, each application level packet has an packet header overhead. Therefore, having an application level packet size similar to TCP packet reduces the overhead. The default application level packet size is set to 64KB to avoid packet header overhead.

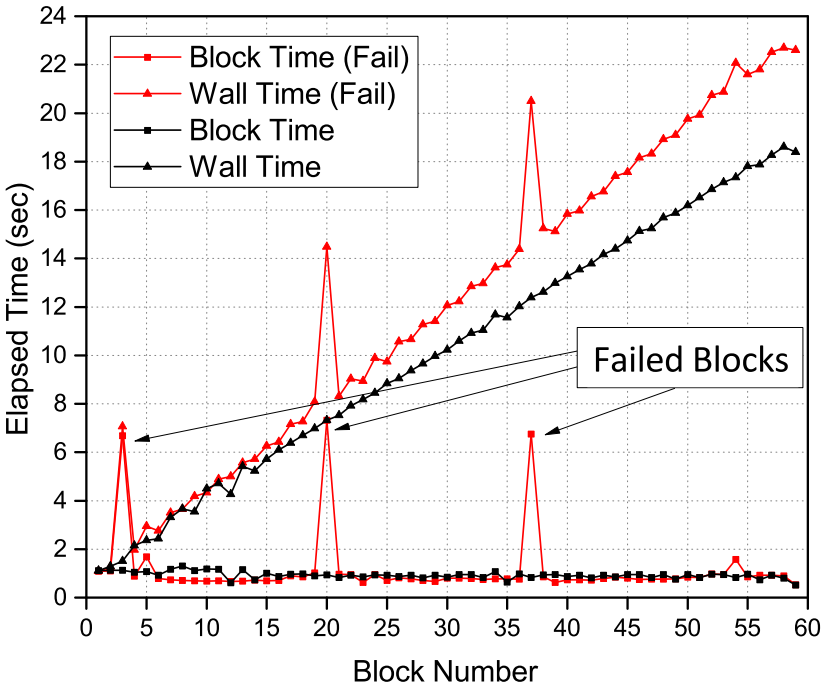

Figure 5.18: Failure test for 15GB file

Figure 5.18 depict the impact of node failure on system performance. In this experiment, an arbitrary node is manually killed right after we start the test. To see the affect of node failure, the individual block delivery times are measured and compared with a test without failure. The black lines show the results for the test without failure, and the red lines show the results for the test with failures. Each square on the lower lines show individual block delivery times, and the upper lines show the overall cumulative times for two tests. As we can see from figure 5.18, the block attempted to be delivered to the failed node by the client takes longer times. After the client realizes that the destination node has failed, it tries another node in the system, and delivers ultimately.

Although the overall file deliver time for the failed case takes longer, it is negligible. Anka-Store is able to provide service even in case of node failures.

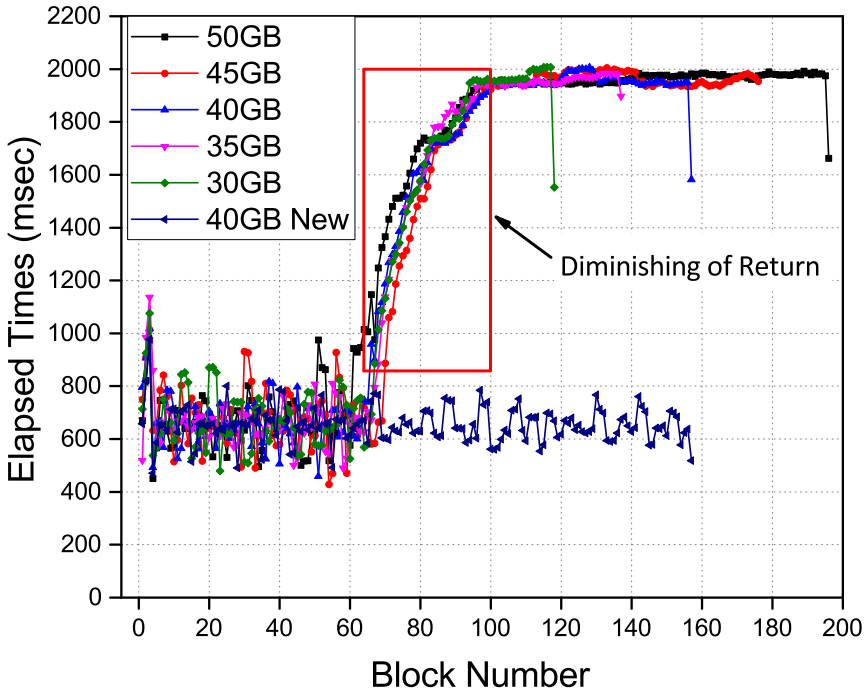

Figure 5.19: Diminishing of return when some resources are saturated.

Figure 5.19 reports several tests results for different file sizes for read operation.These tests are conducted to show "diminishing of return" where a resource is saturated. After the files uploaded to AnkaStore, a client reads files, in other words, downloads files from AnkaStore. Client side memory size becomes bottleneck for read operation since client receives file blocks from multiple nodes simultaneously. The area shown with red rectangle is where the client memory saturation point is reached. When we carefully investigate this issue, we found that the actual bottleneck is local storage write throughput. The default dirty page ratio set by operating system, the amount of memory to be used before flushing data in to hard drive, defines how much data can be accumulated in memory when the amount of received data exceeds the amount of data can be written in hard drive. All the tests consistently show that read operation slows down around 15GB. When we increased dirty page ratio, the saturation

point postponed as “40GB New” test shows this result.

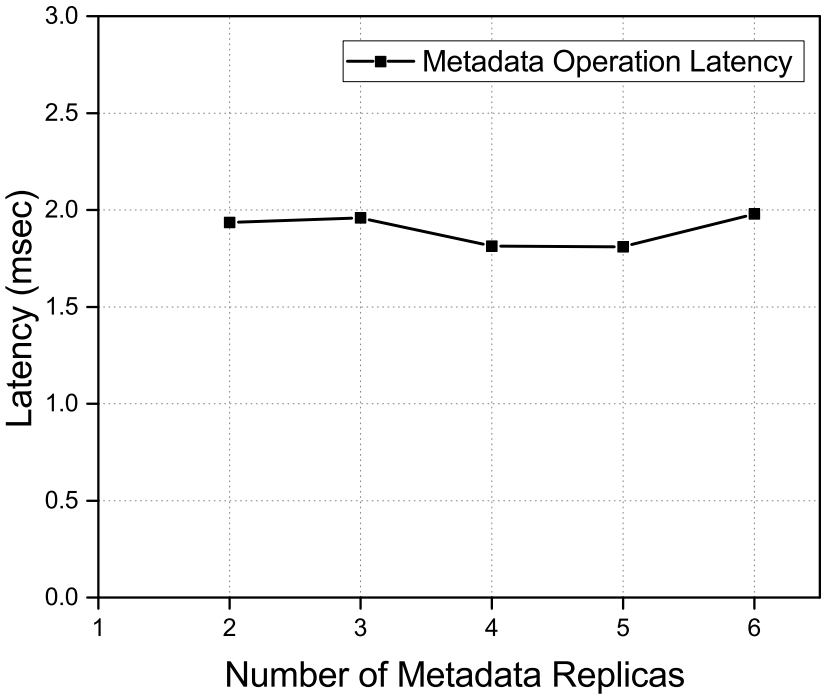

Figure 5.20: The effect of metadata replication on latency.

Figure 5.20 investigates the performance behavior of the system when we increase the replication factor for metadata. We set metadat replication factor for different configuration $2-6$, and report the results in figure 5.20. Experiments show convince that the overhead of increasing metadata replication factor up to 6 replica does not have high impact on performance since each metadata operation consists of small amount of data.

Chapter 6

ANKACOM: A DEVELOPMENT AND EXPERIMENT FOR EXTREME SCALE COMPUTING

6.1 Introduction to AnkaCom

Unhandled failures impede application's scalability. For parallel applications, the number of permanent and transient failures scales proportionally as the application grows in size. Regardless the efforts in reducing checkpoint overheads, all checkpoint-restart (CPR) methods have monotonically increasing costs as the application processing infrastructure grows in size.

In 2012, a concept of Statistic Multiplexed Computing (SMC) was proposed to handle transient software and hardware failures using the principles of packet switching. It was suggested that SMC applications would be able capable to deliver incrementally better performance and reliability at the same time. This paper reports the development and computational

experiments of AnkaCom - an experimental SMC parallel processing system.

AnkaCom is an experimental Java implementation of SMC as a distributed tuple switch network (TSN). The fundamental differentiating point of the SMC platform is the absence of reliability assumptions for computing and for networking components. Data parallelism is leveraged to ease the programming complexities and for automated performance optimization by overlapping computing with communication. For SMC applications, granularity tuning is required for peak application performance. In theory, a SMC application should deliver incrementally better performance and reliability at the same time without explicit checkpoints while up scaling the processing infrastructure. This paper reports that granularity tuned inductive AnkaCom experiments with arbitrary processor resilience can consistently outperform optimized OpenMPI program for a large (9000×9000) floating point dense matrix multiplication application in a dedicated HPC cluster of 12-120 processors. Implementation details of AnkaCom are also discussed.

