Scalable Coordination of Distributed In-memory Transactions

Ryan Emerson
School of Computing Science
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

Computing Science 2015

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor Dr Paul Ezhilchelvan, who not only gave me the opportunity to undertake a PhD, but also provided invaluable support, guidance and insight throughout the last four years. Additionally, I would like to thank Red Hat for sponsoring my research. Finally, I wish to thank my friends and family, without your support this thesis would not have been possible.

Abstract

Coordinating transactions involves ensuring serializability in the presence of concurrent data accesses. Accomplishing it in an scalable manner for distributed in-memory transactions is the aim of this thesis work. To this end, the work makes three contributions. It first experimentally demonstrates that transaction latency and throughput scale considerably well when an atomic multicast service is offered to transaction nodes by a crash-tolerant ensemble of dedicated nodes and that using such a service is the most scalable approach compared to practices advocated in the literature. Secondly, we design, implement and evaluate a crash-tolerant and non-blocking atomic broadcast protocol, called ABcast, which is then used as the foundation for building the aforementioned multicast service.

ABcast is a hybrid protocol, which consists of a pair of primary and backup protocols executing in parallel. The primary protocol is a deterministic atomic broadcast protocol that provides high performance when node crashes are absent, but blocks in their presence until a group membership service detects such failures. The backup protocol, Aramis, is a probabilistic protocol that does not block in the event of node crashes and allows message delivery to continue post-crash until the primary protocol is able to resume. Aramis design avoids blocking by assuming that message delays remain within a known bound with a high probability that can be estimated in advance, provided that recent delay estimates are used to (i) continually adjust that bound and (ii) regulate flow control. Aramis delivery of broadcasts preserve total order with a probability that can be tuned to be close to 1. Comprehensive evaluations show that this probability can be 99.99% or more.

Finally, we assess the effect of low-probability order violations on implementing various isolation levels commonly considered in transaction systems. These three contributions together advance the state-of-art in two major ways: (i) identifying a service based approach to transactional scalability and (ii) establishing a practical alternative to the complex PAXOS-

style approach to building such a service, by using novel but simple protocols and open-source software frameworks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The emergence and proliferation of mainstream cloud computing has facilitated the creation of a large number of Internet-scale web services and applications. Such services serve millions of users across the globe simultaneously and are required to cater for increasingly large numbers of read and write operations on data. Furthermore, these data operations need to occur in the range of milliseconds in order to provide the low-latency experience expected by the user's of these services. Cloud computing is ideal for such workloads, as it enables the web service to scale horizontally by dynamically acquiring resources as the rate or size of requests increases.

Traditionally, applications would utilise a Relational Database Management System (RDBMS) for storing and retrieving data. However, as Internet scale services such as Facebook, Twitter and Google continued to receive increasing numbers of user requests, it became clear that RDBMS systems were unable to provide the low-latency responses required by these services when operating at such scale [56]. Therefore it became necessary to seek alternatives to RDBMS which can maintain small response times when operating under such conditions.

The traditional approach to scaling RDBMS, was to scale *vertically*, by utilising increasingly powerful and expensive servers to handle application requests. Such an approach is not truly scalable as the maximum levels of performance will always be limited by the capabilities of the latest technology, the cost of hardware and the associated running costs. Alternatively, it is possible to *horizontally* scale RDBMS solutions by partitioning data across several nodes in order to increase the number of machines that can handle requests. However, as RDBMS systems depend on a rigid data-schema to structure data, horizontal partitioning is difficult in practice and often requires input from system administrators to maximise its effectiveness [32].

The emergence of cloud computing as a cost effective model, combined with RDBMS's inability to scale elastically [1], has led to the emergence of NoSQL databases as an alternative storage solution. These databases typically offer simpler data models and more relaxed consistency criteria than traditional RDBMS systems, in order to: (i) avoid the need for predefined data schemas that hinder elasticity and (ii) reduce the overhead of maintaining data replicas across multiple nodes [12]. Consequently, NoSQL stores are highly elastic and are also suited to improving availability through replication, as we explain below.

NoSQL databases can effectively utilise horizontal scaling in order to service an increasing number of application requests, whilst also providing increased fault-tolerance, via data replicas that are distributed throughout the cluster. Utilising multiple data replicas allows for increased levels of throughput as application read requests can be serviced by multiple nodes simultaneously. However, a consequence of utilising distributed replicas is that each write operation requires several Remote Procedure Calls (RPC) in order to maintain a consistent state between all of the data replicas; with consensus being required between data replicas for each write operation, as all replica hosts must perform write operations on a given value in the same order. The cost of obtaining consensus between replicas, coupled with the additional latency cost associated with RPCs, is the primary reason for many NoSQL databases choosing to provide weaker consistency guarantees than the traditional 1-copy serialisability provided by ACID transactions in RDBMS.

Infinispan [35, 54] is an open source NoSQL database developed by Red Hat, Inc [62] that utilises the RAM of its host machines to store data, which is exposed to applications using a key/value model. When Infinispan is horizontally scaled over multiple nodes it provides two options for distributing key/value pairs: *full* and *partial* replication. Full replication stores a replica of every key/value on each Infinispan node in the cluster, and consequently is not scalable beyond a small number of nodes. Conversely, partial replication only stores each key/value pair on a small subset of nodes contained in the cluster and as a result is very scalable when the specified *replication factor* is small; where the *replication factor* is the number of distinct nodes at which each key/value pair is replicated [66].

Unlike many NoSQL databases, Infinispan provides support for ACID transactions, how-

[1] Where elasticity is defined as the ability to add or remove nodes from a system without requiring manual intervention or a large performance overhead.

ever it utilises reduced levels of *isolation* compared with the traditional 1-copy serialisability in order to improve performance and scalability. Infinispan's distributed transactions are coordinated between nodes in a *peer-to-peer* (P2P) manner, using either the classical 2-Phase commit protocol or a lock free total order commit protocol that relies on an atomic multicast protocol. Existing research [64] shows that utilising the latter improves performance compared to the 2-Phase commit protocol with respect to transaction abort rates, latency and throughput. This increase in performance is due to the latter's reliance on a total order of messages, as opposed to lock acquisition, which ensures that all data replicas perform write operations in the same order, thereby avoiding deadlocks as no locks are used.

The total order commit protocol provided by Infinispan consistently outperforms 2-Phase commit. However, our performance evaluation (§ 6) shows that the performance of the underlying atomic multicast protocol currently used for coordinating these transactions does not scale as the number of nodes involved in a transaction becomes greater than 3; with the average transaction latency and throughput, increasing and decreasing respectively. The atomic multicast protocol's inability to scale is acceptable when both the replication factor and the number of write operations in a transaction is low, however when the replication factor is greater than 2 or the number of write operations in a transaction is greater than 1, performance will deteriorate. For example, if a transaction consists of 3 write operations that are to be performed on distinct key/value pairs, and Infinisipan utilises the default replication factor of 2, then the total number of nodes involved in that transaction can be anywhere between 2 and 6 nodes depending on how the keys are distributed across the Infinispan cluster. Hence, it is probable that the total number of destination nodes involved in the atomic multicast that is required to coordinate the transaction will be greater than 3 and therefore transaction performance will be poor.

The scalability problem stated above is not unique to Infinispan's atomic multicast protocol, rather the same limitations apply to any distributed system that utilises P2P protocols to coordinate transactions containing partially replicated data. With these observations in mind, the problem statement for this thesis is formulated below.

1.1 Problem Statement

To design alternative protocols for totally ordered multicasts between subsets of cluster nodes, in order to improve throughput and latency when coordinating transactions in partially replicated environments.

1.2 Our Approach

In this thesis we advocate an alternative approach to coordinating distributed transactions, which unlike P2P protocols does not require ordering consensus to be reached between peers. Our approach consists of building two key components:

i An external ordering service that enables atomic multicasts between Infinispan nodes (SCast).

ii A non-blocking atomic broadcast protocol for state machine replication within the ordering service (ABcast).

The external ordering service is an alternative to the P2P approach currently utilised by Infinispan. This service consists of a dedicated set of nodes acting as a replicated state machine which provides a total order value for any multicasts required by Infinispan nodes. The rationale behind this approach is that the number of nodes required to reach a consensus on a message's total order is limited to the number of nodes in the ordering service. Therefore, as the number of nodes participating in a transaction increases, the time required to reach a consensus remains constant, hence the problems associated with P2P protocols in partially replicated environments are circumvented.

Prior to this research, the throughput capabilities of such a service were limited by the protocols available for replicating state between the nodes providing the ordering service. Existing coordination services, such as Zookeeper [34] and Chubby [11], utilise Quorum based protocols for state machine replication. Such protocols are advantageous as they do not block in the event of node crashes, however they are dependent on a 'leader' node for coordinating replication which becomes a performance bottleneck as the number of required replications increases. Such protocols are less attractive for our ordering service, as the service requires a node's state

to be replicated each time an ordering request is received and requests are expected to occur frequently.

An alternative approach to the Quorum based protocols, are group membership dependent atomic broadcast protocols. Such protocols do not utilise a leader node and provide the best possible throughput and latency when no node crashes occur. However, when node crashes do occur, message delivery to the application is blocked until the group membership service informs the participating nodes that a failure has occurred. This blocking makes these protocols equally less attractive for use within an ordering service as a single service node crash will prevent the ordering service from providing a total order on Infinispan's multicast messages, which in turn prevents any transaction in the Infinispan cluster progressing.

Motivated by the limitations of these existing protocols, the second aspect of our research is the development of a hybrid atomic broadcast protocol that can be utilised for fault-tolerant state machine replication. This protocol combines a deterministic atomic broadcast, which provides low-latency message delivery in the absence of node failures, and a probabilistic protocol for non-blocking message delivery in their presence; the latter guarantees atomicity with a probability close to 1.

As this hybrid possible allows for probabilistic message delivery, it is possible for messages to be miss-ordered, therefore in the context of an ordering service it is possible for clients relying on this service to also suffer order violations. Therefore, a key aspect of our work is exploring in detail various strategies that can be adopted by an ordering service to mitigate such ordering violations as well as the additional logic required by client nodes consuming a probabilistic service.

Throughout this thesis we have utilised Infinispan as a basis for designing and implementing our solutions. Infinispan was utilised because it is entirely open source and already provides support for coordinating ACID transactions via atomic multicasts. Furthermore, given that this research was funded in its entirety by Red Hat, Inc who are responsible for developing Infinispan, our choice was natural.

1.3 Thesis Contribution

The research presented in this thesis makes several key contributions:

i A new system model, Atomic Multicast as a Service (AmaaS), for coordinating distributed transactions in partially replicated environments and a new fault-tolerant atomic multicast protocol that has been designed specifically for AmaaS.

ii A hybrid atomic broadcast protocol called ABcast, which is designed for providing state machine replication within the AmaaS model. ABcast provides low-latency message delivery in the absence of node failures and non-blocking message delivery in their presence, by utilising both a deterministic and probabilistic atomic broadcast protocol to deliver messages up the network stack.

iii An extensive performance evaluation of both the AmaaS model and the ABcast protocol.

1.4 Thesis Structure

Chapter 2 - Background

Presents the key prerequisite information required to understand the problem domain.

Chapter 3 - AmaaS

Introduces a new system model for coordinating partially replicated transactions, that we call Atomic Multicast as a Service - AmaaS. This new model is inherently different from the P2P approach previously utilised by the Infinispan database, therefore we formalise a new fault-tolerant atomic multicast protocol, SCast, for coordinating transactions that utilises AmaaS.

Chapter 4 - ABcast

Presents the rationale, design assumptions and important implementation details for the hybrid atomic broadcast protocol ABcast. As well as detailing AFC, a bespoke flow-control protocol designed for ABcast.

Chapter 5 - Probabilistic SCast

Explores the consequences of utilising the ABcast protocol for state machine replication between service nodes in the AmaaS model. More specifically, it focuses on the potential repercussions of message miss-orderings at the service level and how they can impact client nodes that depend on the ordering service.

Chapter 6 - Performance Evaluation

Provides a thorough performance evaluation of the AmaaS model compared to the existing P2P approach, as well as investigating the effect of utilising different atomic broadcast protocols within the AmaaS service. Furthermore, we evaluate the performance of the ABcast protocol when node failures occur, in order to ascertain the effectiveness of the protocols non-blocking message delivery.

Chapter 7 - Conclusions

Presents a summary of the findings presented throughout this document and speculates on potential future research made possible by our findings.

Chapter 2

Background

2.1 Network Communication Paradigms

Solutions to distributed problems are commonly associated with one of two paradigms: Synchronous and Asynchronous communication. Each paradigm has defining characteristics that can be both beneficial and limiting depending on a system's requirements. This section defines both of the common paradigms, before introducing a third paradigm that is central to our research.

2.1.1 Synchronous

Synchronous communication refers to a communication model, in which, a fixed upper bound can be placed on the communication delay experienced when sending data packets between any two nodes in the network. If a data packet exceeds this upper bound then the transmission is deemed to have failed due to a timing failure, requiring the data packet to be retransmitted.

In order to successfully calculate the fixed upper bound on communication delays a synchronous network needs to establish an upper bound on the number of faulty nodes present in the network, the maximum load of the network and the transmission rate of data packets[15]. These requirements make the synchronous paradigm unsuitable for use in middleware and distributed database systems, as such systems are typically executed on commodity hardware. Therefore, none of our contributions utilise the synchronous paradigm.

2.1.2 Asynchronous

The Asynchronous communication model does not define a known upper bound on communication or processing delays, instead these delays are considered finite and arbitrary, resulting in the performance bounds required in the synchronous model becoming redundant[15]. Placing no bounds on network load or the number of faulty nodes makes the asynchronous model much more flexible than the synchronous approach, enabling the asynchronous model to be implemented over various network topologies that utilise commodity hardware.

A limitation of the asynchronous model is the inability to distinguish between a slow or a crashed node, due to the lack of an established upper bound on communication delays. Consequently many applications using the asynchronous model must rely on configurable timeout parameters, which can result in *false suspicions*; where a slow node is incorrectly suspected of having crashed. Conversely, it is also possible for the time outs to be too large, resulting in the system waiting longer than necessary to detect a crashed node.

The inability to distinguish between slow and crashed nodes leads to the FLP impossibility discovered by Fischer, Lynch and Patterson[27]. The FLP impossibility formally proves that it is impossible for distributed consensus to be achieved in a deterministic manner in an asynchronous environment when a single crashed node is present.

2.1.3 Probabilistically Synchronous

Recent papers have called for an alternative to the asynchronous model to be utilised when designing distributed systems. Aguilera and Walkfish[1] argue that the asynchronous model is inherently unsafe. They believe that removing assumptions about synchrony at the lower layers of a system can sacrifice liveness throughout the system. Furthermore the inability to distinguish between a crash and a slow process can result in users of a system having to guess on the appropriate action to take in order to remedy the situation, potentially violating safety.

Ezhilchelvan and Shrivastava[22] introduce a new communication model, Probabilistic Synchronous Model (PSM), which aims to overcome the previously stated issues with asynchrony. PSM is based upon the assumption that, in datacentres and cluster-based environments, there is a correlation between the past and near future "performance" of the network; where performance is the probability distribution of delays. The recent past performance of the network

can then be used as an input parameter for distributed protocols, which utilise these values to calculate probabilistic guarantees. Monitoring the recent past performance of the network also enables protocols to utilise time outs that are considered accurate to a certain probability, R. Therefore enabling processes to be distinguished as either slow or crashed with the probability of *false suspicions* being $1 - R$.

2.2 Atomic Broadcast and Multicast Protocols

Atomic broadcast and atomic multicast protocols, *abcast* and *amcast* for short, are *one-to-many* network protocols that provide specific guarantees on message delivery to ensure that messages are delivered reliably and in the same order at all destinations; where message delivery is defined as the passing of a message up the network stack to a higher level protocol or application.