Unhandled failures impede an application's performance and reliability. They prohibit extreme scale deployments. Transient and permanent failures scale proportionally as we increase the processing infrastructure sizes. These elusive errors crash applications causing losses in time, money and human lives. In 2012, it was reported that amongst all factors that could cause application crashes, the wide-spread use of the virtual circuit (VC) concept is the primary culprit[32]. The assumed VC reliability is not supported by existing communication protocols. Therefore, the use of VC leaves vast number of unhandled application states (timeout) that are directly responsible for the elusive crashes and data losses.

The 2012 report also proposed a Statistic Multiplexed Computing (SMC) framework for rectifying the unhandled failure problems. SMC is a generalization of the packet switching concept for general purpose distributed and parallel computing. It has three principles: a) a common unit of application data transmission or $< key, value >$ tuple, b) a store-and-forward application tuple switching architecture, and c) an end-to-end retransmission

discipline. Although (a) is currently common in large scale distributed and parallel systems, such as Hadoop and Spark, (b) and (c) remain absent. Theoretical studies showed that SMC applications are capable of delivering incrementally better performance and reliability at the same time. Thus they do not have scalability limits.

AnkaCom is an experimental Java implementation for distributed and parallel applications based on the SMC principles.

Although the SMC fault tolerance feature is easier to accept, it was not clear that if they could compete in performance against explicit parallel programs. This paper reports that SMC application running on optimized granularity can deliver consistently better performance against optimized OpenMPI application for a large (9000×9000) dense matrix multiplication application using a dedicated HPC cluster using 12-120 processors.

This paper is organized as follows. Section 2 introduces the SMC framework and explains why it could rectify the VC problems. Section 3 introduces the Tuple Switching Network and the unidirectional virtual ring – the critical component of the SMC machinery for compute intensive applications. It explains why most SMC applications do not need explicit checkpointing but a rare type of application to ensure result correctness. Section 4 describes AnkaCom and its implementation. Section 5 describes the setup for the computational experiments and reports the results of the computational experiments with explanations. Section 6 contains current status, section 7 and 8 describe summary and acknowledgments respectively.

6.2 SMC and Scalability

Currently, application scalability has varied interpretations for different communities. It is interesting to observe that the simple application performance and reliability requirements

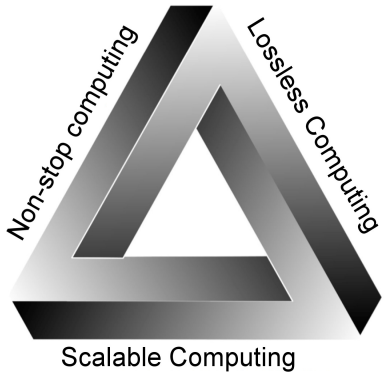

Figure 6.1: The impossible triangle.

got "moulded" into different shapes and forms. In practice, things are simpler. Application performance and application reliability are the two very basic requirements. When the processing infrastructure needs expansion in order to handle application's growth, the basic requirements should not be altered. However, this has been considered too difficult or impossible, the "performance-only scalability" measure has been dominating.

A closer look into this parallel application scalability difficulty reveals a grid lock of impossibilities.

Figure 6.1 illustrates the impossible triangle formed by connecting three practically desirable but theoretically impossible requirements: "Non-stop computing", "Scalable Computing" and "Lossless Computing". The word "impossible" was derived directly from the theoretical impossibility studies [15].

Ironically, the similar challenges existed for data communication network architectures if the word "computing" is replaced by "communication". Approximately five decades ago, the packet switching (or statistic multiplexing) network was proposed [16] for meeting the data communication challenges of a similar grid lock. Although the proposal was broadly rejected initially, today's Internet is not possible if we had not endured the packet switching paradigm shift. The packet switching network has proven the feasibility of the impossible triangle by infinitely approaching the ultimate solutions using massively many unreliable components.

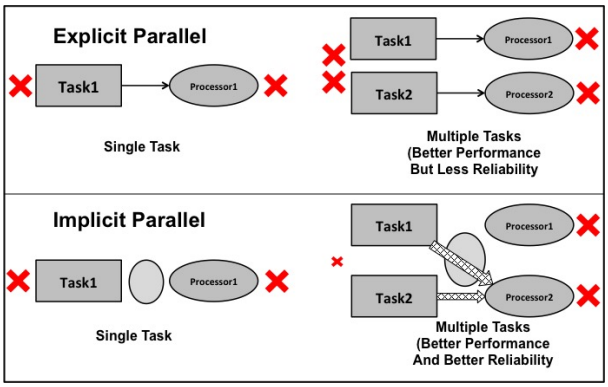

Figure 6.2: Inductive Parallel Architecture

In contrast, today's distributed and parallel applications are built based on the assumptions of virtual circuits (VC). The use of virtual circuits forces each application to form an application specific network starting from the first instance of its running. Any transient software or hardware failure can break this network crashing the entire application. The probability of these failure increases proportionally as we up scale the processing infrastructure. Since the communication protocols only support packet-level reliability, most existing distributed and parallel applications fail at large scales. Statistic multiplexed computing or SMC breaks away from this tradition by leveraging the proven packet switching principles for extreme scale computing.

In this paper, parallel application scalability is defined as the application's ability to deliver incrementally better performance and reliability at the same time as we up scale the processing infrastructure. Strictly speaking, extreme scale computing is not possible for applications that do not scale.

To show the feasibility of parallel application scalability, Figure 6.2 illustrates the inductive nature of SMC application architecture as compared to the VC-based explicit parallel application architecture. In this figure, the circles represent processing nodes and rectangles represent application tasks. The number and size of the red-crosses indicates the probability of transient failures. The left side of Figure 6.2 contains the base cases, $P = 1$, for explicit

parallel systems, such as OpenMPI, and for implicit data parallel system such as Synergy [33] where the probability of application failures are identical for implicit and explicit systems. The induction cases are on the right side of Figure 6.2, where the explicit parallel program enjoys the increased performance but suffers the loss of reliability when more processors are involved. Due to the "soft binding" of the statistic multiplexing mechanism, the SMC architecture can deliver better performance and reliability at the same time incrementally (indicated by the smaller red cross). According to the principles of mathematical induction, the SMC application does not have scalability limits. Note that the economic rules of "diminishing of returns" still apply for each application with a fixed problem size.

6.3 Tuple Switching Network

The Tuple Switching Network (TSN) (Figure 2.4) was designed to implement the principles of SMC for practical distributed and parallel processing. There are two design goals: a) ease application programming and b) deliver unlimited scalability by not assuming reliable (computing and communication) components ([1]). The TSN framework combines the tuple space API (application programming interface) ([49]), data parallel semantics with a statistic multiplexed tuple space engine. In particular, the tuple space API delivers the ease of programming using the popular $< key, value >$ pair convention with implied data parallelism ([33]) for automatic task scheduling and load balancing. The statistic multiplexing engine eliminates all single-point failures for SMC applications. Unlike the traditional explicit-parallel commands, such as $< send >/< receive >$ and $< read >/< write >$, the $< key, value >$ pair tuple space API supports three implicit parallel operators: *read*, *put* and *get* [49] and [33]. This simple API allows the SMC runtime to inject the statistic multiplexing effects for both computing and communication components. Specifically, a unidirectional virtual ring (UVR) was designed to statistic multiplexing user-defined tuples in order to

leverage potentially massive number of network switches and processor nodes. The use of UVR breaks the traditional dependency on virtual circuits allowing the elimination of all single-point failures. This also means that SMC applications are capable of tolerating massive permanent and transient software/hardware failures while delivering unlimited scalability.

Programming ease was one of SMC design objectives. Unlike MPI parallel programs where the programmer specifies both the master and worker in the same program body, a SMC application has the parallel master and parallel worker in separate files for each compute intensive kernel. This style of coding is similar to serial programming making it possible to diagnose the parallel program using sequential debuggers. Unlike MPI programs that are automatically activated on all processors from the start, the SMC programs are only activated on available processors in parallel when the matching $< key, value >$ pair(s) become available. Since the TSN runtime ensures that each $< key, value >$ pair is processed by at least one processor (using a technique called shadow tuple [33], the worker crashes can be easily recovered without explicit checkpoints. Even under extreme safety conditions where each tuple is replicated 2-3 times, the overhead of tuple replication is surprisingly small (see the performance reports in the later section). To complete the statistic multiplexing protocol for user-defined tuples, the application program is responsible for the elimination of redundant result tuples as part of tuple retransmission discipline. Under the SMC framework, master checkpointing is only necessary for one out of four application stability types (Figure 6.3), if multiple masters are launched in parallel [44].

6.4 AnkaCom

AnkaCom is an experimental Java implementation of TSN. In addition to compute intensive applications, it's design also includes the preliminary support for data intensive applications where each data state change must be permanently preserved. The freely scalable archi-

		Output	
		Deterministic	Non-Deterministic
Input	Deterministic	Supported. Examples: Linear and nonlinear solver, FD, Molecular Simulation, etc.	Supported. Examples: Parallel Search Unstable Parallel Sort, etc.
	Non-Deterministic	Not Supported. Examples: Parallel Random Search, Random date-time based Reduction.	Supported. Examples: Monte Carlo Simulations, etc

Figure 6.3: Application Stability Types

tecture for data intensive parallel applications requires breaking the CAP limitations [30]. A counter-intuitive gateway based data replication architecture is required [36] in order to build the statistic multiplexed transaction (SMT) network. This paper reports the AnkaCom implementation that incorporates the gateway based architecture for in-memory high performance computing. As data intensive high performance computing becomes indispensable for meeting the current and future scientific discovery and social computing needs, the reported AnkaCom experiences can shed a different light on the commonly considered impossibilities.

6.5 System Architecture

AnkaCom is a distributed peer-to-peer parallel processing system. Every node in AnkaCom runs a daemon with four components: a) NodeManager responsible for the UVR formation and maintenance, b) Gateway responsible for tuple coordination and replication, c) StorageEngine responsible for implementing the tuple space functions and future file system

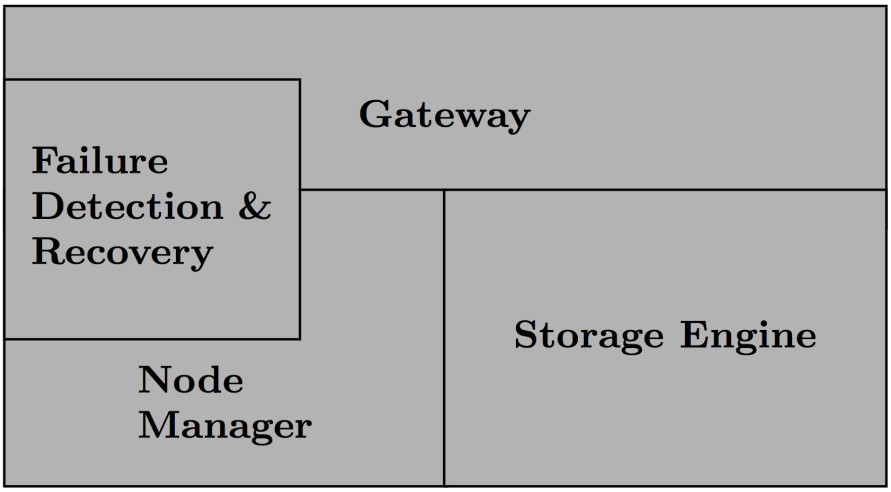

Figure 6.4: AnkaCom daemon architecture.

interfaces, and d) FaultManager responsible for failure detection and handling.

6.5.1 NodeManager

The NodeManager runs on all computing nodes. It is responsible for maintaining the global UVR. The UVR implementation borrows the "consistent hashing" concept in the Chord system[45]. Every node is capable of switching application level tuples within the global tuple space context employing massively many computing nodes and network switches. A node communicates by injecting its $< key, value >$ tuple into the global tuple space. The tuple switching algorithm delivers the tuple to its destination via UVR. The tuple switching algorithm allows exploitation of multiple switches in parallel. Transient and permanent software/hardware failures can cause arbitrary number of nodes dropped off from UVR. New or recovered nodes can also join UVR at arbitrary time. The current AnkaCom implementation forms the UVR based on ordered RingID using the Jenkinson hashing algorithm[71]:

$$RingID = H(DataCenterID, RackID, IPAddress, port)$$

After the new node receives a *JOINOK* tuple, it sends a *JOINCOMMIT* tuple, which completes the UVR expansion task.

NodeManager is also responsible for its own failure detection and recovery. It sends heartbeats to its predecessor node. If a node does not respond in some specified time, it's marked as failed. A *DECOMMISSION* tuple will be sent to replace the failed node with the next alive node in order to maintain a valid UVR structure. More details on UVR can be found at [3]

6.5.2 Gateway

Gateway is responsible for synchronous data replication that may encounter race conditions. As indicated in [50], the gateway delivers dynamic serialization for eliminating race conditions, timeout-retransmission discipline without compromising the data change ACID properties, and non-stop resynchronization for reconciliation of inconsistent datasets. The gateway ensures ACID properties are enforced for all data changes while performing synchronous parallel data replication whenever there are no update conflicts. The non-stop data resynchronization algorithm is a new form of 2-phase commit protocol that does not force service shutdown when replicas return inconsistent states. Instead, an arbitrary replica is selected to service clients while forcing others to resynchronize. This new form 2-phase commit protocol does not impose performance penalty. The cost is a brief (less than 60 seconds) worst case downtime per resynchronization incident, independent of dataset sizes. The non-stop resynchronization algorithm is not yet implemented in AnkaCom.

Unlike compute intensive applications, the timeout/retransmission discipline for data intensive requests varies according to different data access types. The gateway will not automatically retransmit data if the application is transactional. However, retransmission discipline for key-value pair tuple is necessary. This is similar to Cassandra [39] and HBase [73] systems.

The retransmission discipline is the integral part of the SMC framework in order to counter the rapidly declining MTBF (Mean Time Between Failures) of large distributed systems including the high performance computers [74], [75]. Unlike other systems, SMC applications are also responsible for completing the SMC protocol by including the application-level redundant result elimination logic.

The non-stop resynchronization algorithm is required for completing the new 2-phase commit protocol when one or more replicas become inconsistent due to permanent or transient failures. This algorithm is applicable for dynamic UVR expansion as well as dynamic UVR repair when a failed node rejoins the ring.

In the current AnkaCom implementation, every node runs a gateway responsible for all data coordination with other nodes in UVR and the outside world. The gateway functions are the top layer functions in the NodeManager.

Data parallelism of Tuple Space operation semantics allows automatic data migration towards available computing resources. Tuple distribution and replication mechanisms can also be configured to suite specific applications. For example, the user can distribute tuples based on some hashing scheme. In this scenario, each node is assigned for a partition of key set when it joins the UVR. Once the ring becomes stable, the key distribution is done for the first time. Later if a node wants to join the ring, a portion of the responsibility partition of its predecessor and successor is assigned to the new node. Each node's range is calculated with the following equation.

$$Range_i = \left[\frac{(S-i) \times INT}{S} + 1, \frac{(S-i+1) \times INT}{S} \right]$$

where S is the total number of nodes in the UVR, i is the current node, and INT is the max integer value. *write* and *read* requests are first received at the gateway that is only

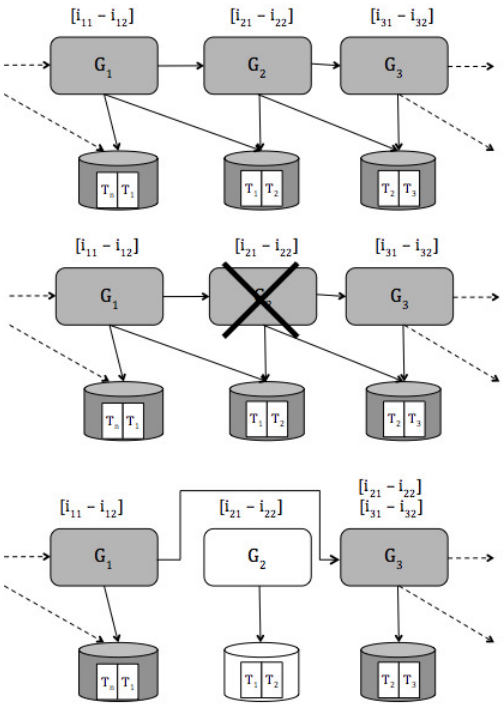

Figure 6.5: Failure Detection and Handling.

responsible based on its assigned ranges. The *read* request can be satisfied by any node if it has the requested data. Therefore, each node is also assigned to its predecessor(s) partition to replicate data for them and to service read requests.

On the other hand, all the *write* requests are forwarded to the corresponding gateway. The number of replicas is defined by the replication factor configured before the application started. For example, if the replication factor is 3, then each node has one key range for itself to service write and read requests and 2 key ranges for its predecessors to service read requests and maintain data durability.

The gateway checks the data request whether it falls into its partition. If it is responsible for the request, the request is serviced against the local tuple space with the results stored in the same space and its replicas. If the request cannot be satisfied by the local tuple space,

the request is forwarded to the next node, until either the TTL (time to live) counts down to zero or it finds a satisfying node on UVR. Since each node has m ($m = logn$, where n is the total number of nodes in the ring) successor nodes in its finger table, a tuple finds the desired node at most at $logn$ hops.

6.5.3 Storage Engine

This component stores tuples in memory in the current implementation. Each storage engine stores for local tuple space and replica spaces for other nodes. It only serves local and remote tuple requests against the local tuple space. The future file system interface will be placed in this component.

6.5.4 Failure Detection, Handling, and Recovery

AnkaCom has two failure detection mechanisms. The first is the NodeManager's heartbeats. Nodes fail to respond within specified time will be considered "dead".