Below we consider the guarantees required by broadcast and multicast protocols, in order for them to be considered atomic. For the purpose of brevity, we refer only to *amcast* here, however these guarantees also apply to *abcast*s. The following guarantees must be maintained to ensure that a multicast is atomic, with regard to the delivery order and the set of destinations that deliver the message.

G1 - *Validity*: If the source of m_i does not crash until it *amcast*s m_i, then all operative destinations of m_i deliver m_i.

G2 - *Uniform Agreement*: If the source of m_i crashes while *amcast*ing m_i, and if any destination delivers m_i, then all operative destinations of m_i must deliver m_i.

G3 - *Uniform Integrity*: If m_i has already been delivered by a destination d, then d cannot deliver m_i again.

G4 - *Uniform Total Order*: If two *amcast*s, m_i and m_j, have common destinations, then all such destinations that deliver both m_i and m_j, must deliver them in an identical order (i.e. either $< m_i, m_j >$ or $< m_j, m_i >$)

As previously stated, message delivery is a one-time, irreversible operation that occurs when a message is passed up from the *amcast* protocol to the next layer in the network stack. Once a

message has been delivered, it cannot be undelivered, therefore any violations of message guarantees cannot be undone at the *amcast* level. Therefore meeting G1-G4 presents two challenges, C1 and C2, that need to be met by all *amcast* protocols.

Consider m is to be *amcast* to a set of destinations $m.dst$, where $m.dst$ contains the source of an *amcast* message, as the receiving of the message incurs no additional network cost and enables the source application to receive its own message. C1 and C2 are stated below:

C1 - If an operative $d \in m.dst$ receives m, then every operative $d' \in m.dst$ must be able to receive m so that G1 and G2 are not violated.

C2 - Every d that receives m must determine a *safe* moment to deliver m so that G3 and G4 are not violated.

Meeting both C1 and C2 is not a trivial task, as such there is a large amount of literature[16] on *amcast* protocols spanning several decades. From the literature, it is clear that there exists two distinct approaches to solving the challenges of *abcast* and *amcast*; *Group Membership* and *Quorum based* based protocols. This section will describe each of these approaches and explore notable examples of each approach.

2.2.1 Atomic Broadcast vs Atomic Multicast

So far we have considered *abcast* and *amcast* protocols to be *one-to-many* network protocols, that must satisfy guarantees G1-G4 in order to be considered atomic. However, there are key differences between broadcast and multicast protocols. Below we provide a definition for both broadcast and multicast protocols and explore the strengths and limitations of each type.

2.2.2 Broadcast

In the literature[16] broadcast protocols, and hence *abcast*, are defined as a *one-to-many* network protocols that only allow messages to be sent between a single destination set, with all destinations in the set receiving each broadcast. For example if the total number of destinations is equal to 5, then $|m.dst| = 5$ is always true.

Restricting *one-to-many* communication to a single destination set can provide performance benefits over protocols that allow multiple destination sets, when the size of the destination set

is small ($e.g. |m.dst| < 5$). This is because such protocols can employ various optimisations, such as as piggybacking meta information on messages, as they know that $m.dst$ remain the same for all broadcasts when no node failures occur. Furthermore, broadcast protocols do not have the overhead of handling more complex message routing problems such as the overlapping subset problem described in 2.2.3.

Due to $m.dst$ always being the same, the scalability of single destination set protocols is underwhelming, with performance degrading dramatically as the number of destinations increase.

2.2.3 Multicast

In contrast to broadcast protocols, multicast protocols allow different messages to be sent to multiple destination sets. Such protocols fall into two categories, those that only allow *disjoint* destination sets and those that allow *overlapping* destination sets.

The creation of disjoint *amcast* protocols is trivial, as the majority of *abcast* protocols can be converted into disjoint protocols with only a few minor adjustments [16]. Disjoint protocols are not applicable to Infinispan, as the ability to only multicast to disjoint sets of destinations does not provide a solution to either the *partial* or *full* replication problem, as described in 2.5.

Creating *amcast* protocols that support overlapping destination sets is a challenging task, as any destination contained in two overlapping subsets has to satisfy G4 for all messages involved in both destination sets. Say node a multicasts m_i to $m.dst = \{a,b,c\}$ and node d multicasts m_j to $m.dst = \{b,c,d\}$ the challenge is ensuring that the common destinations $\{b,c\}$ deliver both messages in the same order; either m_i before m_j or vice versa. Furthermore, solving C2 becomes more difficult as we need to ensure that $\{b,c\}$ do not miss m_i or m_j, in a way that is not overly-restrictive on performance.

It is worth noting that, by definition, a *amcast* protocol can always be converted to a *abcast* protocol as the multicast protocol can simply multicast all messages with the same destination set.

2.2.4 Group Membership based approaches

This section details the Group Membership (GM) approach to solving the problem of creating *abcast* and *amcast* protocols, before providing an example of a GM based *amcast* protocol. For

the remainder of this section, we consider *amcast* protocols, however the use of GM is also applicable to *abcast*.

The GM approach to *amcast* protocols refers to a group of protocols that rely on a higher level service/protocol to maintain a current *view* of nodes that are correct (not crashed). The *amcast* protocol leaves all crash detection to the GM protocol, and assumes that the latest *view* v_i issued by the GM protocol is representative of the network's current state. When the GM service detects that a node has crashed, it publishes a new view v_j, which the *amcast* protocol will utilise until a subsequent view is published. The *amcast* protocol is responsible for ensuring that guarantees G1-G4 are maintained by taking appropriate action upon receiving a new view from the GM service.

GM dependent protocols always operate on the assumption that the current view v_i provided by the GM protocol is accurate. A consequence of this is that when v_i no longer represents the actual state of the network, $|v_i| \neq |ActiveNodes|$, the *amcast* protocol will block until v_j is published. This blocking will result in a loss of availability for any *amcast* messages required by higher level protocols/applications, however it is necessary to ensure that G1-G4 are maintained. Upon receiving v_j, the *amcast* protocol will safely remove any messages that have been received from the crashed node, but have not yet been delivered by any destination (G1). A *virtually-synchronous*[8] closure is typically used to ensure that all *amcast* messages sent by the crashed node, that have been delivered by at least one correct destination, are delivered by all of the remaining destinations (G2).

Newtop

Newtop[23] is a GM based *amcast* protocol developed by Ezhilchelvan *et al.* that supports multicasting to overlapping destination sets. It utilises acknowledgements and logical clocks, to ensure that C1 and C2 are met, respectively.

To ensure that C1 is met, the delivery of a message m, sent by $m.o$, is delayed until each $d' \in m.dst - \{m.o\}$ has acknowledged m by sending $ack_{d'}(m)$ to every $d \in m.dst$ and each $d \in m.dst$ has received $ack_{d'}(m) \forall (m.dst \setminus \{m.o, d\})$. This ensures that it is impossible to violate C1, as every d has confirmation that m has been received by all members of $m.dst$. C2 is addressed by each m and each $ack_{d'}(m)$ being *tentatively* timestamped with a value that is one

more than the timestamp ever seen or used by the respective source [46]. Once m and the $ack_{d'}(m)$ of every $d' \in m.dst - \{m.o\}$ are received, $d \in m.dst$ *finalizes* a timestamp ($m.ts$) for m as the largest of all these tentative timestamps.

Figure 2.1 shows how timestamps and logical clocks are used to finalise a timestamp for a given m; where each node's logical clock is represented as LC and each message contains a reference to m, the address of the source node sending the message and the associated timestamp.

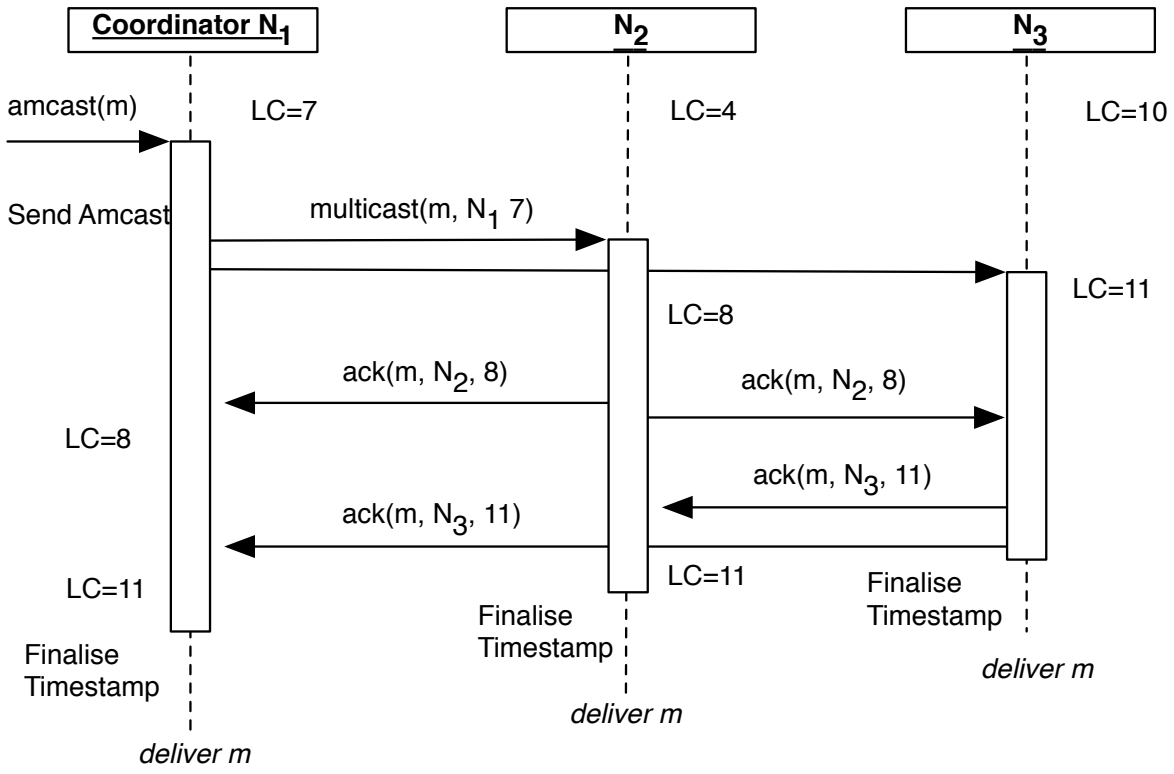

Fig. 2.1: Newtop Protocol Sequence Diagram

When d delivers every received m as per (finalized) $m.ts$, all guarantees are met. Proofs are in [8, 23, 46] and the intuition is given below.

Since $m.ts$ is finalized only after having received a tentative timestamp from every node in $m.dst$, for any $d \in m.dst$, $m.ts$ cannot be smaller than any of the tentative timestamps proposed for m, when d finalizes $m.ts$, it must have received any m' whose $m'.ts$ *could* be finalized as $m'.ts < m.ts$. So, if d finalizes $m.ts$ before finalizing $m'.ts$, it will wait for $m'.ts$ to be finalized before delivering m.

Say, $d' \in m'.dst - \{m.o\}$ is crashed; When $d \in m'.dst$ does not receive $ack_{d'}(m')$, it is blocked from finalizing $m'.ts$ until the GM protocol confirms that d' is crashed and $ack_{d'}(m')$

does not exist (through *virtually synchronous* closure). Say, d has proposed $ack_d(m').ts$ and also it finalizes some $m.ts$ while d' remains crashed (Note: d' is not in $m.dst$). If $m.ts > ack_d(m').ts$, d is also blocked from delivering m until $m'.ts$ is finalized, because d knows that $m'.ts$ can be finalized as $m'.ts < m.ts$.

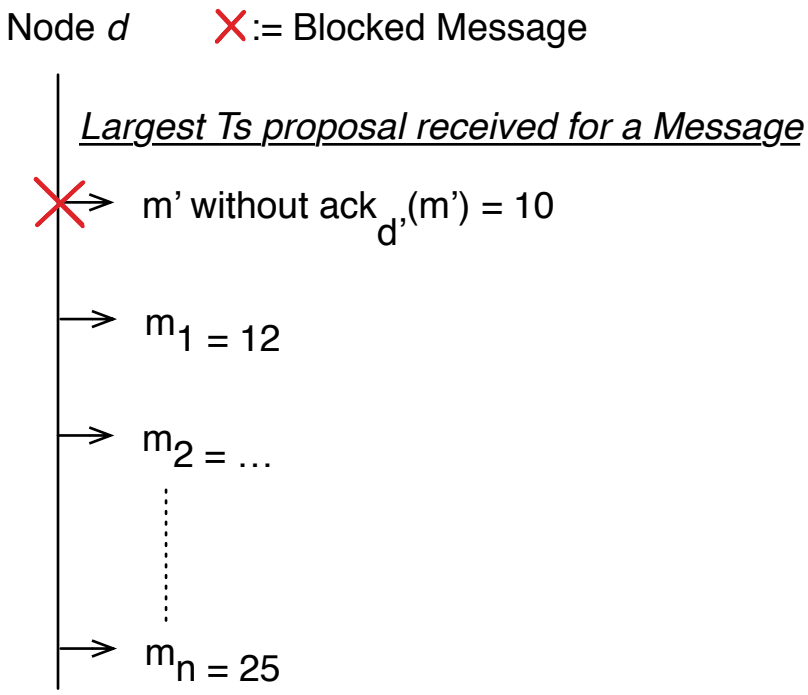

Fig. 2.2: Newtop Timestamp Blocking Diagram

Figure 2.2, shows the stages detailed above. Messages m_1 cannot be delivered until $m'.ts$ has been finalised, *i.e.* the crashed node d' has been detected by the GM service and node d finalises $m'.ts$ as it knows that $ack_{d'}(m')$ will never be received. The largest proposed ts for m_i can only increase with time, therefore the longer the GM service takes to detect d' has crashed, the larger n in the figure becomes. Thus, when *amcast*s are initiated by an application, a large delay between a node crashing and the GM service detecting it, results in an increased probability of many nodes receiving *amcast*s that have the crashed node in their destination set. As the number of nodes participating in *amcast*s involving the crashed node increases, the size of n also increases, resulting in more finalized *amcast*s that cannot be delivered to the application. Ultimately leading to a loss in the system's liveness, as no progress can be made by any node waiting for an acknowledgement from a crashed node, until the GM service detects the node crash.

Although the Newtop protocol blocks in the presence of node crashes, in their absence it provides the smallest achievable latency available to *amcast* protocols. This is because Newtop finalises $m.ts$ within $2 \times x_{mx}$, where x_{mx} is the maximum message latency between the nodes in $m.dst$. Therefore allowing the protocol to deliver messages within $2 \times x_{mx}$ when blocking does not occur.

Finally, if the Newtop protocol is utilised in a single destination environment, hence as a *abcast* protocol, it is possible to finalise $m.ts$ and deliver messages with a single broadcast from $m.o$. This is achieved by all d piggybacking $ack_d(m)$ on subsequent messages, therefore reducing the load on the network and increasing throughput.

2.2.5 Quorum Based approach

This section details the canonical example of a Quorum based *abcast* protocol. It is possible to solve *amcast* utilising a quorum based protocol, however it is a non-trivial task which is beyond the scope of this thesis, therefore we only consider *abcast* protocols.

Quorum based *abcast* protocol are *abcast* solutions that utilise a *master* node to coordinate *slave* nodes. The master node is responsible for sending all messages, and proposing $m.ts$, which is confirmed when it receives $ack(m)$ from a *majority* of nodes; hence the name quorum-based. Thus each *abcast* requires at least 3 communication steps, resulting in a minimum latency of $3 \times x_{mx}$.

The advantages of such protocols is that they preserve liveness in the presence of node failures, this is because messages can still be delivered, without blocking, when the number of slave failures is less than $|\frac{m.dst}{2}| - 1$. However, mild blocking does occur whenever the master node is *suspected* of crashing, as it is necessary for a new master node to be elected before *abcast* delivery can resume. Furthermore, as the master node requires a quorum of acknowledgements to proceed, it is not possible for such protocols to be utilised when $|m.dst| < 3$.

Paxos

Paxos[47][48] is arguably the most famous distributed consensus algorithm, in part because of its notoriety as a difficult algorithm to understand, but more notably that it was the first provably

resilient consensus algorithm for asynchronous networks. Although the FLP [27] impossibility states it is impossible for a deterministic consensus algorithm to guarantee progress in the presence of node failures, Paxos is considered *resilient* as it guarantees safety, at the expense of availability, in such circumstances. As consensus and *abcast* are equivalent problems[16], what holds for Paxos holds for any *abcast*.

Although the Paxos algorithm has a reputation for being difficult to understand and even harder to implement, the protocol has been widely utilised in both research and industrial settings. The most famous example of which is its use in the distributed locks system Chubby [11], which was created by Google for coordinating their internal services. In addition to the basic Paxos algorithms, there exists many variations of Paxos that allow the protocol to cater for different application needs [52], such as: handling byzantine failures [50], reducing total message cost [51], reducing latency [49] and increasing throughput [53, 65].

In addition to this suite of Paxos protocols, there is also the increasingly popular RAFT protocol developed at Standford University by Ongaro *et al.* [58]. RAFT provides all of the safety guarantees provided by Paxos, however it has been designed with understandability and simplicity of implementation in mind. This has led to widespread adoption of the protocol in distributed systems, with open source implementations of the protocol now available for the majority of mainstream programming languages.

The remainder of this section provides a brief overview of how the basic Paxos algorithm can work as an *abcast* protocol. We examine the protocol from the perspective of both master and slave nodes.

Master

1. Propose, $propose(m, p.s)$, an *abcast* m to $m.dst$ with a sequence number $p.s$.
2. Wait x amount of time for a quorum of nodes to respond. If a quorum cannot be reached in x then abort and start a new proposal. If a quorum is reached in favour of rejecting $p.s$, then record the largest sequence number returned $p.s'$, set the masters local sequence to greater than $p.s'$ and start a new proposal. If a quorum is reached in favour of accepting $p.s$, then send a commit $commit(m, p.s)$ to all $m.dst$.
3. If the majority of $m.dst$ respond with an accept $accept(m, p.s)$ message, then the *abcast* is considered successful, otherwise start a new proposal.

Slave

1. Upon receiving a proposal, compare its sequence $p.s$ with the highest sequence that this node has currently agreed to, $p.s'$. If $p.s > p.s'$ then reply with an $accept(m, p.s)$, otherwise send a $reject(m, p.s)$ message with $p.s'$ to the proposer node.
2. When a $commit(m, p.s)$ message is received, if this node agreed to $p.s$ then m has to be delivered.

2.3 Coordination Services

Coordination services are centralised systems that can be utilised by distributed systems to provide commonly required services that aid process coordination. Examples of such services are: distributed lock management, low-volume storage and naming. System engineers can implement these features without using a coordination service, however these features are very complex due to their distributed nature[11]. A coordination service hides this complexity, enabling the engineer to focus on the core functionality of their system. Furthermore, incorporating a coordination service into an existing system only requires calls to the services API, whereas retrospectively introducing distributed services into even the simplest of systems is fraught with difficulties.

Coordination systems are usually used by a system to store data that is crucial to their operation, therefore high availability and fault-tolerance are required. Typically, coordination services are implemented as a state machine ensemble, consisting of a small number of nodes[1] that are all replicas of each other. Maintaining a distributed state machine between service nodes allows the service to maintain availability and continue to service client requests when node failures occur within the service. Utilising multiple nodes can also increase the throughput of the service, as each node can simultaneously process client requests and, in the case of read requests, it can respond to the client without interactions with other replica nodes.

In order to maintain a consistent state across the service's replicas, it is necessary for a consensus to be reached between the replica nodes when a client's request modifies the service's state. Without consensus the service's state would become inconsistent between replicas and the

[1] Typically 3-5 nodes and certainly no greater than 10.

distributed state machine would no longer be valid. Such an occurrence could be catastrophic for client applications that depend on the service, as responses from different replicas could return conflicting data, potentially causing an irrecoverable state amongst client nodes.

As consensus is essential to the correct operation of a coordination service, the underlying consensus protocol used by the service greatly affects the characteristics and performance of the service. For example the availability of the service when node failures occur or the throughput and latency of client requests. Furthermore, it is also possible for the service to expose its consensus protocol to client nodes, so that they can utilise its primitives in order to solve agreement problems[30].

2.3.1 Chubby

Chubby[11] is a distributed lock manager developed by Google that is based upon the Paxos[47][48] consensus algorithm. Chubby cluster's typically contain five nodes, however only one node is able to service a client's read and write requests at any one time; this node is called the master. The role of the master is to service client requests and to ensure that the state of all replicas is updated when a write operation is requested. Client write requests are coordinated between replicas using the Paxos consensus algorithm, with a client request being completed when a Quorum of replicas have confirmed the write operation.

Figure 2.3 shows the basic steps involved when a write request is received by the chubby master node; the master node receives the client request (stage 1), *abcast*s it to all slave nodes (stage 2), before processing the request locally and sending a response back to the client node (stage 3).

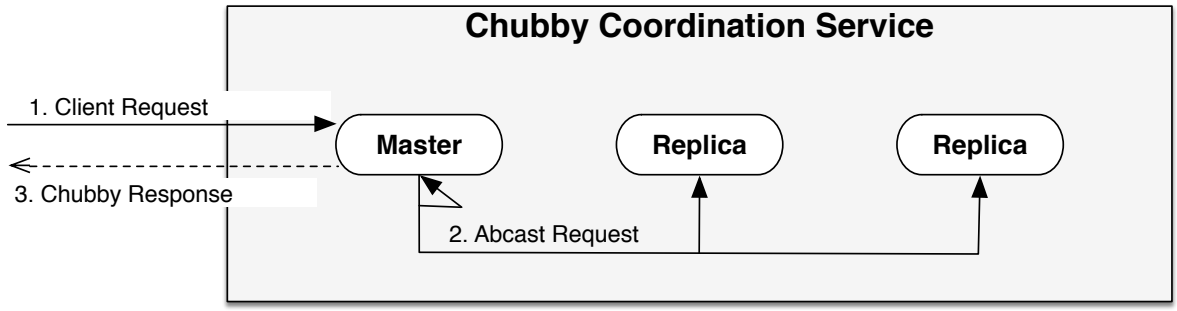

Fig. 2.3: Chubby Write Request at Master Node

Figure 2.4 shows the steps required if a client request is received by a replica node instead

of the master; the replica node forwards the client request to the master node (stage 1), at which point the master node executes the request as if it was originally received by the master node (stages 2-4).

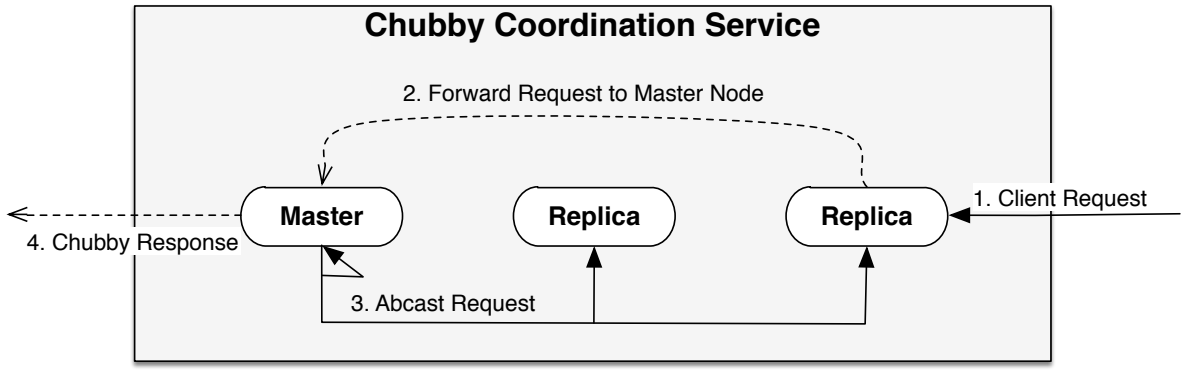

Fig. 2.4: Chubby Write Request at Replica Node

The main advantage of the Chubby system is its focus on high availability and reliability, with production instances reported to have executed for over a year. However, the limitations of the chubby system are caused by its excessive use of the master node. Utilising a single node to handle all read/write requests severely limits the system's throughput as the master node will always become a bottleneck as the number of requests to the service increases. Furthermore, because Chubby utilises the Paxos consensus algorithm, the minimum number of nodes in a chubby coordination service is 3 as a quorum needs to be reached between service members. This can be a disadvantage in systems where fast writes are required.

2.3.2 Zookeeper

Zookeeper[34] is an open source general purpose coordination service released under the Apache Software License Version 2.0 [2]. Similar to Chubby, Zookeeper also employs a master node, as it utilises the quorum-based protocol ZAB[41] to update each replica. However, unlike Chubby, in Zookeeper read requests from client nodes are not restricted to the master node, rather any replica node can handle them. This is advantageous, as it reduces the number of requests that the master node must handle, enabling the master to focus on servicing write requests and processing its own read requests.

Figure 2.5 shows how Zookeeper services read and write requests from clients. Stages 1-3 are the steps executed when a client write request is received, with each replica required to

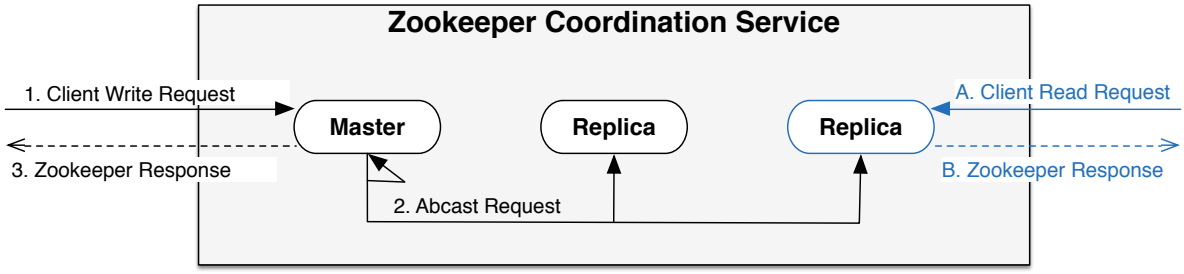

Fig. 2.5: Zookeeper Write at Master, and Read at Replica Node

forward a client write request to the master node. This process is identical to that utilised for all requests in the Chubby system, as shown in figure 2.3. However, unlike in Chubby, client read requests can be handled by any replica node including the master. Stages A and B, in 2.5 show how a Zookeeper service handles client read requests that are received by replica nodes; with a request being received by a replica node (*stage A*), and a response containing the latest version of the requested data being returned to the client node (*stage B*).

Distributing read requests across the entire Zookeeper service mitigates the bottleneck observed in the Chubby service, however the problem is not eradicated as all write requests are still served by a single master node. This limitation results in Zookeeper services favouring workloads that have a read/write ratio of 10:1. Furthermore, when client read requests are handled by the replica nodes it is possible for stale values to be returned to the requesting client. This occurs if a client requests a value from a replica that has missed an *abcast* by the master node due to the replica not participating in an earlier quorum. Therefore, a Zookeeper service can only be considered weakly-consistent.

2.4 In-Memory Databases

In-memory databases[29, 33, 35, 59, 60, 66] are database systems that aim to provide scalable, low-latency data storage. Data is stored in RAM to provide fast data access, and is partitioned across multiple nodes in a cluster for scalability. In order to provide availability and durability in the presence of node failures, each partition is replicated across distinct nodes; the number of replicas utilised for each data partition is the *Replication Factor* (*RF*). Storing data in RAM provides superior read/write performance to traditional disk-based databases, whilst the distributed nature of the database allows it to elastically scale by simply adding additional nodes to store data partitions.

The emergence of simpler NoSQL based data models, such as key/value pairs, has enabled in-memory databases to become a reality. Previously, RDBMS services were the de facto standard for database solutions, however their rigid structure greatly limits their ability to elastically scale due to the difficulty of maintaining table structures and distributing records across multiple nodes[12, 14]. Distributing data across many nodes is essential for in-memory databases, not only to provide availability, but to provide sufficient storage capacity for a database system; RAM per node is typically measured in gigabytes, whereas disk based storage is measured in terabytes.

Finally, due to RAM being a volatile storage medium (i.e its state is lost when power is lost) it is common for in-memory databases to provide a means for persistent storage in the event of power-loss or node crashes. Typically, this is achieved through asynchronous write-requests to a persistent database that utilises the same data model as the in-memory database. However, in some deployments, such as distributed caches, this is forsaken in order to provide applications with the lowest possible response time.

2.4.1 Replication Schemes

Typically, in-memory databases offer two types of replication schemes: *full* and *partial* replication.

Full Replication is a data scheme where each data partition is replicated on every distinct node in the cluster, $RF = |nodes|$. This greatly limits the scalability of the database, as the total number of RPCs required to update each key/value pair increases when additional nodes are

added to the cluster. Furthermore, the maximum storage capacity of the database will always be equal to the RAM size of the least capable node in the cluster. The advantages of using *full* replication is that it can provide high availability, with only a few cluster nodes, as well as providing high-performance when workloads are read dominant; full replication is extremely effective for creating a highly-available distributed cache that sits between the application and a persistent data store.

Partial Replication is a data scheme where each data partition is replicated across a subset of nodes in the cluster, with no node hosting more than a single replica of a given partition and no node storing all partitions of the database [66]. The advantage of this partial replication, is the total size of the database is not limited by the weakest node, rather the collective memory pool of all nodes in the cluster and the RF configured by the system administrator. Therefore, elastic scalability is possible as the database's capacity can be increased by simply adding an additional node to the cluster. Ultimately, the scalability of a partially replicated database is determined by the size of RF; a high RF value ($RF > 3$) will provide high levels of availability and fault-tolerance at the expense of write latency (as all RF replicas need to be updated); whereas too low a value will provide low-latency writes at the expense of fault-tolerance and availability. Defining the optimum size of RF and the total number of nodes required within an in-memory database is a non-trivial task which is explored in detail in [17].

The total capacity of a partially replicated database can be expressed as:

$$\frac{Memory - Sys.Reserve}{RF}$$

Where $Memory$ is the total amount of RAM available across the cluster, $Sys.Reserve$ is the RAM required by other system resources (operating system etc.) and RF is the configured *replication factor*.

2.5 Infinispan

Infinispan [35, 54] is an open-source in-memory database system developed by Red Hat, Inc [62]. that provides users with a JSR-107 [39] compliant, key/value data model. It can be used as a distributed cache, or as a transactional NoSQL key/value store, and supports both full

and partial replication. From its inception Infinispan has been designed to be highly-scalable, this section describes how Infinispan has addressed the challenge of implementing scalable transactions, and the limitations of their current solutions.

As previously stated (§ 2.4.1), full replication is not scalable, therefore the rest of this document will focus on the challenges posed by partial replication schemes; henceforth any reference to key/value replication assumes partial replication. Furthermore, all references to read/write operations are assumed to be in the context of Infinispan transactions, non-transactional operations are not considered.

2.5.1 Key Distribution

A key challenge of implementing distributed data stores is ensuring that each node in the cluster is aware of where each data item is located, so that any node can access data when required. This problem is further exasperated by Infinispan's need to elastically scale.

A naive solution is for each node in the cluster to maintain meta-data about each key/value pair stored in the database, however the maintenance of such data would create a large overhead. Not only would the database require additional space to store the meta-data, but it would also have to update the meta-data stored on each node in the cluster every time a node was added or removed from the cluster. Clearly this is not a scalable solution.