A $DECOMMISSION$ tuple will be sent to inform others in UVR. The "dead" node will be replaced by a alive node in its successors list. The timeout of every tuple operation is the second failure detection and recovery mechanism that can automatically recover multiple component failures. Figure 6.5 demonstrates a failure handling process. The top row in the figure shows a stable state in which all the nodes are alive. The second row shows that the middle node, G_2, fails to respond to any request. G_2 is marked as failed by G_1 and removed from the UVR. G_3 becomes the successor of G_1 and the UVR is repaired. Since G_2 was removed, its responsibility range is assigned to its successor G_3 and changes the state of T_2 from *read* to *write*. The *write* requests for that range will be forwarded to G_3. These activities are very fast and transparent to the user. Whenever a node joins or decommissioned from UVR, the nodes update the finger table, which maintains the vital

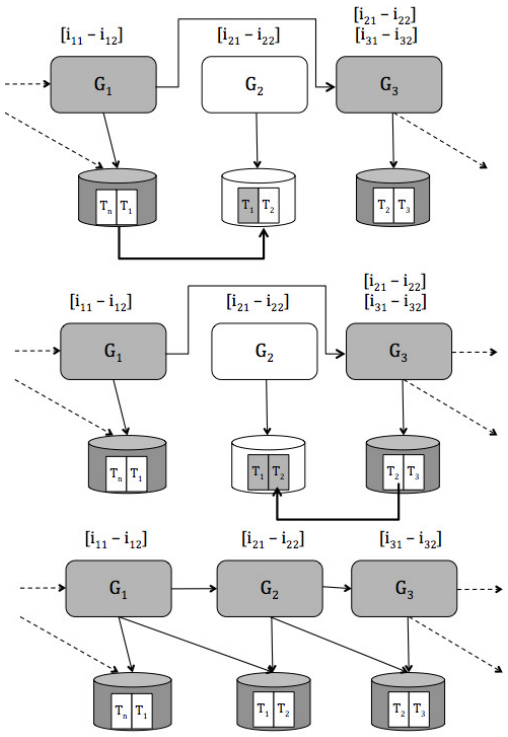

Figure 6.6: AnkaCom Node Recovery.

information of UVR. Note that it is possible to have multiple isolated UVRs running in parallel due to massive transient failures at the same time. Each is self-consistent and capable of running applications on its own.

When a node joins or rejoins the UVR, its arrival is announced to the ring. It is considered to have zero history. The UVR recovery process will build the new node to become consistent with others by recovering its local tuple space (and replicas) from its new predecessors and successors. AnkaCom invokes the resynchronization process using the nodes' tuple update logs. Depending upon the application, a brief pause on the new predecessor and successor nodes may be necessary for the resynchronization process to complete. Figure 6.6 shows this recovery process. The above mentioned mechanisms can tolerate massive permanent and transient worker failures. Launching multiple redundant masters can deliver master fault tolerance for three out of four application types [44]. For compute intensive applications, all computational states move forward. At least one master will eventually complete the job.

6.6 Application Programming Interface

The current AnkaCom implementation supports two operations:

$put(\text{"}KeySpaceName\text{"}, tuple, G)$

$get(\text{"}KeySpaceName\text{"}, \text{"}TuplePrefix\text{"})$

```
//Code logic for Master process
Client client = ankaStore.getClient();
ankaStore.createKeySpace("INPUT");
ankaStore.createKeySpace("OUTPUT");
MatrixTuple tupleA = new MatrixTuple("A", A);
ankaStore.put("INPUT", tupleA, G);
ankaStore.get("OUTPUT", "C");
for (int i = 0; i < numResults; i++) {
        tuple = client.getRecReqQueue().take();
        mt = tuple.getPayload().getValue();
        for (int j = 0; j < mt.numRows; j++) {
           result[mt.startRow + j] = mt.data[j];
        }
     }


//Code logic for Worker process
ankaStore.get("INPUT", true);
while (true) {
   tuple = client.getRecReqQueue().take();
   mt = tuple.getPayload().getValue()
   C.id = "C_"+ A.getRequestNum();
   C_ = process(A.data, B.data);
   ankaStore.put("OUTPUT", C);
```

```
}
```

Note that unlike explicit parallel systems, the AnkaCom implicit parallel API allows the application to alter processing granularity after compilation. In other words, the "payload" can be of arbitrary size allowing manual or automatic granularity tuning without recoding. For explicit parallel programs, such as MPI, the granularity is fixed: N/p. This is the new performance optimization dimension only available to SMC applications.

6.7 Wrapping Existing Applications

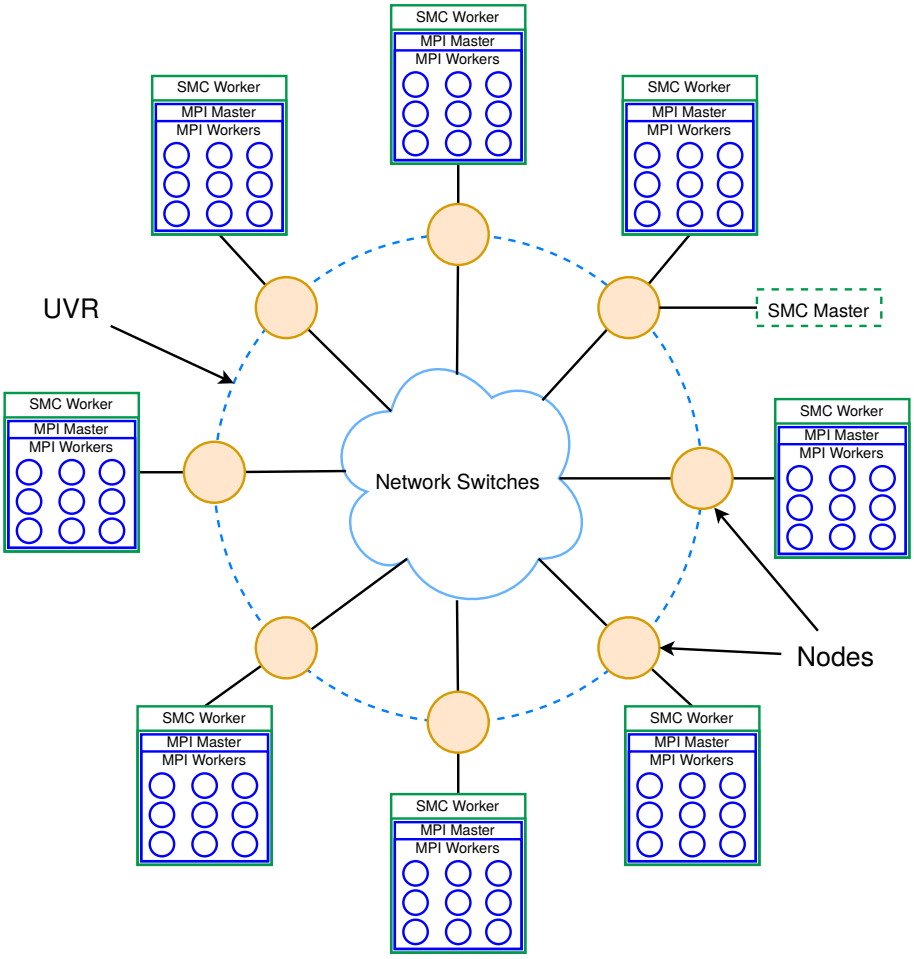

Figure 6.7: Wrapping existing MPI application.

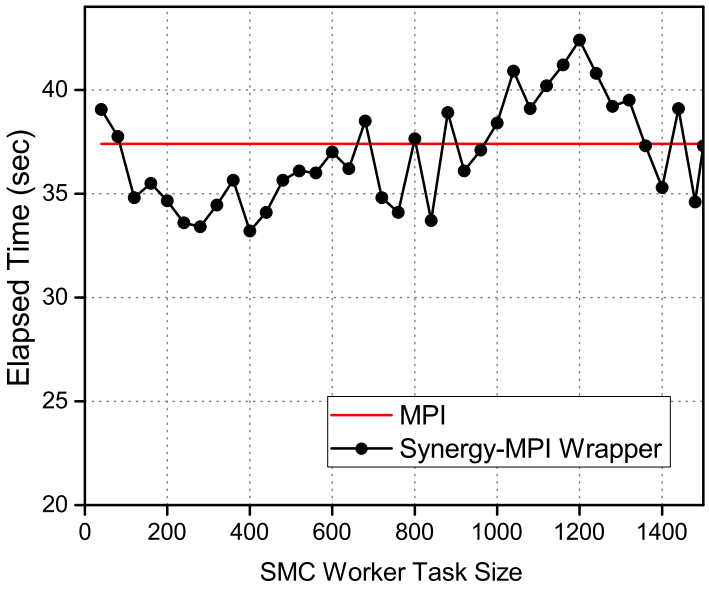

Figure 6.8: SMC wrapper performance.

A complete programming paradigm shift is generally impractical and costly since most of existing scientific applications have been developed by several contributors all around the world. The SMC paradigm is designed to support wrapping current applications. Figure 6.7 represents a concept of SMC wrapper with MPI. It shows an UVR ring with 8 nodes. Each node runs a SMC worker that wraps an MPI master, which forks several MPI workers. One of the nodes runs SMC master. SMC master first generates tasks and distributes them to SMC workers. An SMC worker then runs a MPI master on a received task. The role of SMC worker is to receive bigger tasks from SMC master and run spawn a MPI master on the task. Finally, the MPI master forks several workers on the received task by dividing the task into smaller tasks. After the MPI master receives results from MPI workers, it informs the SMC worker for the task completion. The SMC worker then collects the result and put it into distributed tuple space. There are two level of tasks sizes for wrapping applications. If a problem is distributable and can be computed in parallel, it can be further distributed by SMC framework with wrapper. Wrapping existing applications has three advantages:

(a) Reducing the size of current application's sizes to smaller tasks, which reduces the probability of application failure, eliminates application check pointing.

(b) Having ability to tune the task granularity which ensures time equilibrium for task completion.

(c) preserving the integrity of the legacy application logic.

Figure 6.8 shows SMC wrapper performance. In this experiment, 4 nodes each with 4 cores run two sets of tests. The first test finds the performance for 6000 x 6000 matrix multiplication MPI application. The second set of tests is to find performance for the same application using SMC wrapper with MPI application. The results suggests that there are some tasks sizes where SMC wrapper delivers better performance.

6.8 Computational Experiments

The reported computational experiments were designed to answer the following questions:

1. Can SMC application deliver incrementally better performance and reliability at the same time?

2. When will the rule of "diminishing of return" apply?

3. How can SMC application win the performance contest against MPI?

4. How much overhead would it be if the user requests triple redundancy $R > 2$?

6.9 Experiment Setup

The computational experiments were conducted on Temple's Owlsnest [65]. All the tests run on the "Normal queue" of multiple Dell R610 with 2x Intel Xeon X5660, 2.8GHz, 12 cores, 12GB memory, 120GB local disk space. These nodes were connected by a 10GigE Ethernet and a QDR Infiniband switch. The software includes JDK7 for AnkaCom and OpenMPI 1.4.4 for the MPI matrix multiplication application. The Linux distribution is Ubuntu 12.04 for RHEL 6.2. Each time measurement was taken three times and we report the average for each run. We used the naive matrix multiplication program with $O(n^3)$ complexity for two different matrix sizes, 6000 and 9000. The choice of matrix multiplication algorithm is irrelevant for the final performance comparisons since both targets use the same algorithm. For parallel processing, both MPI and AnkaCom use the same resource allocation strategy: every node hosts as many worker as the number of cores except for one node that will host workers and the master.

6.10 Computational Results

To prove the scalability concept, we started with a test of 2 nodes (12 cores total) and expand the size of the serving nodes one by one up to 20 nodes (240 cores total). Each node runs an AnkaCom daemon and 6 workers except for one node, which also runs master application.

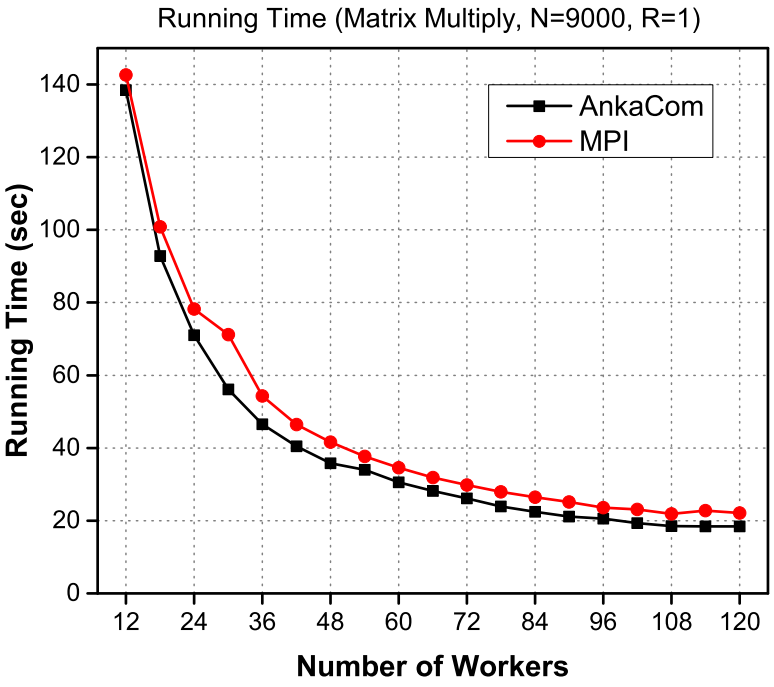

Figure 6.9: Scalability Test Results.

Figure 6.9 shows the results for this experiment. All matrices are 9000×9000 squares. The AnkaCom performances were obtained by fine tuning granularity for different core counts. The MPI performance was obtained by averaging three runs. Both AnkaCom and MPI scale nicely up to 18 nodes (216 cores total). This cut-off line reflects the rule of "diminishing of return" for the matrix size of $9,000$.

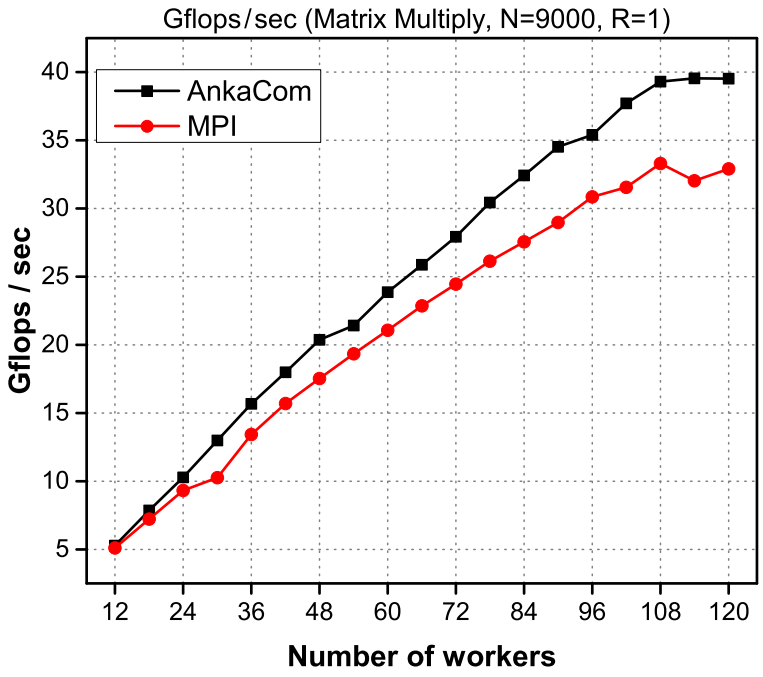

Figure 6.10: Harvested Computing Powers.

Since the SMC application was run with full failure protection turned on and MPI run does not, Figure 6.10 shows that the SMC application meets the new scalability definition. According to the discussions based on Figure 6.2, unlimited scalability is also proved. In other words, for larger matrix sizes, adding processors can deliver better performance and better reliability at the same time while observing the rules of diminishing returns. There exists no scalability limit for this SMC application given unlimited resources.

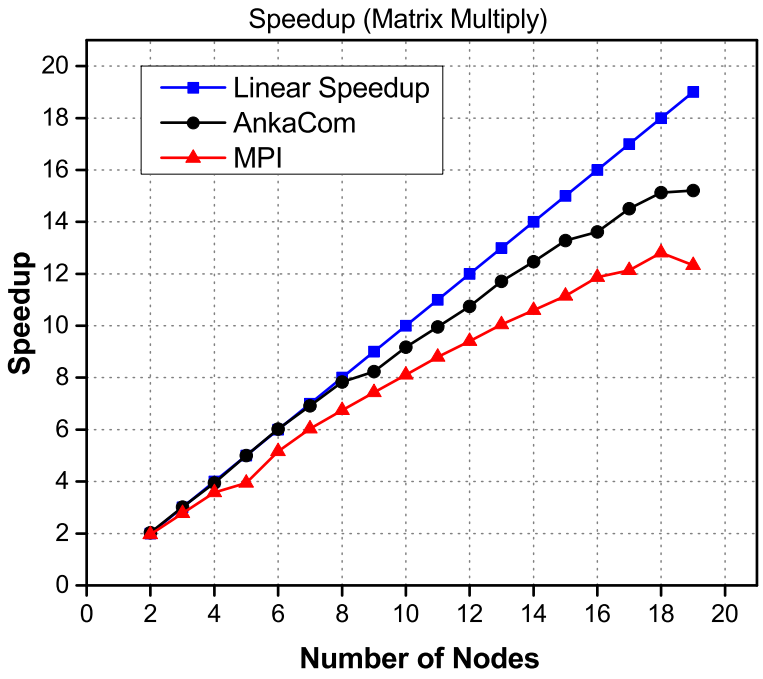

Figure 6.11: Speedup Measures.

Figure 6.11 shows that AnkaCom delivered surprisingly consistently better performances against the "raw metal" MPI program, given that the MPI C program was compiled using $-O3$ switch with optimized loop orders where the Java compiler has no optimization option. AnkaCom results are closer to the theoretical linear speedup.

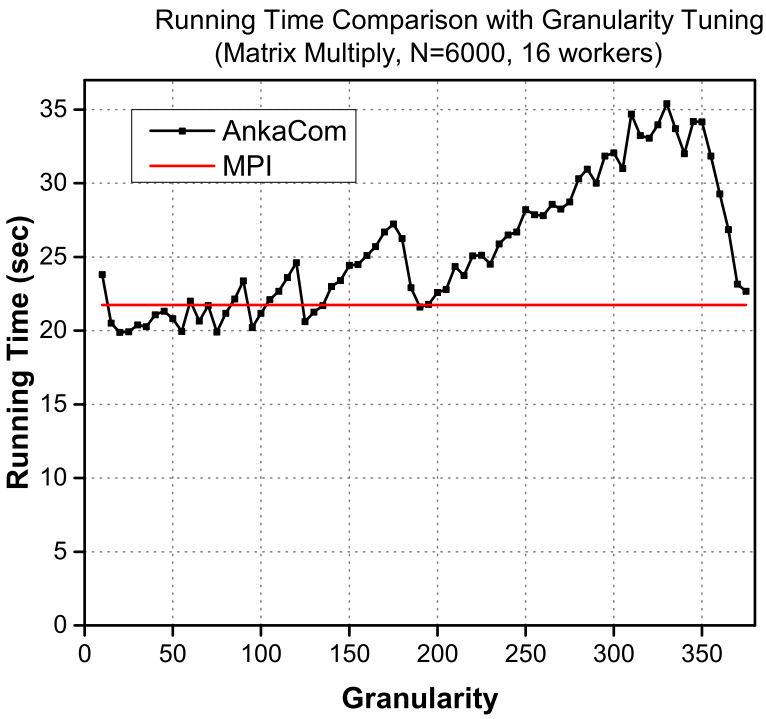

Figure 6.12: Performance effect of granularity tuning.