Infinispan solves this problem by utilising a modified consistent hashing algorithm[35, 42, 64] to determine where key/value pairs should be stored; the algorithm utilises the key as a parameter for computing the hash. The hashing algorithm divides the cluster into segments, with each hashed key mapping to a single segment, and associates RF nodes with each segment. The nodes for each segment are stored in an ordered list, with the index of a node determining its replica status. Nodes stored at index 0 are considered the *primary* owner of all keys stored in that segment, and nodes with an index greater than 0 are considered *backup* owners; *primary* owners are used by Infinispan to coordinate various database operations; *backup* owners store data purely for fault-tolerance.

Utilising this consistent hashing algorithm means it is possible for a node to determine the *primary* and *backup* location of any key k in the cluster by calling $hash(k)$. This enables any node in the cluster to deterministically calculate the storage location of any key/value in the

database without a single RPC, therefore reducing the number of RPCs and aiding scalability.

2.5.2 Key/Value Operations

Infinispan provides implementations of all of the operations specified in JSR-107[39], as well as additional operations for interacting with the underlying key/value store. However, two operations $get(k)$ and $update(k,v)$ are particularly significant as they provide the foundations of more complex operations. In this section we detail the purpose of each of these operations, as well as the cost and workflow of these operations with respect to RPCs.

Get

The $get(k)$ operation is a simple read-only operation that returns the value, v, associated with the key, k. This operation may require RF RPCs in order to retrieve v, as it is possible that the operation is executed on a node that does not host the segment which contains k. Hence, it is necessary for v to be retrieved from a remote node.

Infinispan performs $get(k)$ as follows: If the local node, n, executing $get(k)$ hosts k, then simply return the local v associated with k. Otherwise, n must send an RPC to all RF nodes contained within k's segment to request the latest value of k. Figure 2.6 shows the RPCs required for a $get(k)$ operation when k is not hosted on the node executing the operation.

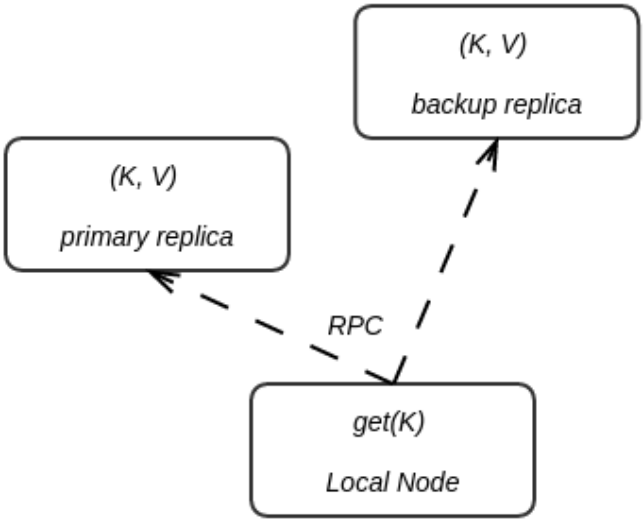

Fig. 2.6: Infinispan Get Operation

Update

The $update(k, v)$ operation associates the key, k, with a new value v irrespective of the previous value associated with k. Unlike $get(k)$, the $update$ operation always requires at least one RPC when $RF > 1$, as k's association with v must be propagated to all RF replicas in k's segment. Figure 2.7 shows the RPCs required by $update(k, v)$ when the primary replica for k is hosted on the node executing the operation. *N.B.* when $update(k, v)$ is executed on a node that does not contain any replicas of k, then the RPCs required will be the same asin Figure 2.6.

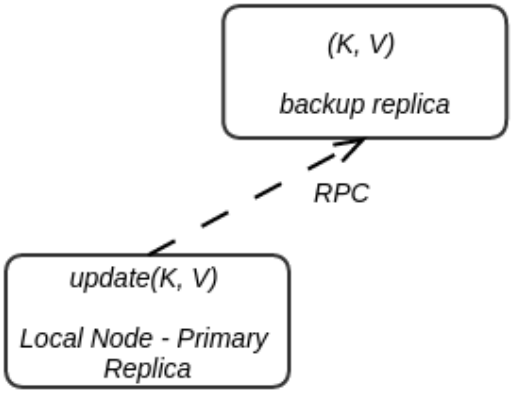

Fig. 2.7: Infinispan Update Operation

2.5.3 Transactions

Unlike many NoSQL databases Infinispan can be used as a transactional data-store, with both the JTA[40] and XA[69] standards supported. Both optimistic[45] and pessimistic transactions[5] are provided, however optimistic transactions are used in the default Infinispan configuration in order to reduce contention and the chance of deadlock. The traditional Two Phase Commit protocol (2PC)[4] is utilised for implementing the locking strategy in both optimistic and pessimistic transactions. In addition to the two traditional lock based transactions, Infinispan also offers a lock-free total order transaction scheme that relies on an atomic multicast protocol to coordinate transactions.

The remainder of this section details the topology of Infinispan transactions, the relaxed ACID guarantees that they provide, and an in-depth explanation of how transactions are coordinated using locking (2PC) and lock-free (Total Order) schemes.

2.5.3.1 Transaction Topology

For all transaction flavours offered by Infinispan, the following always applies. A Transaction (Tx) is executed *locally* by the transaction coordinator $Tx.c$, before a prepare $prepare(Tx)$ message is sent to all nodes $Tx.dst$ involved in the Tx. Only $get(k)$ operations are executed locally by $Tx.c$. Once all $get()$ operations have been satisfied locally, the values of k are included in a $prepare(Tx)$ message that is sent to $Tx.dst$. Each member of $Tx.dst$ then validates Tx, $validate(Tx)$, before committing Tx $commit(Tx)$. It is only during the $commit(Tx)$ operation that write operations are executed and these operations are only executed on nodes that host a key that is being inserted or updated. For example if k is stored on N_1 and N_2, the operation $update(k,v)$ will only be executed by nodes N_1 and N_2.

Note, Infinispan does offer some operations, such as $put(k,v)$, that require a $get(k)$ to be executed locally by $Tx.c$, followed by $update(k,v)$ at the nodes hosting k, however for the sake of brevity it is assumed that all operations are exclusively read or write. Furthermore, write operations are often executed on nodes other than the host of a key being updated so that they can be stored in a local cache in order to reduce the total number of RPCs, however this optimisation is not core to the Infinispan protocol and can be disabled, therefore this functionality is also omitted for the sake of brevity.

2.5.3.2 Relaxed ACID

Infinispan transactions abide by the ACID[31] properties, however the *Isolation* and *Durability* guarantees are more relaxed than those provided by traditional RDBMS transactions; Durability is relaxed as a consequence of RAM being volatile, and Isolation level is relaxed from the traditional 1-copy serialisability in order to reduce the overhead of maintaining distributed transactions in a *partially replicated* context.

Durability

Infinispan provides two mechanisms for providing durability: the first is the use of redundant key backups, i.e. the *replication factor*, and the second is an optional mechanism that allows key/value pairs to be persisted to a separate persistent database. The latter is available in two different configurations, *write-through* and *write-beyond*.

Write-through is a synchronous operation, that ensures that an insert/update operation on an Infinispan key will not complete until the value has been updated in the cache and it has been updated at the persistent store. This ensures that the contents of the cache will always be consistent with the persistent store, therefore guaranteeing that in the event of a system wide crash all committed key/value pairs will be preserved. The disadvantage of this approach is that the users of the cache lose the performance benefits provided by in-memory storage as any update/insert operation will always take at as long as storing the pair in a persistent store. Ultimately, write-through is only an appropriate solution in read-heavy workloads that require a strong emphasis on durability.

Conversely, write-beyond offers asynchronous persistence. Key/value updates and inserts complete as soon as the operation has completed in the cache, and the values are persisted in the background using a separate thread to the users request. This ensures that the users operation is returned as quick as possible, and low-latency is maintained, however it presents a small window in which the cache is not consistent with the persistent data store. Therefore, it is possible for the most recent key/value write operations to be lost if all nodes containing *RF* backups simultaneously crash or the entire cluster goes down.

Isolation

Infinispan does not provide support for 1-copy serialisability in its transactions, instead it provides two different isolation criteria: *Read Committed (RC)* and *Repeatable Read (RR)*.

RC - All calls to a key $get(k)$ return the last value of k committed by a transaction.

RR - The value returned by the first call to $get(k)$ will be used for all subsequent calls to $get(k)$ within a transaction.

RC is advantageous as it ensures that all calls to $get(k)$ always return the last committed value of k at the time of the $get(k)$ operation, however each call to $get(k)$ may require a RPC as k may not be stored on the local node due to k only being partially replicated across the cluster. Thus, RC can be detrimental to performance in transactions that consist of multiple reads to the same key. If such transactions are prevalent in a workload, it may be advantageous to utilise RR isolation in order to reduce the total number of RPCs required by a transaction.

Infinispan implements RR by storing the value returned by $get(k)$ in the transaction manager's context, and simply returns the stored value for any subsequent calls to $get(k)$ within this transaction. A consequence of adopting RR is that the potential for stale values to be utilised by a transaction, Tx_j, increases. This occurs if another transaction, Tx_i, commits before Tx_j and updates k, as Tx_j will still be utilising the previously committed value of k. If Tx_j attempts to perform a write operation utilising this stale value of k, then the values committed in Tx_i will be ignored; this is referred to as a *write-skew*. To detect when *write-skews* occur, Infinispan provides an optional *Write Skew Check (WSC)* that enables transactions to be aborted when such anomalies are detected[2].

The WSC determines whether a read operation, $v = get(k)$, in a transaction Tx_j, has been invalidated by a concurrent transaction Tx_i committing a $update(k, v')$ operation during the lifetime of Tx_j. If Tx_i has committed a write operation on k between Tx_j performing $get(k)$ and $update(k, v+1)$, then the WSC will detect this and allow the transaction manager to abort Tx so that the erroneous value $v+1$ is not committed.

2.5.3.3 Two-phase Commit Protocol

The Two-phase commit protocol (2PC)[4] is the traditional approach to coordinating distributed transactions, and as such its benefits and limitations are well understood. Infinispan utilises the 2PC protocol for coordinating both optimistic and pessimistic transactions.

Example Scenario

Consider the following scenario: a node N_1 executes a transaction Tx that consists of a single write operation $update(k, v)$, however the primary and backup of k are stored on N_2 and N_3 respectively, therefore it is necessary for Tx to be committed at N_1, N_2 and N_3 ($Tx.dst = N_1, N_2, N_3$). To ensure that all three nodes come to the same conclusion about the transaction, whether to commit or abort, the 2PC protocol is used.

[2] WSC is only available when Repeatable Read isolation is utilised, as *write-skews* are not possible with Read Committed isolation.

Protocol Details

The 2PC is leader-based consensus algorithm that is specifically designed for coordinating distributed transactions, as the name suggests the protocol consists of two distinct phases: *voting* and *commit*.

Voting Phase. The transaction coordinator $Tx.c$, ($Tx.c = N_1$) sends a prepare, $prepare(Tx)$, message to all $Tx.dst$. Upon receiving the $prepare(Tx)$ message, all members of $Tx.dst$ will validate, $validate(Tx)$, the transaction and decide whether the Tx should be committed or aborted. Once a decision has been made by a node, it sends its vote, $vote(Tx)$, to $Tx.c$, and awaits further instructions.

Commit Phase. Once the $Tx.c$ has sent $prepare(Tx)$ to all $Tx.dst$, and has validated Tx, it waits to receive a $vote(Tx)$ from all $Tx.dst$. If all $Tx.dst$ respond with a commit verdict, then $Tx.c$ sends a final commit message, $commit(Tx)$, to all $Tx.dst$, and commits Tx locally. However, if $Tx.c$ receives a single vote in favour of aborting Tx, then Tx must be aborted, so $Tx.c$ sends a abort message $abort(Tx)$ to all $Tx.dst$; $Tx.c$ does not need to wait for all $vote(Tx)$ before issuing $abort(Tx)$, instead $abort(Tx)$ is issued as soon as the first abort vote has been received. Finally, upon receiving a $commit(Tx)$ or $abort(Tx)$ each member of $Tx.dst$ will abort or commit Tx locally.

Figure 2.8 shows all of the sequences involved in 2PC based upon the example scenario, and assumes that all $Tx.dst$ vote in favour of committing Tx.

Key Locking

As previously stated, Infinispan has the notation of *primary* and *backup* owners of key/value pairs. This is utilised by many functions within Infinispan, but one of the most important uses is for determining which data replica should be locked during write transactions. Infinispan always locks the *primary* owner of k for write operations, never a *backup*, therefore allowing each transaction to acquire a lock on k at a single node. Thus limiting the number of RPCs required to one, instead of RF RPCs. For example in the previous scenario, if the *primary* owner of k was N_2 and the backup N_3, then the write lock for k would only be acquired at N_2.

The time at which k's lock is acquired is a defining characteristic of how transactions are handled. Infinispan provides two different approaches, the more cautious Pessimistic trans-

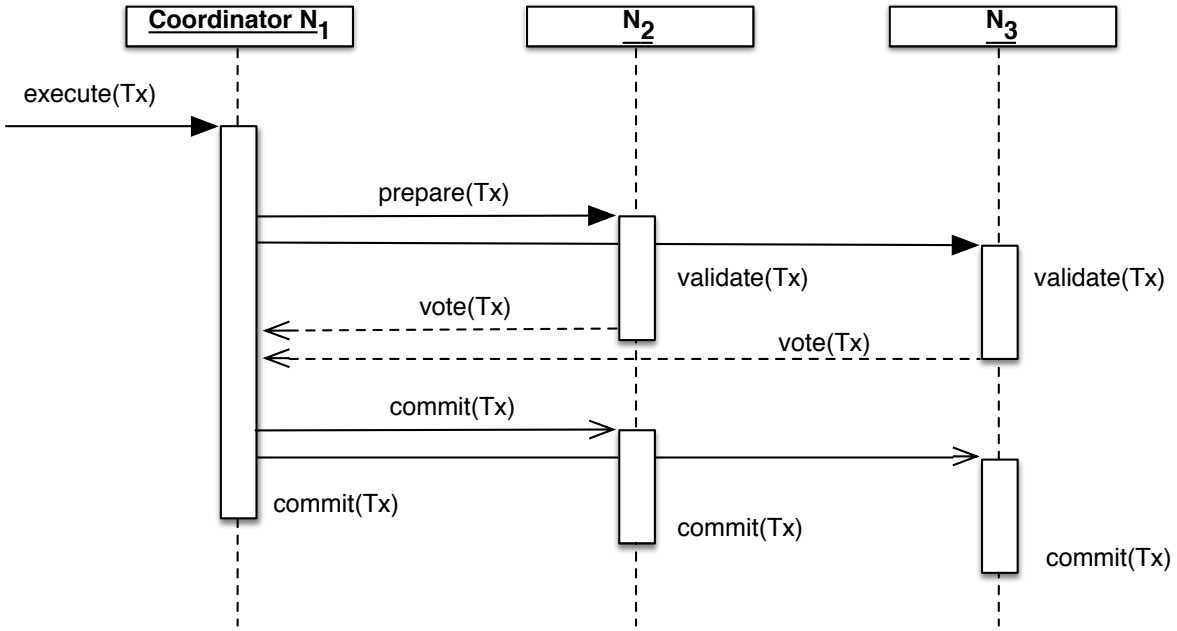

Fig. 2.8: 2PC Sequence Diagram

actions which utilises pessimistic locking, and Optimistic transactions that utilises optimistic locking. Each approach is detailed below along with the benefits and limitations of each approach.

Pessimistic Locking. When pessimistic locking[5] is used, the lock on k is acquired the first time that a write operation is performed on k in Tx and it is held until Tx either commits or aborts.

```
Tx.begin();
update(k,v); // Lock is acquired
Tx.commit(); // Lock is released
```

This means that a RPC is issued for every write operation in the transaction, at the time the operation is encountered. Not only does this result in an increase in network traffic, due to the number of RPCs required being equal to the number of write operations, but it also means that locks are held for a longer period of time, increasing the likelihood of deadlocks occurring [3].

Optimistic Locking. When optimistic locking[45] is used, the lock on k is only acquired during the $prepare(Tx)$ phase of the transaction. This means that no additional RPCs are required for locking, instead the lock is acquired by k's primary owner when it receives $prepare(Tx)$

[3]Details of which can be found in the next section "2PC Limitations".

from $Tx.c$ and Tx is processed locally. The lock is then released by k's primary owner when Tx commits or aborts.

Acquiring locks during the prepare phase of a transaction means that is possible for a *write-skew* (§ 2.5.3.2) to occur, therefore an optimistic transaction can be aborted due to failing the WSC (if enabled). However, acquiring locks during the prepare phase also reduces the total number of RPCs required by a transaction, which can improve scalability and throughput.

Optimistic locking is the default locking strategy employed by Infinispan, henceforth all references to lock-based transactions in Infinispan assume optimistic locking.

2PC Limitations

The key limitation of utilising the 2PC protocol with locking (both optimistic and pessimistic) is that it is susceptible to deadlocks. Deadlocks occur when two concurrent transactions are trying to acquire a lock on the same set of keys. Consider a situation where Tx and Tx' are executing concurrently, and both transactions want to write to key k_1 and k_2. It is possible for Tx to acquire a lock on k_1 with $update(k_1)$, and Tx' to acquire k_2's lock with $update(k_2)$. In this scenario, it is impossible for either Tx or Tx' to progress as both are waiting to acquire the locks held by each other, hence deadlock.

Infinispan utilises timeouts in order to recover from deadlocks, with a default timeout of 10 seconds. The transaction coordinator will wait a maximum of 10 seconds for k's lock to become available, if k's lock does not become available during this period, then the transaction coordinator aborts the transaction. The limitations of this approach are that it is possible for *false suspicions* to occur, as transactions can timeout due to other circumstances, such as high network load. Furthermore, in workloads where high levels of contention are present, deadlock becomes increasingly likely, resulting in more transactions aborting, which ultimately leads to a drop in transaction throughput and Infinispan's request latency increasing.

2.5.3.4 Total Order Commit Protocol

In addition to the 2PC locking approach, Infinispan also provides a lock-free total order commit protocol, that provides the same guarantees as 2PC (*Read Committed*, or *Repeatable Read*) without locking key/values during write operations.

The key benefit of a total order commit, is that it does not utilise locks to ensure ACIDity. Instead it utilises the guarantees (G1-G4 § 2.2) provided by *abcast* and *amcast* protocols to ensure that transactions are processed sequentially and in the same total order at all destinations. The absence of locks removes the potential for distributed deadlocks, which reduces the total number of aborting transactions, therefore an increase in transaction throughput is expected.

Ruivo *et al.*[64] conducted a thorough performance evaluation of the Total Order Commit protocol, utilising Red Hat's bespoke benchmark, RadarGun[61], and the industry standard benchmark, TPC-C[67]. For each benchmark they found that when RC or RR consistency guarantees (§ 2.5.3.2) were utilised, the transaction abort rate was reduced dramatically when key/value pairs were exposed to both high and low levels of contention. As expected this resulted in an increased throughput rate and a reduction on the average latency encountered per transaction. The difference between abort rates when comparing 2PC locking and Total Order Commit, was much smaller when the benchmarks were performed using RR consistency with the WSC enabled. This is because when the WSC check is enabled it is possible for transactions to abort if a key/value pair has become invalidated by another concurrent transaction. However, despite the difference in abort rates being reduced, the Total Order Commit protocol still provides a marked improvement in transaction throughput and latency over the 2PC approach.

In addition to eliminating deadlocks, the use of total order commit allows transactions to be committed in only one phase (1PC) when RR or RC is used. The workings of 1PC and how the WSC is executed in Total Order commits are explored below.

One-Phase Commit

Consider the scenario in 2.5.3.3. The transaction coordinator $Tx.c$, ($Tx.c = N_1$), executes the transaction locally (i.e. all $get(k)$ operations are resolved) and sends $prepare(Tx)$ using an atomic multicast protocol to $Tx.dst$. Because $prepare(Tx)$ is sent to $Tx.dst$ using an *amcast* protocol, we can guarantee that all $Tx.dst$ will receive $prepare(Tx)$ in the same total order. Therefore if RR or RC is used, each transaction can be committed as soon as it is received by a node without violating Infinispan's ACID properties. Figure 2.9 , below, shows the sequences involved in a 1PC transaction.

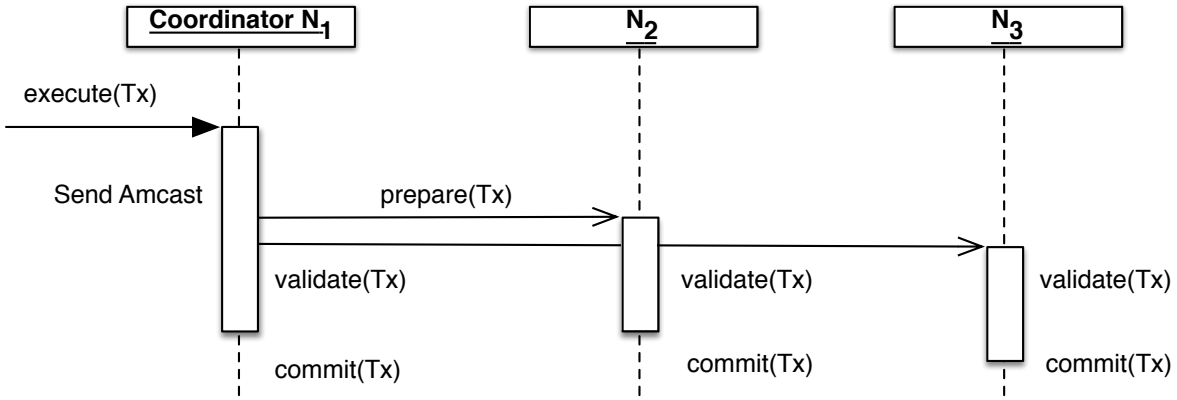

Fig. 2.9: Total Order 1PC Sequence Diagram

Two-Phase with WSC

If RR is utilised with WSC enabled, the total order commit becomes a two-phase protocol. The first phase is the same as above, with $Tx.c$ sending $prepare(Tx)$ to all $Tx.dst$ using an *amcast* protocol, however it is not possible to commit the transaction instantly, instead an additional voting stage is required. All $Tx.dst$ validate the transaction based upon the WSC criteria and decide whether the transaction should be committed or aborted, this vote $vote(Tx)$ is then sent to $Tx.c$. The $Tx.c$ waits to receive a $vote(Tx)$ from all $Tx.dst$, before sending a $commit(Tx)$ or $abort(Tx)$ to all $Tx.dst$. Finally, upon receiving a $commit(Tx)$ or $abort(Tx)$ each member of $Tx.dst$ will abort or commit Tx locally.

The overhead of this additional phase is slightly reduced by two minor optimisations. First, the $Tx.c$ does not have to receive a vote from all replicas hosting a key, just one, as the processing of a transaction is deterministic it is guaranteed that all replicas reach the same conclusion during WSC validation. Secondly, like 2PC, as soon as a single $abort(Tx)$ vote is received by $Tx.c$ the transaction is aborted.

Figure 2.10 shows the sequences involved in a transaction that utilises the WSC. In this figure we have assumed that $Tx.c$ receives $vote(Tx)$ from both N_2 and N_3 before sending $commit(Tx)$, however, as stated above, this is not essential, and it is valid for $Tx.c$ to have only received $vote(Tx)$ from N_2 or N_3 before sending $commit(Tx)$.

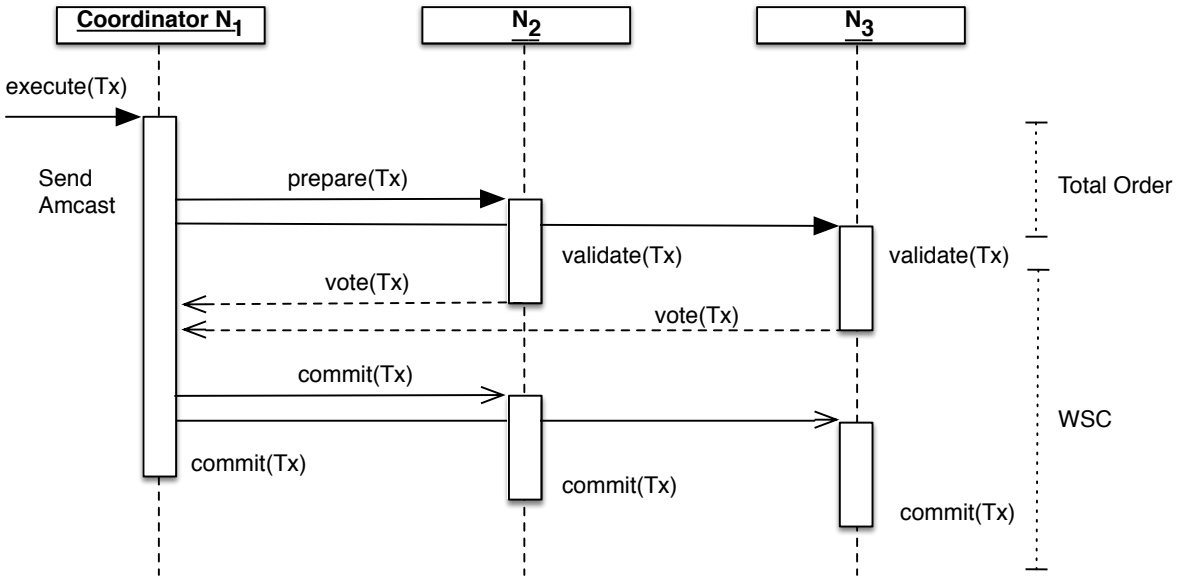

Fig. 2.10: Total Order Commit with WSC Sequence Diagram

2.5.3.5 Total Order Anycast - Atomic Multicast Protocol

Total Order Anycast (TOA)[64] is the *amcast* protocol currently utilised by Infinispan for coordinating Total Order transactions (2.5.3.4). It is a GM dependent protocol that, like Newtop[23], utilises logical clocks and acknowledgements, to solve C1 and C2 (2.2) respectively.

TOA's structure is very similar to the 2PC protocol, in that it also consists of two distinct phases (shown in Figure 2.11), both of which are required for a message m to be delivered. The *ack phase* requires that all destinations in $m.dst$ acknowledge the message origin $m.o$, and the *delivery phase* involves $m.o$ instructing all $m.dst$ to deliver m. Thus the ack and delivery phases are the equivalent of the *vote* and *commit* phases, respectively.

The ack phase consists of all $d' \in m.dst - \{m.o\}$ acknowledging m by sending $ack_{d'}(m)$ to $m.o$. Once $m.o$ has received all $ack_{d'}(m)$, the ack phase is complete, and C1 is guaranteed as all $m.dst$ are known to have received m.

The delivery phase in TOA is necessary to ensure that all $m.dst$ know the final total order of m. Like the Newtop protocol, TOA ensures C2 by piggybacking the timestamp of a sending node's logical clock onto all m, $ack(m)$ and $deliver(m)$ messages sent from that node. However, in TOA the final timestamp of m is always finalised by $m.o$ after the ack phase has completed, with the $deliver(m)$ message dictating the final timestamp of m to all $m.dst$ in order to dictate m's place in the total order. Figure 2.11 shows the communication stages required for *amcast*ing

m between nodes N_1, N_2 and N_3.

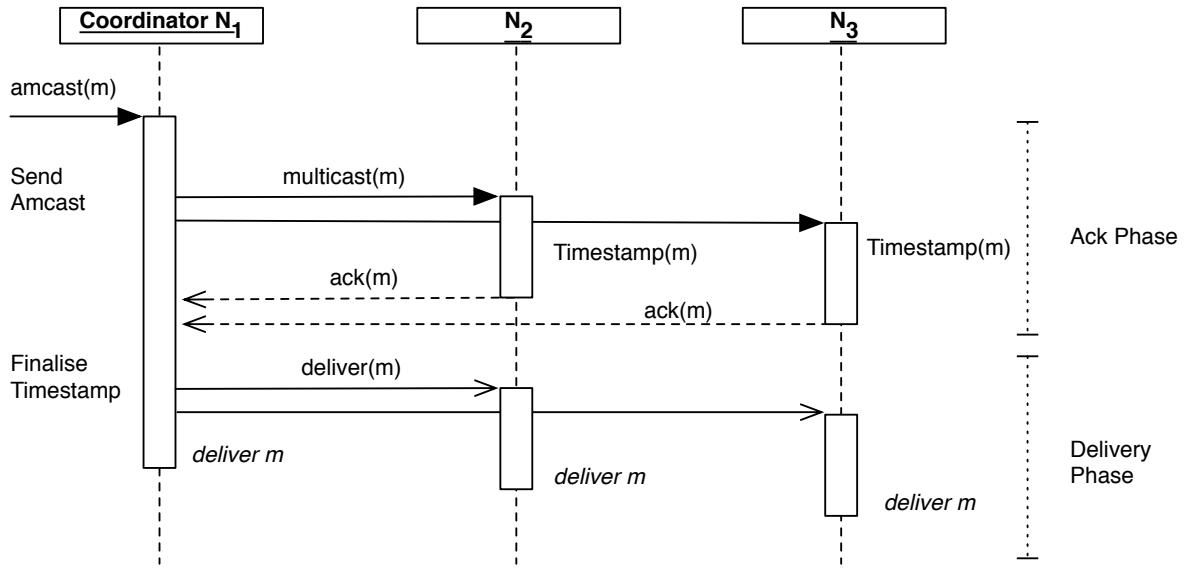

Fig. 2.11: Total Order Anycast Sequence Diagram

The advantage of utilising $m.o$ as a central coordinator, opposed to all $m.dst$ acknowledging each other, see Figure 2.1, is that the total number of messages involved in a single *amcast* is reduced as $|m.dst|$ increases. The total remote message cost for TOA and Newtop is expressed below:

Let $x = |m.dst| - 1$. In TOA, $m.o$ transmits $2x$ messages and the other nodes in $m.dst$ transmit 1 ack each. So TOA's message cost is $3x$. Whereas, in Newtop, each node in $m.dst$ sends x messages each, hence the cost is $x(x+1)$. Note that:

$$x(x+1) > 3x \quad \forall \quad x > 2$$

i.e. when $|m.dst| > 3$.

TOA Limitations

The TOA protocol suffers from the same limitations as other GM based protocols, such as NewTop, most notably that message delivery is blocked in the presence of node failures. In the context of Infinispan total order transactions, a crashed node c will cause all transactions that interact with c to block until the GM service issues a new view. This causes the *liveness* of all nodes interacting with c to be lost in the interim period, as the blocked transactions will not be

able to commit, resulting in a loss of throughput.

Another limitation of the TOA protocol is that it does not scale well as the number of destinations N increase. All multicast protocols incur $1->N$ communication (i.e. $m.o$ multicasting m to all $m.dst$) as it is necessary for each destination to receive m. However, $N->1$ communication (i.e. N destinations in $m.dst$ sending an acknowledgment to $m.o$) is expensive, as the total time taken is equal to the slowest N. Therefore, any protocol that relies on $N->1$ communication is more liable to encounter slow or crashed nodes as N increases, and is ultimately more likely to block. This means that as the number of operations in a transaction increases and keys become more distributed, the latency involved in each transaction will also increase, severely hampering Infinispan's ability to scale elastically.

Finally, the TOA implementation used by Infinispan does not guarantee *uniform agreement* (§ 2.2) when a message originator crashes. Consider the scenario shown in Figure 2.11. If the coordinator N_1 fails after sending $deliver(m)$ to N_2, but before sending the same message to N_3, then N_2 delivers m but N_3 cannot as it never received $deliver(m)$ from N_1.