Figure 6.12 shows the detailed effects of granularity tuning. Other studies [76], [77] also reported similar effects in finding the optimum synchronization point for parallel applications using heterogeneous resources. It is not clear why granularity tuning would make such big performance differences in dedicated homogeneous environment such as the Owlsnest. Closer inspection finds that all processors share the same interconnect(s). The distribution of the communication capacities is non-deterministic. This results in non-uniform task completion times. Thus, the load balancing problem for homogeneous processors becomes identical to heterogeneous processors. The optimal granularity allows the maximal overlapping of computing and communication tasks resulting in the shortest elapsed time. Finer task size increase the probability of task computation start time and overall task completion time, which is defined as "Time equilibrium."

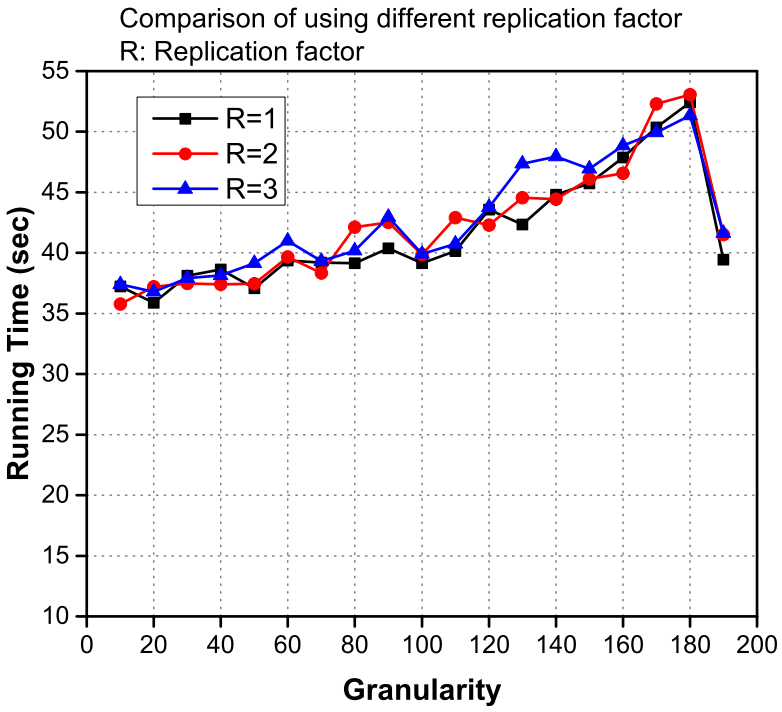

Figure 6.13: Replication Overhead Impacts.

Figure 6.13 reports the computational experiments with $R = 1-3$ (number of replicas). Since the AnkaCom application runs purely in memory and there are multiple network switches in the high performance cluster, the replication overheads are minuscule compared to the overall SMC performance advantages. The first node receives a replica starts processing task right away without waiting replication completion. Therefore, task processing and data replication are overlapped.

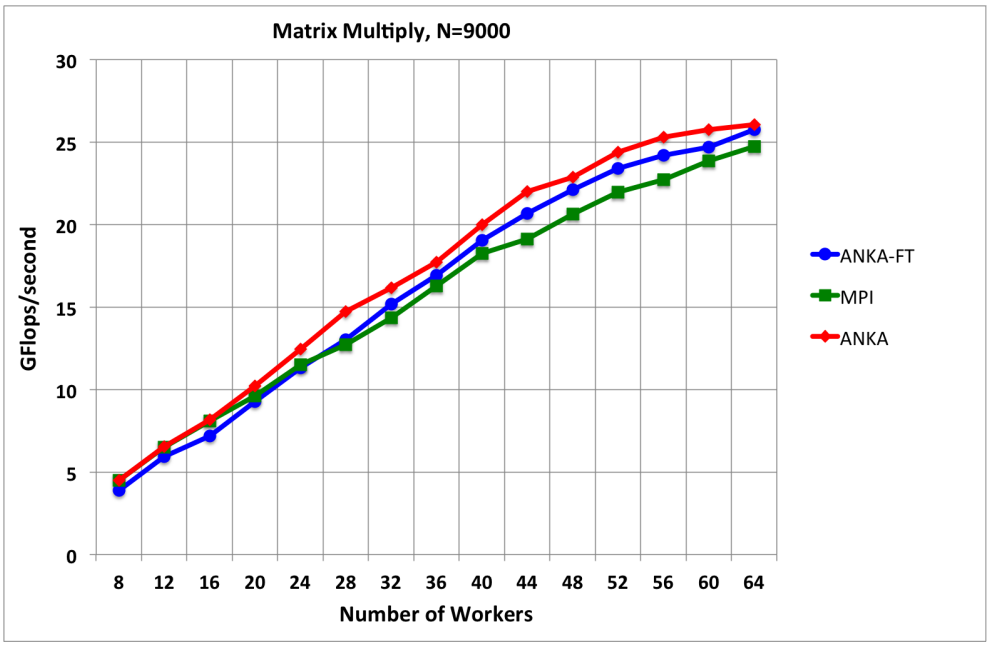

Figure 6.14: Failur impact of performance.

Figure 6.14 reports the failure effect on overall AnkaCom performance. In this set of experiments, we compute 9000×9000 matrix multiplication program and killed one node for every experiment right after the experiment starts. We also run same test for AnkaCom and MPI without injecting failures. ANKA-FT line shows the experiment results with failure injection. The results suggests that the effect of failure on performance is minimal.

Chapter 7

CONCLUSIONS AND CONTRIBUTIONS

7.1 Conclusions

In this dissertation, we investigated the practical feasibility of the concept of Statistic Multiplexed Computing or SMC. The SMC framework was designed to eliminate the wide spread fallacy in distributed and parallel computing in existing production and experimental systems. Although "assuming the network is reliable" was cited as the top fallacy in distributed and parallel computing, it was never clear exactly what it means and how it should be rectified. Consequently, the hop-by-hop protocols became de facto standards for distributed and parallel computing and reliable fault detectors are assumed in almost all existing "fault tolerant" systems. Through the SMC feasibility study and observations of the Blockchain experiments, we gained the following insights:

a) Direct use of hop-by-hop protocols for any distributed and parallel computing service forms an application-level overlay network that can be easily broken when any of the

component is compromised. This is the root cause of the scalability dilemma.

b) Since every program-program communication protocol will form the application-level overlay network, a correct protocol cannot assume the reliability of networks, processors and storage, since they are all critical components in the overlay network.

c) Although this requirement seems overly difficult, its feasibility is not questionable. The Blockchain experiments demonstrated that a correctly designed end-to-end protocol can indeed offer a lossless computing platform in extreme scales. The design of the SMC framework offers an optimized version of the lossless decentralized computing protocol for generic compute and data intensive services without assuming decentralized authority.

d) Decentralized reliable computing protocols should build inductive applications that the reliability properties will be maintained regardless the scale of the application. The Blockchain protocol is such an example.

e) To offer performance gains for a decentralized computing service, the "communication" protocol must automatically enable all forms of parallelisms. These include SIMD, MIMD and pipeline clutters. Thus the protocol must also include automatic program synchronization (dataflow) and automatic data replication, consistency guarantee and non-stop failure recovery functions. The term of "active content addressable networking" was thus invented by my adviser.

f) The end-to-end design of SMC protocols paved a solid foundation for future end-to-end security protocol developments that will ultimately eliminate the "man-in-the-middle" cyber security threats. The elimination of the scalability dilemma for mission critical services makes future DDoS attacks exponentially more expensive.

The body of the SMC research has span more than three decades. This dissertation inves-

tigates the feasibility of the performance claims of SMC without compromising the fundamental reliability and security features of the framework.

7.2 Contributions

This dissertation has made the following contributions:

a) A minimal viable feasibility prototype for data intensive (DI) applications. It is called AknaStore. AnkaStore is implemented in Java with selected currently "most efficient" Open Source tools. This prototype features dynamic O(lg n) UVR formation , automated non-stop UVR maintenance complexities, integrated synchronous data replication/resynchronization gateways and automated process activation and termination. For this study, only *put*, *get*, *rm* and *ls* operators are implemented.

b) A minimal viable feasibility prototype for compute intensive (CI) applications. This prototype is called AnkaCom. It is also implemented in Java with another set of selected Open Source tools. This prototype also features dynamic UVR formation and automated non-stop UVR maintenance in $O(lgn)$ complexities, decentralized Tuple Space, all (put, get, read) Tuple Space operators are implemented. The runtime also automated process activation and termination.

c) Designed and carried out decentralized computational experiments for AnkaCom against the popular MPI framework under the identical bare metal HPC clutters (Owlsnest at Temple and NSF Chameleon bare metal clusters).

d) Designed and carried out decentralized computational experiments for AnkaStore against the Hadoop Distributed File System (HDFS) with results reported earlier. These experiments included simulated failures under extreme conditions (TCloud at Temple and NSF Chameleon bare metal clusters).