2.6 JGroups

JGroups [38] is a network framework written in the Java programming language, which provides implementations of many network protocols that can be utilised on their own, or as part of a network stack. Furthermore, the framework provides an abstraction that allows users to write their own network protocols that can be utilised within the network stack alongside existing JGroups protocols. Infinispan utilises the JGroups framework for all distributed communication and consequently all of the network protocols presented in this thesis have also been implemented using the JGroups framework.

As previously stated, the JGroups framework provides implementations of various network protocols. Of particular interest to this project are TOA and GMS, as they are both utilised by Infinispan for atomic multicast and group membership services, respectively. The TOA protocol has already been discussed in detail in section 2.5, therefore the remainder of this section details the inner working of GMS and the protocols from which it depends. It is necessary to detail these protocols in order to show that, with a very high probability, node crashes will be detected within a matter of seconds by the GMS protocol. Furthermore, the inner workings of

these protocols are essential for understanding the design decisions made in chapter 4 and the experiments conducted in chapter 6.

Group Membership Service

The GMS protocol keeps track of the current members of the network group by issuing network views, with each view containing the address of each respective member. Upon a node joining or leaving the network, a new view is issued to all nodes whose address appears in the updated view of the network. The purpose of the GMS protocol is to update the current view of the network when changes occur, however it does not detect these changes itself, instead it relies on lower level protocols in the JGroups stack. Discovering new nodes is trivial, therefore the inner workings of these operations are not detailed further. However, the detection of node failures is non-trivial, due to the FLP impossibility stated earlier, and as such the GMS protocol relies on three other protocols to detect node crashes. These three *failure detection* protocols are called *FD_SOCK*, *FD_ALL* and *VERIFY_SUSPECT*; all of which pass a SUSPECT message up the stack when a node is suspected of crashing. The remainder of this section details the workings of each protocol as well as providing a short conclusion that states how effective these protocols are at correctly identifying a node as crashed.

FD_SOCK

FD_SOCK is the lowest of the three protocols in the stack, and it utilises a 'ring' of TCP sockets, which is established between each node in the current view, to detect if one or more nodes become inoperative [4]. If a node's TCP socket is abruptly closed, then FD_SOCK suspects that the node has crashed and issues a SUSPECT message. Conversely, if a node wishes to leave the view gracefully, i.e it has not crashed, then a leaving message is sent around the ring of TCP sockets before the node closes its socket. This leaving message is sent when the JGroups shutdown hook is activated during the normal shutdown process of a Java program (calling System.exit() or requesting the process is terminated gracefully at the OS level).

[4] All of the experiments detailed in this thesis utilise UDP packets for sending unicasts, however the TCP sockets are still open as part of the FD_SOCK protocol and are present purely for failure detection.

FD_ALL

FD_ALL is a failure detector protocol that utilises a simple heartbeat protocol [3] to issue SUSPECT messages. Each node periodically sends a heartbeat message to all other nodes in the current view, and suspects another member of crashing if a heartbeat message has not been received after a specified timeout. By default, FD_ALL utilises a timeout value equal to 40 seconds with each heartbeat message sent every 8 seconds.

VERIFY_SUSPECT

Finally, the highest failure detection protocol in the stack, is the VERIFY_SUSPECT protocol. This protocol aims to reduce the chances of a node being falsely suspected of crashing by intercepting SUSPECT messages, sent from lower in the stack, and attempting to contact the suspected node for a final time. If no response is received within 1.5 seconds, then the SUSPECT message is sent upto the GMS protocol and the node will be excluded from the current view. Otherwise, the original SUSPECT message is discarded as the suspected node must be alive if it was able to respond to this protocol.

Summary

When utilised simultaneously the three protocols described above provide an effective method for detecting node crashes, with initial experiments showing that the FD_SOCK protocol was particular effective at detecting crashed nodes due to it not relying on large timeout values. Furthermore, due to the combination of TCP sockets, the large timeout of FD_ALL and the additional waiting period of VERIFY_SUSPECT, the probability of a node being falsely suspected of crashing is very small.

Chapter 3

AmaaS - Atomic Multicast as a Service

This chapter introduces the concept of providing *amcast* messaging as a service to members of a cluster.

First we describe the rationale behind Amaas, then we explore the requirements of such a service and the challenges involved in meeting them. This is followed by the introduction of SCast - an *amcast* protocol that utilises the AmaaS model. Finally, we discuss the limitations of existing *abcast* protocols in the context of an AmaaS ordering service, and propose the need for a new non-blocking *abcast* solution.

3.1 Rationale

Total order commit protocols can be utilised by distributed systems to coordinate transactions without the use of locks. They reduce the abort rate of transactions when contention is high, as system deadlocks cannot occur when distributed locks are not present. Therefore, they can aid scalability and improve transaction throughput [64].

The limiting factor of a total order commit protocol is the underlying mechanism used to provide atomic guarantees on message delivery. For example, the *amcast* protocol, TOA, currently utilised by Infinispan, does not scale well as the number of destinations N increase, as $N->1$ communication is expensive (§ 2.5.3.5). Similarly, other GM protocols such as Newtop [23], exasperate the problem, as the number of messages required to perform an *amcast* increases dramatically as N increases. Finally, quorum based protocols provide even less scalability, than GM based protocols, as their inability to *amcast* messages to disjoint sets of nodes

typically requires all nodes in the cluster to participate in an *abcast*.

As the atomic multicast protocols required by the total order commit protocol are inherently unscalable, we argue that transaction coordinators should not be burdened with the responsibility of reaching a consensus on transaction ordering. We propose that transaction ordering should not be conducted between the transaction coordinator and the Infinispan nodes participating in the transaction, rather transaction ordering should be provided by an independent ordering service. This decoupling of ordering and transactions, allows the transaction coordinator to request and receive a total order from the service, before multicasting its $prepare(Tx)$ message to all nodes participating in the transaction. Such an approach results in the number of nodes involved in a transaction having no effect on the total number of nodes participating in consensus, instead the number of nodes in a transaction only increases the number of unicasts required when sending the transaction's $prepare(Tx)$ message [1].

The most efficient implementation of such an ordering service, in terms of latency, would consist of a single node providing transaction ordering to all Infinispan nodes. However, as the progress of all Infinispan transactions is dependent on this ordering service, it is necessary for crash-tolerance to be provided. Therefore, we envisage such a service consisting of a dedicated set of nodes which act as a single state machine, with consensus being required amongst all nodes within the service, when a new ordering request is received from an Infinispan transaction. Hence, such an approach limits the number of nodes required to reach a consensus on ordering to the total number of nodes providing the ordering service, regardless of the number of nodes involved in a transaction.

We call this system model Atomic Multicast as a Service (AmaaS), and refer to the existing Infinispan approach as *peer-to-peer* (P2P). The next section of this chapter provides a detailed description of this system model.

3.2 System Model

The AmaaS approach introduces the concept of a dedicated set of nodes providing ordering to disjoint sets of nodes involved in distributed transactions. We define the nodes providing

[1] Such a cost is unavoidable as the number of destinations will always determine the minimum number of unicasts required to disseminate a $prepare(Tx)$ message.

the ordering service as service nodes, *s*-nodes for short, and denote them as $N_s1 \ldots N_sn$, where $n \geq 2$. As we consider a crash-tolerant ordering service essential, we do not consider $n = 1$. We refer to the consumers of this ordering service as client nodes, or *c*-nodes for short, and denote them as $N_c1 \ldots N_cx$; where x is the total number of *c*-nodes utilising the ordering service.

For all unicasts sent between correct nodes, we assume that messages are sent via a reliable network protocol, such as TCP[13] or Reliable UDP[63], and they arrive within some unknown delay bound. Furthermore, all references to a message being *multicast* assumes that the message is unicast to all of the destinations in the message's destination set. Therefore, if all unicasts between correct nodes are guaranteed to be received, it is also guaranteed that all multicast messages sent between correct nodes will eventually be received by all destinations.

In our system model we assume that any node, both *s*-nodes and *c*-nodes, can crash at any time, however we do not consider other types of node failures such as byzantine failures. In order to detect node crashes we assume that a GM service and associated failure detection protocols, such as the ones detailed in section 2.6, are utilised by both service and client nodes. We assume that a node crash will eventually be detected by the GM service and an updated view of the current network will be received by all nodes once the GM service has detected the crash; hence all nodes within the current view will eventually know of a crash and receive a new view. Furthermore, we assume that client and service nodes utilise their own GM service, as it is envisaged that the service will operate independently of the client nodes. Therefore, if an *s*-node crashes, only *s*-nodes will receive a new view, with clients only becoming aware of the *s*-node crash when they interact with the ordering service. Similarly, if a *c*-node crashes, only *c*-nodes will receive a new view.

3.3 AmaaS Requirements

The AmaaS model consists of two distinct entities: *s*-nodes and *c*-nodes. This section will explore the requirements that need to be met in order for the AmaaS model to be effective. We consider requirements from the perspective of both client and service nodes.

Client Requirements

CR1 A *c*-node must be able to send *amcast*s to multiple destination sets that may overlap.

CR2 A *c*-node should be able to submit its *amcast* requests to any one of the correct *s*-nodes.

CR3 Upon receiving *amcast* m_j via the ordering service, a *c*-node must be able to deduce every m_i that is ordered before m_j and has itself as a multicast destination:

$$\text{all } m_i: \quad c\text{-node} \in m_i.dst \cap m_j.dst \text{ and } m_i \text{ is ordered before } m_j$$

Service Requirements

SR1 The service must provide crash-tolerance ($|s\text{-nodes}| > 1$).

SR2 The service must be highly available and non-blocking in the event of an *s*-node crashing or even being suspected of having crashed.

SR3 All *s*-nodes must process client requests in the exact same order.

SR4 All *s*-nodes should be able to handle client requests.

3.4 SCast: Atomic Multicast Protocol for AmaaS

We have developed a protocol for the AmaaS model, which we call SCast; as the protocol offers atomic ordering for multicast messages as a service, hence Service Multicast - SCast. The SCast protocol enables atomic multicasting by *c*-nodes utilising an ordering service, precisely defining the interactions required between *c*-nodes and the ordering service in order to ensure the atomicity of each multicast.

Inside the ordering service, the SCast protocol maintains a replicated state machine amongst *s*-nodes that stores the total order timestamps attributed to *c*-node requests. SCast's ability to meet requirements SR1 - SR3, is dependant on the characteristics of this underlying *abcast* protocol utilised for state machine replication between *s*-nodes, with only SR4 guaranteed regardless of the *abcast* protocol used [2]. For example, consider requirement SR2. If the under-

[2]This can be guaranteed even if a leader based *abcast* protocol is utilised, as client requests can simply be forwarded to the leader node. Such an approach can be seen in both Chubby and Zookeeper

lying *abcast* protocol is GM based, then message delivery will block in the event of a node crash, therefore it is not possible for SR2 to be met with a GM based protocol as the *s*-nodes will become blocked if an *s*-node fails. Ultimately the performance and *liveness* of an ordering service implementing SCast is tightly coupled to that of the underlying *abcast* protocol which is utilised for state machine replication between *s*-nodes.

As SCast defines how *c*-nodes interact with the ordering service and how *s*-nodes maintain a replicated state, it is necessary for the protocol to be explained from the perspective of both client and service nodes. In the explanations below, for simplicity, we assume that Infinispan is executing a 1-Phase Total Order transaction, without a second WSC phase, and that the transaction has already been successfully executed locally. Finally, we refer to a collection of *s*-nodes providing the *amcast* service as the *ordering service*.

3.4.1 Protocol Overview

Client Nodes

Stage 1 of the client SCast protocol starts once a transaction coordinator, $Tx_i.c$, has completed its local execution of Tx_i and it is ready to *amcast* a $prepare(Tx_i)$ message to $Tx_i.dst$ as required by the total order commit protocol. The key stages of the SCast protocol from the perspective of a client node are detailed below:

C1 Choose and Inform Backup Coordinator - Transfer the contents of Tx_i to a backup coordinator.

C2 Request Ordering - Send an ordering request $req(Tx_i)$ to, and wait for a response message $rsp(Tx_i)$ from, the service.

C3 Receive Ordering and Multicast Transaction - Receive $rsp(Tx_i)$ and multicast Tx_i with its ordering data to all $d \in Tx_i.dst$ as $mcast(Tx_i)$.

C4 Order Transaction - Receive $mcast(Tx_i)$ from $Tx_i.c$, and deliver Tx_i locally with respect to its specified place in the total order.

Figure 3.1 illustrates the four key stages of the SCast protocol outlined above. Note, that for any $d \in Tx_i.dst$, that is not $Tx_i.c$, only step 4 of the protocol is required. The content and

significance of the operations $req(Tx_i)$, $rsp(Tx_i)$ and $mcast(Tx_i)$ are discussed in section 3.4.3.

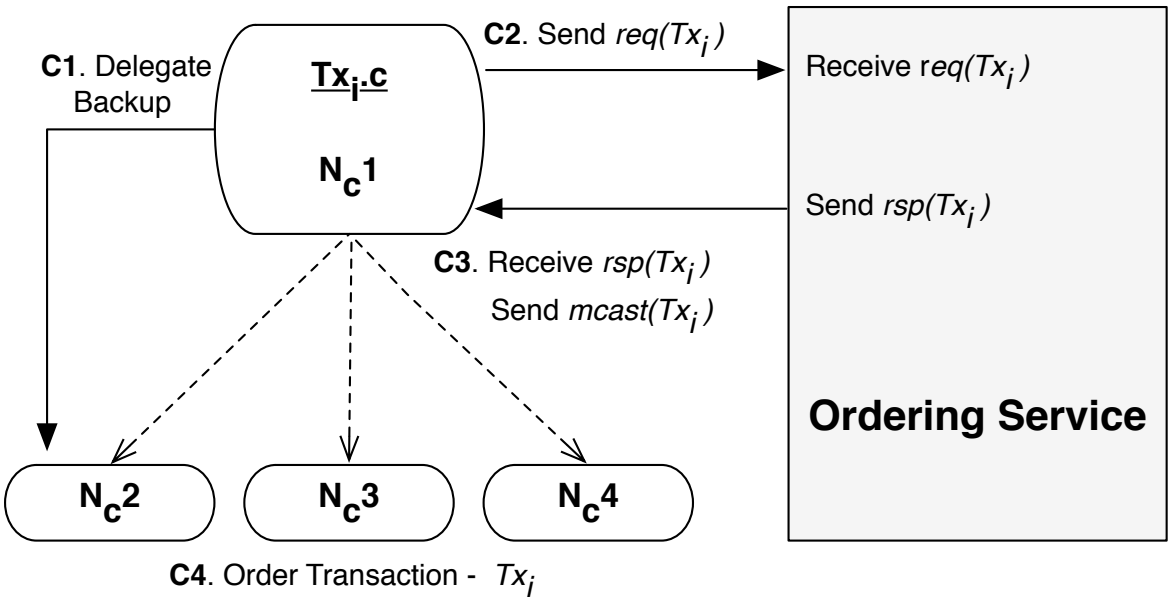

Fig. 3.1: SCast Client Interaction Diagram

Service Nodes

Stage 1 of the service protocol starts when an ordering request, $req(Tx_i)$, is received by an s-node, N_s1. It is anticipated that each s-node will handle many requests simultaneously, however for the sake of brevity our explanations assume that the service is only handling a single request at a given time. The key stages of the SCast protocol from the perspective of N_s1 are detailed below:

S1 Receive Request - Receive client request $req(Tx_i)$.

S2 Send Atomic Broadcast - Process $req(Tx_i)$ and *abcast* it to all s-nodes in the service, $abcast(Tx_i)$.

S3 Update Ordering Data - Deliver $abcast(Tx_i)$ locally and update the stored ordering data to include Tx_i.

S4 Return Ordering - Sends a response message, $rsp(Tx_i)$, to $Tx_i.c$, which contains all of the data required by c-nodes to order Tx_i.

Figure 3.2 illustrates the four key stages of the SCast protocol outlined above.

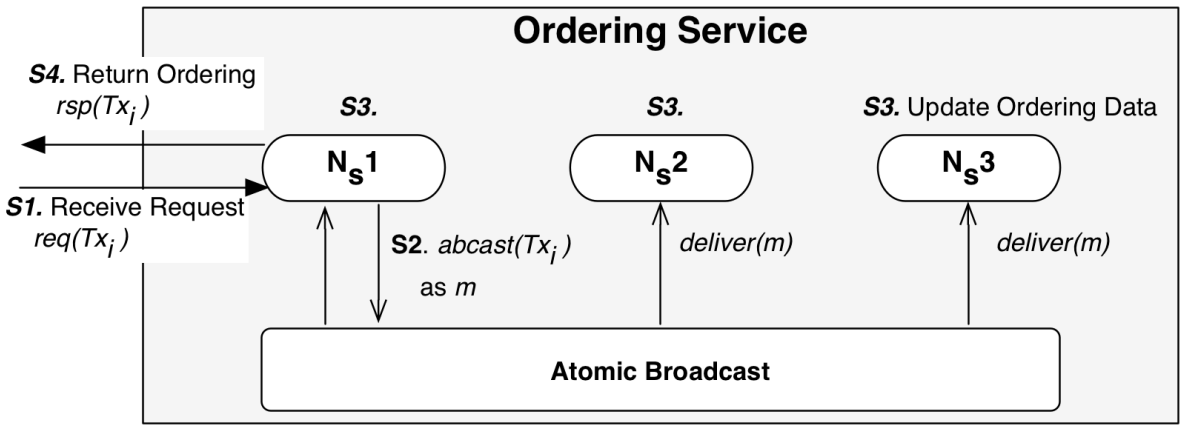

Fig. 3.2: SCast Service Interactions Diagram

3.4.2 Atomic Multicast Guarantees

The SCast protocol is a deterministic *amcast* protocol, therefore for all *amcast*s the protocol ensures that guarantees G1-G4, as stated in section 2.2, are met.

3.4.3 Protocol Details

In this section we explore the inner workings of each stage of the SCast protocol detailed in 3.4.1. We describe each stage in the order in which they are executed by the protocol, and assume a single transaction Tx_i is attempting to *amcast* its $prepare()$ message to all $Tx_i.dst$. For the sake of clarity, each stage of the protocol utilises the same numbering and naming scheme as section 3.4.1. Furthermore, all stages which are enacted by *c*-nodes are distinguished by an offset background and vertical bar in the left margin.

1. **C1 - Delegate Backup Coordinator**

   For the purpose of crash-tolerance, the first stage of the protocol is for $Tx_i.c$ to create a backup coordinator; where $Tx_i.c$ denotes the original coordinator for transaction Tx_i. $Tx_i.c$ selects any destination, d, $d \in Tx_i.dst$, and sends it the payload of the message that is to be multicast to $Tx_i.dst$ ($prepare(Tx_i)$). $Tx_i.c$ then waits for an acknowledgement of receipt from d before proceeding to stage 2 of the protocol. If an acknowledgement is received, d is designated as the backup coordinator $Tx_i.bc$. However, if an acknowledgement is not received within a configurable amount of

time, then $Tx_i.c$ simply selects another destination d' from $Tx_i.dst$ and restarts the process.

The purpose of creating a backup transaction coordinator is to allow for the possibility that $Tx_i.c$ may crash during the multicast process. In the event of $Tx_i.c$ crashing, the group membership service for c-nodes detects the crash and the backup coordinator assumes responsibility for multicasting $prepare(Tx_i)$ to all $d \in Tx_i.dst - \{Tx_i.c\}$. In order to accommodate such occurrences, we denote the currently active transaction coordinator as $Tx_i.\tilde{c}$, the original coordinator as $Tx_i.c$ and the backup as $Tx_i.bc$; $Tx_i.\tilde{c} = Tx_i.c$ if $Tx_i.c$ does not crash or $Tx_i.\tilde{c} = Tx_i.bc$, otherwise.

When $Tx_i.\tilde{c}$ takes over from the crashed $Tx_i.c$, it completely restarts the multicast process and selects another node from $Tx_i.dst$ to be a backup coordinator. In the unlikely event that the original coordinator and multiple backup coordinators crash, it is possible that there wont be a member of $Tx_i.dst$ left to utilise as a backup coordinator. In which case a backup will not be created, as $Tx_i.\tilde{c}$ will be the last destination alive in $Tx_i.dst$, therefore if $Tx_i.\tilde{c}$ crashes the transaction is aborted by default.

2. **C2 - Request Ordering**

   Once a backup coordinator has been established, it is possible for the transaction coordinator to request a multicast ordering from the service. *Amcast*s are initiated by the $Tx_i.c$ randomly selecting a s-node, N_s, from the *ordering service* and sending an *amcast* request to that node - $req(Tx_i) -> N_s$; where $req(Tx_i)$ contains $Tx_i.dst$, $Tx_i.\tilde{c}$ and the unique id of the transaction, $Tx_i.id$. Hence, $req(Tx_i) = \{Tx_i.id, Tx_i.dst, Tx_i.\tilde{c}\}$.

   The content of the $prepare(Tx_i)$ message is not sent to N_s as $Tx_i.c$ only requires a global order for Tx_i, therefore the contents of the message to be *amcast* to $Tx_i.dst$ is irrelevant and including it in the payload would increase bandwidth usage. However, $req(Tx_i)$ must include $Tx_i.dst$ as this is the destination set of the *amcast* that $Tx_i.c$ is trying to send, and the *ordering service* needs this data to ensure that requirements CR1, CR2 and CR3 are satisfied.

Wait for a response from the ordering service ...

If $Tx_i.\tilde{c}$ does not receive an ordering from the *ordering service* after a specified timeout, then a different *s*-node, N'_s, is selected and the ordering request is resent - $req(Tx_i) -> N'_s$.

3. **S1 - Receive Request**

   Upon receiving $req(Tx_i)$, an *s*-node places the request in its *Abcast Request Pool* (ARP), which holds client requests until they are *abcast*. If an *s*-node's ARP becomes full, *amcast* requests are rejected and a *reject* response is sent to $Tx_i.\tilde{c}$. If $Tx_i.\tilde{c}$ receives a *reject* response from all *s*-nodes, it can either abort Tx_i or resend the *amcast* request to another *s*-node after a configurable amount of time [3]. The ARP is necessary to ensure that if the *ordering service* starts to become overloaded by client requests there is a 'feedback' mechanism that makes *c*-nodes aware of the service's current limitations and allows clients to restrict user operations if necessary.

4. **S2 - Send Atomic Broadcast**

   A single thread, called the *send* thread, is utilised for retrieving requests from the ARP and *abcast*ing them to all *s*-nodes for ordering. The *send* thread retrieves an ordering request from the ARP and sends an *abcast*, *m*, to all *s*-nodes containing the request. Requests are retrieved from the ARP in the order that they were originally received (FIFO), however if no requests exist in the ARP then the *send* thread waits for the pool to become non-empty before resuming *abcast*ing.

   All *abcast* messages *m*, sent by an *s*-node, include the fields associated with Tx_i which are sent as part of $req(Tx_i)$ *i.e.*

$$m.tx\_id = Tx_i.id$$

$$m.tx\_\tilde{c} = Tx_i.\tilde{c}$$

$$m.dst = Tx_i.dst$$

   Furthermore, *m* also includes the the address of the sending *s*-node and a sequence number, as *m.snid* and *m.seq*#, respectively. Where *m.seq*# is an integer which increases by

[3] This cycle will not continue indefinitely, as eventually the transaction will timeout and abort.

one every time an *abcast* is sent from this *s*-node.

5. **S3 - Update Ordering Data**

   When an *s*-node, N_s, delivers m via *abcast*, it checks its records to see if it has already processed a client request $req(Tx_i)$ using the globally unique $m.tx\_id = Tx_i.id$. If not, m is accepted and processed via stages *a-c* detailed below; otherwise, the process detailed in section 3.4.4 is initiated[4].

   a. Establishing a Total Order

   Upon delivering and accepting m, N_s assigns a global order to m and the associated $m.tx\_id$, which is represented as $m.order$.

$$m.order = ts \oplus m.seq\# \oplus m.snid$$

   Where $\oplus$ is the append operator and $m.ts$ is the final timestamp provided by the underlying *abcast* protocol that is utilised between *s*-nodes. As ts is generated by the underlying *abcast* protocol and $m.seq\# \oplus m.snid$ is specified before broadcast, it is guaranteed that all *s*-nodes will produce the same $m.ts$.

   b. Defining Total Order

   The ordering of *amcast* messages, and hence transaction ordering, is defined as follows. We denote that an *amcast* message m precedes another *amcast* m' in the global total order, *i.e.* $m.order < m'.order$ as $m \prec m'$ and define this relationship as:

$$\begin{aligned} m \prec m' \implies & m.ts < m.ts \\ & \vee \left(m.ts = m'.ts \wedge m.seq\# < m'.seq\#\right) \\ & \vee \left(m.ts = m'.ts \wedge m.seq\# = m'.seq\# \wedge ranking(m.snid) < ranking(m'.snid)\right) \end{aligned}$$

   Where $ranking()$ is a deterministic function that returns an integer value for a given $snid$.

[4]Multiple client requests for Tx_i can be *abcast* between *s*-nodes as a consequence of $Tx_i.c$ crashing, or timing out, as these events result in $req(Tx_i)$ being sent multiple times.

Furthermore, we state that m, *immediately* precedes m' in the total order if the following statements are true:

(i) $m \prec m'$

(ii) There is no $m'' : m \prec m'' \prec m'$

and we denote this relationship as $m \lll m'$.

As SCast is an *amcast* protocol, it is possible for each m to have a different destination set. Consequently, a message's ordering also needs to be defined relative to the other messages that have been received by each $d \in m.dst$. Therefore, let *amcast* $\tilde{m}$ precede m for any destination d, if

$$d \in \tilde{m}.dst \cap m.dst \wedge \tilde{m} \prec m$$

and we denote this relationship as $\tilde{m} \prec_d m$.

Note: $\tilde{m} \prec_d m \implies \tilde{m} \prec m$, however $\tilde{m} \prec m \not\Longrightarrow \tilde{m} \prec_d m$. For example, consider m_i and m_j, if $m_i \prec m_j$ but $m_i.dst \cap m_j.dst = \{\}$, then it is not possible for m_i to precede m_j for any $d \in m_i.dst$.

Finally, we state that an *amcast*, $\tilde{m}$, *immediately* precedes m, with respect to destination d, if:

(i) $\tilde{m} \prec_d m$

(ii) There is no $m' : \tilde{m} \prec_d m' \prec_d m$

and we denote this relationship as $\tilde{m} \lll_d m$.

c. Maintaining the Total Order

Once an order has been associated with Tx_i and m, it is necessary for the s-node to calculate $m.history[]$. The purpose of $m.history[]$ is to ensure that *amcast* guarantee G4, is maintained by all $d \in m.dst$ upon delivering m; as detailed in stage 8 of the protocol.

We define $m.history[]$ as an associative array that utilises the address of each $d \in m.dst$ as an index and stores the $\tilde{m}.order$ value of $\tilde{m}$ that satisfies $\tilde{m} \lessdot_d m$. If no $\tilde{m} \lessdot_d m$ exists, then a *null* value is stored in its place. This is formalised below:

$$\forall \quad d \in m.dst : m.history[d] = \tilde{m}.order$$
$$\text{where } \tilde{m} \lessdot_d m$$

and Algorithm 1 presents the pseudocode for populating $m.history$ for an *amcast* m.

Algorithm 1 Compute Message History

```
1: for all d ∈ m.dst do
2:    if m̃ ≪_d m then
3:       m.history[d] ← m̃.order;
4:    else
5:       m.history[d] ← null;
6:    end if
7: end for
```

The $\tilde{m}$ values used to populate $m.history[]$ are retrieved from N_s's local ordering history, which is an associative array that utilises the same indexing scheme as $history[]$ and also stores past $order$ values. We refer to this array as $order\_history[]$. The purpose of $order\_history[]$ is to maintain a history of *amcast* orderings for all known c-nodes; where a known c-node is any node that has been involved in a prior transaction's destination set. Once N_s has populated $m.history[]$ for m, it is necessary for $m.order$ to be added to $order\_history[]$ for all $d \in m.dst$, as $m.order$ is now the latest *amcast* involving each d. The Algorithm for populating $order\_history[]$ is presented in Algorithm 2.

Algorithm 2 Compute $order\_history[]$

```
1: for all d ∈ m.dst do
2:    if No order_history[d] exists then
3:       create index order_history[d];
4:    end if
5:    order_history[d] ← m.order
6: end for
```

For example, consider two ordering requests that have been *abcast* as m_i and m_j, with destination sets equal to N_c1, N_c2 and N_c1, N_c3, respectively. Assume, that m_i has already

been delivered by an s-node N_s, and m_j is now being processed by N_s; hence $m_i \prec m_j$. The calculated history for m_j, $m_j.history[]$, is shown below in Figure 3.3.

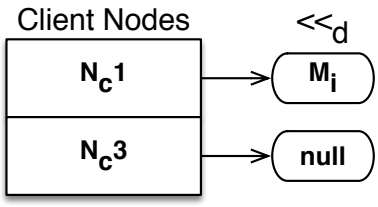

Fig. 3.3: Message History Array

Once N_s has calculated $m_j.history[]$, it is necessary for m_j to be added to the $order\_history[]$. The resulting $order\_history[]$ is shown below in Figure 3.4.

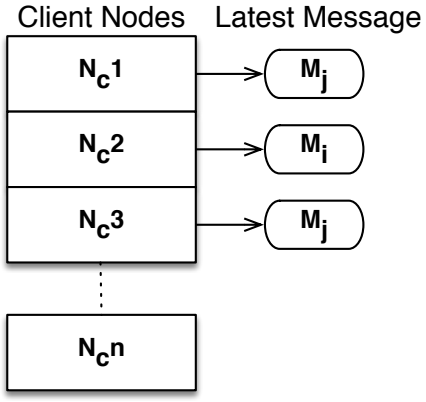

Fig. 3.4: Order History Array

6. **S4 - Return Ordering**

   At this stage the local s-node, N_s, has completed the ordering process for m, therefore it is necessary for m to be sent to $Tx_i.\tilde{c}$ as a $rsp(Tx_i)$ message in order for the *amcast* process to continue; where $rsp(Tx_i)$ consists of $m.tx_id$, $m.tx\_\tilde{c}$, $m.order$, $m.dst$ and $m.history[]$. To ensure that only one s-node responds to $Tx_i.\tilde{c}$, the $rsp(Tx_i)$ message is only sent by the s-node whose $snid = m.snid$.

   Crash-Tolerance: Each s-node maintains a short history of $rsp()$ messages as past message orderings may be requested by c-nodes in certain circumstances [5].

[5] As described in section 3.4.4.

7. **C3 - Receive Ordering and Multicast Transaction**

   When $Tx_i.\tilde{c}$ receives $rsp(Tx_i)$ from N_s, it appends the ordering information to the original $prepare(Tx_i)$ message and multicasts $prepare(Tx_i)$ as $mcast(Tx_i)$ to all $Tx_i.dst$ including itself; where a multicast consists of $mcast(Tx_i)$ being unicast to all $d \in Tx_i.dst$.

   Crash-Tolerance: For the purposes of crash-tolerance, $mcast(Tx_i)$ must be unicast last to Tx_i's backup coordinator when multicasting $mcast(Tx_i)$ to all $d \in Tx_i.dst$. This ensures that if the backup coordinator receives $mcast(Tx_i)$, then at least one copy of $mcast(Tx_i)$ will have been sent to all $d \in Tx_i.dst$.