7.3 Observations

a) In small scale experiments for 12-24 nodes solving a dense matrix multiplication problem of size 6000-9000, the AnkaCom application demonstrated consistently better performance with automatic failure protection than the MPI counterpart without checkpoints. The AnkaCom experiments are optimized with the best processing granularity. The MPI programs use the fixed N/p partition factors. Diminishing returns are also captured for both AnkaCom and MPI tests. Due to the inductive nature of the decentralized computing protocol, these results suggest that the SMC framework is applicable to applications of any scale, limited only by (d).

b) In small scale experiments for 8 storage nodes, with block descriptor size 80 bytes (sample metadata object in sec. 5.4.4), reading and writing files of sizes from 1-50GB with replication factor $R = 3$, the decentralized AnkaStore application demonstrated consistently better performance than the HDFS with centralized controllers for the same file transfers (put and get) Figure 5.13. Adding storage nodes will allow larger number of files and file sizes. The maximal number of files is constrained by the collective memory sizes of all storage nodes. By the nature of the decentralized storage protocol, these results suggest that the SMC storage framework is suitable for transactional storage of any scale, limited only by (d).

c) Manual failure injection tests were conducted. An arbitrary node failure resulted in slight degradation of performance for the CI experiment (Figure 6.14). Manually introduced multiple storage node failures also resulted in slight performance degradation (Figure 5.18). There is no service downtime during these tests. For practical applications, starting from a given set of resources with known reliability estimates, the SMC powered infrastructure ensures no reliability degradation regardless the infrastructure expansions.

d) The UVR size and the economic law of diminishing returns are still the practical scaling limits. The $O(lgn)$ UVR maintenance complexity ensures the UVR scaling to very high number of nodes, depending upon the available networking and computing resources. The economic law of diminishing returns applies to application of different resource needs. Programming quality will impact the eventual performance delivery.

These results demonstrated the practical feasibility of the SMC framework for both compute intensive and data intensive applications.

Bibliography

[1] Justin Y. Shi. Statistic multiplexed computing - a neglected path to unlimited scalability. In *2nd Softwaren Engineering Assembly*, NCAR, Colorado, April 2013.

[2] Deborah K. Gracio Ian Gorton. *Data-Intensive Computing: Architectures, Algorithms, and Applications*. Cambridge University Press, 2012.

[3] Aakash Pradeep. P2phdfs: An implementation of statistic multiplexed computing architecture in hadoop file system. Master's thesis, Temple University, January, 2013.

[4] Gordon E. Moore. Cramming more components into integrated circuits. *Electronics*, 1965.

[5] John Markoff. Designing the next wave of computer chips. [online] https://www.nytimes.com/2014/01/10/science/designing-the-next-wave-of-computer-chips.html.

[6] Computing Community Consortium White Paper. 21st century computer architecture. *[Online] http://csl.stanford.edu/ christos/publications/2012.21stcenturyarchitecture.whitepaper.pdf*, 2012.

[7] Message Passing Interface Forum. A message-passing interface standard version 3.1, June 2015. [online] http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[8] *RPC: Remote Procedure Call.* https://en.wikipedia.org/wiki/Remote_procedure_call, 1988.

[9] *Java Remote Method Invocation.* http://en.wikipedia.org/wiki/Java_remote_method_invocation, 2012.

[10] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thera Angskun, Jack J. Dongarra, Jeffery M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Dieter Kranzlmller, Peter Kacsuk, and Jack J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 353–377. Springer Berlin/Heidelberg, 2004.

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[12] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[13] *Hyperledger Fabric*. https://wiki.hyperledger.org/projects/Fabric, 2016.

[14] Justin Shi and Yasin Celik. A study of single-sided statistic multiplexed computing using chameleon bare-metal cluster. *A Study of Single-Sided Statistic Multiplexed Computing Using Chameleon Bare-Metal Cluster*, November 2015.

[15] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[16] Paul Baran. Reliable digital communications systems using unreliable network repeater nodes. *RAND Corporation*, 22:303–312, 1960.

[17] J. Saltzer, D. Reed, and D.D. Clark. End-to-end arguments in system design. *Second International Conference on Distributed Computing Systems*, pages 509–512, April 1981.

[18] John M. Spinelli. Reliable data communication in faulty computer networks, June 1989. PhD thesis, MIT.

[19] Gilbert and Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM SIGACT News (2002)*, volume 33(2), page 59. ACM, 2002.

[20] Justin Y. Shi, Moussa Taifi, Abdallah Khreishah, and Jie Wu. Tuple switching network – when slower maybe better. *International Journal of Parallel and Distributed Computing*, 72, 2011.

[21] Justin Y. Shi. On the reproducibility and scalability of extreme scale applications. In *WSSSPE 2014*. xsede, 2014.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[23] Bill Joy and Peter Deutsch. Fallacies of distributed computing. *http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing*, 1994.

[24] Justin Y. Shi. Osi standards and the top fallacy of distributed computing. In *International Workshop on Sustainable High Performance Computing*. IEEE, 2016.

[25] Peter Deutsch. Eight fallacies of distributed computing,. [online] https://blogs.oracle.com/jag/resource/Fallacies.html.

[26] Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, August 2009.

[27] Alan Fekete, Nancy Lynch, Yishay Mansour, and John Spinelli. The impossibility of implementing reliable communication in the face of crashes. *J. ACM*, 40:1087–1107, November 1993.

[28] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems*. IEEE, 1999.

[29] Eric Brewer. Cap twelve years later: How the "rules" have changed. In *Computer*, volume 45, pages 23–29, Feb 2012.

[30] Eric Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. ACM, 2000.

[31] Kostantin V. Shavchko. Hdfs scalability: The limits to growth. In *http:c59951.r51.cf2.rackcdn.com5424-1908-shvachko.pdf*. Yahoo Inc., 2009.

[32] Justin Y. Shi. *Chapter 19: Fundamentals of cloud application architectures, Cloud computing: methodology, system, and applications*. CRC, Taylor & Francis Group, 2011.

[33] Yuan Shi. *"Stateless Parallel Processing Prototype: Synergy"*, 1996. [online] https://github.com/jys673/Synergy30.

[34] Justin Y. Shi. System for high-level virtual computer with heterogeneous operating systems. *U.S. Patent #5,381,534*, 1995.

[35] Justin Y. Shi. Multi-computer system and method. *U.S. Patent #5,517,656*, 1996.

[36] Justin Y. Shi and Suntian Song. Apparatus and method of optimizing database clustering with zero transaction loss. *U.S. Patent Application: #2008018969*, 2008.

[37] Justin Y. Shi. Fault tolerant self-optimizing multiprocessor system and method thereof. *U.S. Patent Application #20090019258*, 2007.

[38] Justin Y. Shi. System and method for fault tolerant scalable computing. *U.S. Patent #605725*, November 2016.

[39] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[40] Archley Kattt. Spark – lightning-fast cluster computing. Berkeley, CA, USA, 2011. University of Califorinia.

[41] Justin Y. Shi. A distributed programming model and its applications to computation intensive applications for heterogeneous environments. In *International Space Year Conference on Earth and Space Information Systems*, pages 10–13, Pasadena, CA., February 1992.

[42] Yasin Celik, Aakash Pradeep, and Justin Y. Shi. Ankacom: A development and experiment for extreme scale computing. In *CIT/IUCC/DASC/PICom*, pages 2010–2016. IEEE, 2015.

[43] Yasin Celik and Justin Y. Shi. Ankastore: A single-sided statistic multiplexed transaction distributed storage system. *IEEE Transactions on Dependable and Secure Computing (Under review)*, 2018.

[44] Moussa Taifi, Justin Y. Shi, and Yasin Celik. Jenergy: A fault tolerant stateless architecture for high performance computing. In *9th IEEE International Symposium on Service Oriented System Engineering*, March 2015.

[45] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[46] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.

[47] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[48] J. Y. Shi. Use hdlc principles for distributed applications. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 2002–2009, Oct 2015.

[49] J. Basney. *A Distributed Implementation of the C-Linda Programming Language*. PhD thesis, Oberlin College, June 1995.

[50] Justin Y. Shi. Symposium: Cap-plus for big data. In *Proceedings of the 2014 International Conference on Big Data Science and Computing*, BigDataScience '14, pages 19:1–19:5, New York, NY, USA, 2014. ACM.

[51] S. Bergamaschi. Big data analysis: Trends amp; challenges. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 303–304, July 2014.

[52] The exponential growth of data. `http://insidebigdata.com/2017/02/16/the-exponential-growth-of-data/`, 2017.

[53] Gil Press. A very short history of big data. `https://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/`, 2017.

[54] Justin Shi. Solving the scalability challenge from the ground up. *Special issue on Recent Advances in Engineering Systems organized by Advances in Science, Technology and Engineering Systems Journal (ASTESJ)*, 2017.

[55] R. S. Nikhil Arvind. Implicit parallel programmin in ph,. In *Introduction to Management Science (10th Ed),*. Morgan Kaufmann, 2011.

[56] David Gelernter and Nicholas Carriero. Coordination languages and their significance. In *Communications of ACM*. ACM, 1992.

[57] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM 2001*, San Diego, CA, USA, 2001. ACM.

[58] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[59] J. Y. Shi. In *A Distributed Programming Model and Its Applications to Computation Intensive Problems for Heterogeneous Environments*, Pasadena, CA, 1992.

[60] Justin Y. Shi, Moussa Taifi, Abdallah Khreishah, Aakash Predeep, and Vivek Anthony. Program scalability analysis for hpc cloud: Applying amdahl's law to nas benchmarks. In *International Workshop on Sustainable HPC Cloud/Supercomputing Conference 2012*. IEEE, 2012. Salt Lake City, Utah.

[61] Brachitoschrone curve. https://en.wikipedia.org/wiki/Brachistochrone_curve.

[62] Apache hadoop. `http://hadoop.apache.org`.

[63] Judith Hippold and Gudula Runger. Task pool teams for implementing irregular algorithms on clusters of smps. Nice, France, 2003. IEEE Press.

[64] Chameleon. A configurable experimental environment for large-scale cloud research. `https://www.chameleoncloud.org/`, 2017.

[65] Temple owlnstest. `http://www.hpc.temple.edu/owlsnest/OwlsnestUserGuide.html`.

[66] Apache.org. Apache zookeeper, 2014.

[67] Wikipedia. Osi model, 2013. [online] http://en.wikipedia.org/wiki/OSI_model.

[68] Justin Y. Shi and SuntainSong. On the assumptions of cap theorem and the big data challenges. In *Technical Report, Parallel Computers Technology Inc.*, 2013. unpublished manuscript.

[69] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 173–182. ACM Press, 1996.

[70] David Bindel John Kubiatowicz, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. Princeton University, NJ., 2000.

[71] Bob jenkins. Jenkinson hashing. http://en.wikipedia.org/wiki/ Jenkins_hash_function.

[72] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, July 2000.

[73] Hbase: Bigtable-like structured storage for hadoop hdfs. `https://hbase.apache.org`.

[74] Top500. Top500. `http://www.top500.org`.

[75] Garth A. Gibson Bianca Schroeder. A large-scale study of failures in high-performance computing systems. In *the International Conference on Dependable Systems and Networks*, June, 2006.

[76] I. Riakiotakis, F. M. Ciorba, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos. Towards the optimal synchronization granularity for dynamic scheduling of pipelined computations on heterogeneous computing systems. *Concurr. Comput. : Pract. Exper.*, 24(18):2302–2327, December 2012.

[77] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, August 1992.

APPENDIX HEADING

SMC FUTURE

SMC as a Solution

Although TCP has been considered as "lossless best effort" communication protocol, proved in [18], given unbounded resources, it does not imply that the communication between programs is reliable. It is now accepted that "the network is reliable" is a fallacy [25] when building distributed systems.

Statistic Multiplexed Computing principle is a new programming paradigm which forces developer to think in a different way. Network, storage, and computing components failures are assumed unreliable in SMC architecture. Therefore, SMC as a complete package can fix many distributed and parallel application reliability and scalability issues. The most common distributed computing problems and corresponding SMC solutions:

a) **Single point of failure:** Several distributed systems suffer from this problem. Any centralized service in distributed systems is undesirable, which can provoke entire system failure. A SPOF is also bottleneck for system performance and is obstacle scala-

bility. In order to avoid SPOF problem, a system must decentralize all of its services. SMC principle does not introduce SPOF by it nature, and any distributed and parallel system developed using SMC principle are safe from SPOFs.

b) **Replication / Consistency:** Distributed systems usually exploit redundancy by duplicating system components, such as data and program, to increase reliability and performance. However, redundancy is prone to data inconsistencies if replication functions are not carefully designed and developed. Mission critical applications require strong concurrency control among write and read operations. 2-phase commit protocol is the most commonly mechanism used to provide guarantees consistent state among replica. However, it is a performance bottleneck since it uses locking, and rollback in case if one replica fails to commit a given operation. Gateway in SMC principle provides synchronous data replication that ensures strong consistent state among replicas without using any locking mechanism. It does not penalize a user because of node failures. Gateway function itself is also duplicated and stateless, therefore, its failure does not affect data consistency.

c) **Fault tolerance and recovery:** Fault tolerance in distributed systems is a desirable property, which enables a system to properly maintain its functionality when some of its components fails temporarily or permanently. Safe recovery after a failure is also crucial, particularly to stay at a high availability state. SMC architecture is intrinsically fault tolerant by its design since every component is necessarily statistically multiplexed similar to packet switching network. Any component failure in system designed with SMC principle does not compromise a total service shot down because of SMC replication schema. Gateway in SMC architecture synchronously reconcile data sets after a failure. A short service shot down, usually in the order of seconds, may be required to finalize the reconciliation process. This service shot down is local that does not affect the entire system. For instance, a single node failure in a system,

whose the replication factor is 3, built with SMC principle will only involve 2 successor and 2 predecessor nodes.

d) **Timeout retransmission discipline:** Although communication protocols at packet level, such as TCP, have utilized timeout-retransmission discipline to ensure reliable packet delivery, it has remained a challenging research problem at distributed application level for both academia and industry. A distributed application can quickly go into unknown state if a node sends an application level packet and the packet is not acknowledged form other end. The sender cannot simply resend the packet unless the proper mechanisms are not in place. SMC principle's timeout retransmission discipline with duplicate elimination mechanism avoids unknown states.

e) **Order of events / Serialization:** Mission critical applications require strong concurrency control among write and read operations. A monetary transactional system, such as banking and stock trading, does not tolerate wrong order among events, otherwise the result could be catastrophic. According to SMC principle, gateway processes request based on their arrival time unless there is a conflict, which is solved by conflict resolution mechanism by ordering requests based on their arrival time. Thus, SMC does not incur high cost for parallel replication.

f) **Byzantine fault tolerance:** Byzantine General's Problem refers an agreement problem among commander when making a particular decision. The root cause of the problem is whether to trust messages received from other commanders since the action is taken based on the majority vote of a group of commanders. SMC principle does not trust unresponsive or crashed nodes, simply dismisses them from UVR. The gateway component continues with successful ACKs if some nodes are unresponsive when replicating an operation. The failed nodes have to reconcile their state by synchronizing with other nodes.

g) **Deadlock:** The distributed systems using locking mechanism are prone to deadlocks

where a resource can be used by only one component and the same component needs multiple resources to commit an operation. In such scenario with two separate components, say A and B, where A may allocate a resource X and waiting for another resource Y to be available, which is held by component B and waiting for the resource X to become available. Deadlock will occur in such a scenario unless a deadlock prevention mechanism is implemented. Deadlocks are not presented in SMC architecture since there is no locking mechanism.

h) **Race Condition:** If the order of events is not carefully implemented in distributed systems or multithreaded programming, race condition may arise, where the final result may be wrong. SMC principle does not allow concurrent updates on same data object by employing conflict resolution mechanism, which puts update operations in order according to their arrival times.

i) **Time complexity:** As the distributed systems expand in size in order to handle extreme large problems, the cost per operation may linearly or even exponentially increase if the system architecture has bottlenecks, and not designed carefully. A distributed service, especially for business practices, has to offer acceptable service level agreement to provide satisfactory user experience. For instance, several systems using distributed hash table, such as Chord, the cost of lookup service is $O(logn)$, where n is the number of nodes in the ring. This cost is generally acceptable if the size of the system does not grow "too big." The time complexity of SMC principle operations is $O(logn)$ in worst case. It can be reduced to $O(1)$ in some scenarios [43].

PUBLICATIONS

Journal Articles

- Y. Celik, J. Y. Shi: AnkaStore: A Single-Sided Statistic Multiplexed Transactional Distributed Storage System. IEEE Transactions on Dependable and Secure Computing. (Under review) 2018
- J. Y. Shi, Y. Celik, A Study of Single-Sided Statistic Multiplexed Computing Using Chameleon Bare-Metal Cluster. Special issue on Recent Advances in Engineering Systems organized by Advances in Science, Technology and Engineering Systems Journal (ASTESJ) 2017.

Conference Proceedings

- J. Y. Shi, T. Evans, K. A. Kosman, and Y. Celik. A study of single-sided hpc communication paradigm using chameleon bare-metal cluster. In 2017 International Conference on Computing, Networking and Communications (ICNC), Jan 2017.
- Justin Y. Shi and Yasin Celik. Single-sided statistic multiplexed high performance computing. In Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering, SE-HPCCSE'16, 2016. IEEE Press.
- M. Taifi, J. Y. Shi, Y. Celik , JENERGY: A Fault Tolerant Stateless Architecture for High Performance Computing, in Proc. of the 9th IEEE International Symposium on Service Oriented System Engineering (SOSE 15), March 2015
- Y. Celik, A. Pradeep, J. Y. Shi, AnkaCom: A Development and Experiment for Extreme Scale Computing, Submitted to International Conference for High Performance Computing, Networking, Storage and Analysis, 2015.

Extended abstracts, Posters and Technical reports

- Y. Celik, "AnkaCom: Statistic Multiplexed Data Intensive Computing Prototype", in Proceedings of 3rd Annual Computational Research on Owlsnest Symposium (CROO 2014), 2014.
- Y. Celik and J. Y. Shi, "P2PGW :A Fully Decentralized Framework Delivers Consistency, Availability, Partition Tolerance and Scalability all-in-one", Poster, Future of Computing Symposium, Temple University, 2014.