8. **C4 - Order Transaction**

   Upon receiving $mcast(Tx_i)$, a c-node destination, d, stores this $mcast()$ as m in the *amcast_wait_queue* (AWQ). This priority queue stores all of the *amcast*s received by this node until they are delivered to the application. Messages are prioritised in the AWQ based upon their $m.order$, *e.g.* $m_1 \lessdot_d m_2 \lessdot_d m_3$ as shown in Figure 3.5, and can only be delivered to the application when $\tilde{m}$, $\tilde{m} \lessdot_d m$, specified in $m.history[d]$, has been delivered. It is necessary for the delivery of m to be delayed until after $\tilde{m}$ to ensure that m's, and subsequent *amcast*s involving this node, total order (guarantee G4) is maintained.

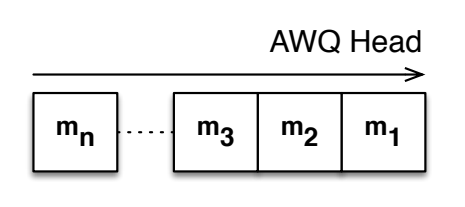

Fig. 3.5: Amcast Wait Queue

   Messages are delivered to the higher-level application via the primitive $am\_deliver()$, which extracts the payload ($prepare(Tx_i)$) of the *amcast* message and sends it up the network stack. A c-node considers the *amcast*ing of Tx_i to be complete when it has called $am\_deliver()$ for $mcast(Tx_i)$, hence the *amcast* is complete when the payload of $mcast(Tx_i)$ has been delivered up the stack.

Note: In order to preserve the total order dictated by the ordering service, a single thread must $am\_deliver()$ messages to the application; we refer to this thread as the *delivery* thread. If multiple threads were utilised, the execution of two *amcast* messages could overlap, resulting in the total order property of the messages being invalidated. Utilising a single thread for message delivery is not unique to SCast, but is instead a limitation of all *amcast* protocols and can be seen in the TOA protocol.

Crash-Tolerance: All c-nodes must store a copy of a delivered $mcast(Tx)$ message, if they were the designated backup coordinator for that Tx. With each c-node maintaining a finite list of such $mcast(Tx)$ messages; we refer to this list as $backup\_history$.

Algorithm 3 presents the pseudocode for processing the AWQ. The variable $last\_amcast$ stores the $m.order$ of the last *amcast* message that was dequeued from the AWQ; hence this is the last *amcast* to have be delivered by d. Furthermore, the array $last\_delivered[s]$ stores the $m.order$ of the last *amcast* message which was processed by the s-node s.

Algorithm 3 Amcast Wait Queue

```
1:  while AWQ is non-empty do
2:      h ← peek(AWQ);
3:      h.snid ← h.order.snid;
4:      h̃.order ← h.history[d].order;
5:      h̃.snid ← h̃.order.snid;
6:      if h̃.order ⪯_d last_delivered[h̃.snid] then
7:          if last_amcast ≺_d h.order then
8:              am_deliver(h);
9:              last_delivered[h.snid] ← h.order;
10:             last_amcast ← h.order;
11:             dequeue(AWQ);
12:             if d is Backup Coordinator for h then
13:                 backup_history.add(h);
14:             end if
15:         end if
16:     end if
17: end while
```

3.4.4 Fault-Tolerance: Node Crashes

Fault-tolerance in SCast must consider the consequences of both crashed *c*-nodes and *s*-nodes. Here we explore the consequences of both *c*-node and *s*-node crashes during various stages of a SCast *amcast*. For the sake of simplicity, we only consider node crashes from the perspective of a single transaction, however it is worth noting that each *c*-node would typically have multiple transactions executing concurrently. In what follows, we consider failure instances with regard to the four steps (C1-C4) in Figure 3.1.

Client Node Crash

Local Tx Execution

If a *c*-node, $Tx_i.c$, crashes during or directly after the local execution of a transaction,

Tx_i, then no action needs to be taken as no interactions with other *c*-nodes or *s*-nodes have occurred.

During C1

If $Tx_i.c$ crashes during the creation of a backup coordinator node, $Tx_i.bc$, then two scenarios are possible:

i $Tx_i.bc$ never receives the $prepare(Tx_i)$ message, in which case the transaction can only be aborted as its contents have been lost.

ii $Tx_i.bc$ successfully receives the $prepare(Tx_i)$ message and attempts to acknowledge $Tx_i.c$; who will never receive $Tx_i.bc$'s acknowledgment as it has crashed. In which case, the GM service will detect that $Tx_i.c$ has crashed and issue a new view to the network. Upon receiving this view, $Tx_i.bc$ deduces that $Tx_i.c$ has crashed and becomes the new active coordinator, $Tx_i.\tilde{c}$, for Tx_i and restarts the multicast process at stage C1.

During C2

If $Tx_i.c$ crashes before or during the sending of a request *req* to the ordering service, then $Tx_i.bc$ simply restarts the multicast process at stage C1 when the GM service recognises that $Tx_i.c$ has crashed.

It is possible that the *s*-node that *req* was sent to, N_s, will still receive *req*, in which case *req* will be processed as normal by the *s*-node. When *bc* sends a new ordering request to the service for Tx_i, req', this request is also processed. The ordering service accepts the request which is delivered first by the *abcast* protocol and sends a response to the $Tx_i.\tilde{c}$ specified in the accepted request. Assuming that both coordinators send *req* and req' to N_s: If the *s*-nodes deliver *req* first in the total order, so that $req \prec req'$, then a response is sent to $Tx_i.c$ as N_s is unaware that this *c*-node has crashed. However, when N_s delivers req' it deduces that both *req* and req' concern Tx_i, and that req' has only been issued because $Tx_i.c$ has crashed. Therefore N_s resends the original $rsp(Tx_i)$ message associated with *req* to the $Tx_i.\tilde{c}$ specified in req' and the *abcast* message of req' is discarded without updating $order\_history$.

Similarly, it is possible that req' is sent to a different *s*-node than *req*, N'_s, in which case

N'_s will also resend the original $rsp(Tx_i)$ message, as N'_s must have received both *req* and *req*′ as per the guarantees of *abcast*.

In addition to the various scenarios described above, it is also possible that *req* was never received, in which case the *s*-node that received *req*′ will send a response message to the coordinator of *req*′ as if it were a normal request.

During C3

If $Tx_i.c$ crashes before receiving a response from the ordering service, then $Tx_i.bc$ takes over and restarts the multicast process. When $Tx_i.bc$'s request is received by an *s*-node, the *s*-node checks its recent history of processed requests and returns the ordering response message associated with Tx_i.

The size of the recent history stored by *s*-nodes should be configurable to allow for varying levels of resilience. This is because as the size of the past history increases, the chances of a prior response message being discarded decreases. Thus a larger record provides a greater level of crash-tolerance, but at the expense of utilising more system resources.

During C4

Assuming that $Tx_i.c$ crashes at this stage, there are three distinct scenarios that can occur:

i $Tx_i.bc$ has not received $mcast(Tx_i)$.

ii $Tx_i.bc$ has received but not yet delivered $mcast(Tx_i)$.

iii $Tx_i.bc$ has delivered $mcast(Tx_i)$.

For all of the above scenarios it is not possible for $Tx_i.bc$ to determine whether any $d \in Tx_i.dst$ has received $mcast(Tx_i)$ without additional communication between nodes. Therefore, in all three scenarios $Tx_i.bc$ pessimistically assumes that at least one d has not received $mcast(Tx_i)$. Hence, $Tx_i.bc$ must send its own multicast of $mcast(Tx_i)$ to all d.

The corresponding recovery mechanism for each of the above scenarios is presented below; here we assume that $Tx_i.bc$ has discovered, via the GM service, that $Tx_i.c$ has crashed and we refer to this crashed *c*-node as *c*.

i $Tx_i.bc$ has not received $mcast(Tx_i)$, therefore it is necessary for the protocol to be restarted.

ii $Tx_i.bc$ has already received $mcast(Tx_i)$, therefore it must designate a new $Tx_i.bc$ before multicasting $mcast(Tx_i)$ to all $d \in Tx_i.dst$.

iii $Tx_i.bc$ has already delivered $mcast(Tx_i)$, therefore it must check its entries in *backup_history* and multicast all stored $mcast()$ messages whose active coordinator was c.

Service Node Crash

Stage S1-S4

If an s-node, N_s crashes after $Tx_i.c$ has sent an ordering request, req, to N_s, then $Tx_i.c$ will timeout waiting for a response for req and will resend the request to another s-node as req'. It is possible for req to have been *abcast* to other s-nodes before N_s crashed, in which case, the other s-nodes will deliver and process req as a normal request. As N_s has crashed, a $rsp()$ message will not be sent to $Tx_i.c$, due to no correct s-node satisfying the condition $snid = m.snid$. However, as all other s-nodes in the ordering service have delivered req, a $rsp()$ message will exist at all correct s-nodes. Therefore, when req' is sent to a correct s-node, the original $rsp()$ message which was associated with req will be returned to $Tx_i.c$.

3.4.5 Fault Tolerance: Split Brain

Split brain refers to a situation whereby the current view of a group of processes has been partitioned into two or more views, which is usually caused by one or more failures occurring at the underlying network layer. Typically, these views will consist of disjoint sets of processes, however it is possible for overlapping to occur between multiple views. Eric Brewer's seminal CAP theorem, states that it is impossible for a system to provide Consistency, Availability and Partition Tolerance simultaneously [9, 10, 28]. Therefore, when designing a solution for handling split brain scenarios, which is a partition by definition, it is necessary for either availability or consistency of part of the system to be compromised.

Handling split brain partitions across a cluster of c-nodes in which SCast operates is ultimately the responsibility of the applications using SCast for *amcast*s. For example, in the case

of Infinispan, if a cluster of *c*-nodes is partitioned then it is the responsibility of Infinispan to determine whether consistency or availability should be preserved. However, as a Infinispan cluster will be dependent on SCast and its *ordering service*, it is necessary for such a service to provide a strategy for handling partitions that occur within the service itself.

Our solution for handling partitions within an SCast ordering service is to utilise a majority partition scheme. When a network partition occurs, the *s*-nodes whose new network view is a majority of the previous view continues to accept client requests and operate as an *ordering service*. Whereas the *s*-nodes who are now in the minority partition sacrifice availability by rejecting future client requests until the juncture of the two partitions. The *s*-nodes in the minority partition reject client requests in order to allow for the consistency of the system to be readily resolved when the two partitions are rejoined. For example, if the availability of the minority partition was not sacrificed, the merging of state required when the two partitions are rejoined would not be trivial, with the predecessor data and active client requests of each partition having to be fused in a way that does not compromise *amcast* guarantees G1-G4. Whereas, when only one partition remains active when the network is divided, it is possible for the *s*-nodes in the minority partition to clone the state of an *s*-node from the majority partition and start accepting client requests again.

The majority partition scheme detailed above works as expected when $|s\text{-nodes}|$ is an odd number, however if it is an even number, then it is possible for the *ordering service* to be partitioned so that no majority partition exists, in which case availability of the entire ordering service must be sacrificed in order to maintain consistency. To reduce the chances of such a scenario occurring it is possible for an additional 'watcher' node to be utilised when $|s\text{-nodes}|$ is even. A watcher node does not participate in the SCast protocol, rather it is used purely for tie-breaking between the views of two *s*-node partitions that would otherwise be equal. Utilising a watcher node reduces the chances of no majority partition existing, however it is still possible as multiple partitions can occur, in which case availability will be sacrificed in the same manner as when a watcher node is not utilised.

3.5 Message Bundling

When utilising AmaaS it is possible for all *amcast* requests received from *c*-nodes to be bundled into a single *abcast* (between *s*-nodes) at a receiving *s*-node, regardless of their destination set. This is because *s*-nodes are only required to send *abcast*s to other *s*-nodes in order for a consensus on transaction ordering to be reached, therefore the destination set for each *abcast* is the same for all *c*-node requests. The ability to bundle multiple *amcast* requests into a single *abcast* reduces the number of times that consensus needs to be reached between all *s*-nodes. Thus further reducing the number of $N->1$ communication steps required, with the total number of *abcast*s reduced by $|bundle|$; where $bundle$ is the number of *amcast* requests from *c*-nodes that are sent as a single *abcast*. As a result of this optimisation, network traffic is significantly reduced when requests are frequent, resulting in the capacity and scalability of an *AmaaS* service increasing. Conversely, message bundling does not compromise performance when the number of service requests is low, as bundling does not require any intensive computation or additional communication steps.

In the case of SCast, message bundling is implemented as follows: The *send* thread retrieves ordering requests from the ARP in their arrival order, and bundles them into a single message bundle *mb*, with the first message being stored at index 0 and so on. This message bundle is then *abcast* to all *s*-nodes. A configurable upper limit can be placed on the maximum size of a bundle message. [6] If this upper limit is reached and the ARP still has available requests, then the *send* thread will start processing the next message bundle, mb', once *mb* has been *abcast*.

Upon *abcast* delivering *mb*, each *s*-node must *unbundle mb* and process each individual *c*-node request in the same manner as if the request had been *abcast* as a single request. A consequence of multiple ordering requests being bundled in to a single *abcast* is that the timestamps utilised in SCast to uniquely order transactions are no longer valid, due to multiple transactions being associated with a single *abcast*, and hence a single ts. Therefore, in order to uniquely order an transaction Tx_i, within *mb*, we redefine the unique order of each request as $m.order = ts \oplus m.seq\# \oplus m.snid \oplus sequence\ number$ of $req(Tx_i)$ within *mb*.

[6] The maximum size could be specified in terms of bytes or the number of messages to be bundled.

3.6 A New Atomic Broadcast Solution is Required

Existing *abcast* and *amcast* solutions are of two types (§ 2.2); quorum based and GM based. GM based protocols are typically leaderless, which allows such protocols to provide higher levels of throughput than quorum based protocols when node crashes are absent [7], however when crashes do occur GM protocols block indefinitely until a new GM view is propagated across the cluster. This blocking behaviour is acceptable when such protocols are utilised in traditional P2P environments like Infinispan, as it is presumed that the blocking will only occur at a small subset of nodes in the cluster. In which case system *liveness* is maintained by the majority of nodes in the cluster. However in the AmaaS model, if a *s*-node utilises a GM protocol for *abcast*ing requests amongst all *s*-nodes and a single *s*-node crashes, all *s*-nodes will block, resulting in no client requests being satisfied. This means that, not only are the *s*-nodes participating in the *abcast* blocked, but as a consequence of this blocking, so to are all of the *c*-nodes utilising the service. Therefore the entire system's *liveness* is lost until the GM protocol is able to detect the *s*-node crash and unblock the ordering service; hence requirement SR2 is undermined.

Alternatively, a quorum based protocol, such as those detailed in 2.3, can be utilised between *s*-nodes. Such protocols perform worse than GM protocols in the absence of node failures, however they only block mildly when a leader node crashes or is falsely suspected of crashing. Both Zookeeper and Chubby coordination services utilise a quorum based *abcast* protocol for state machine replication; with each service utilising a single *master* node to coordinate all *abcast*s. Consequently, the write throughput of each of these services is limited by the maximum throughput capabilities of the designated *master* node. This is significant for AmaaS, as each ordering request sent to the ordering service requires a single *abcast*. Therefore, as the number of concurrently executing transactions increases, the throughput of an ordering service utilising a quorum based protocol does not scale with demand.

In order to maximise the effectiveness of the AmaaS system model, a new *abcast* protocol is required. This protocol must provide non-blocking message delivery in the presence of node failures, whilst allowing for low-latency, high-throughput *abcast*s in their absence.

[7] Higher levels of throughput and lower latency atomic broadcasts, are possible with GM protocols as *abcast*s can originate from any participating node. This is in contrast to quorum based protocols which typically require a single master node to coordinate message delivery amongst participants, ultimately resulting in this master node becoming a performance bottleneck for said protocol.

3.7 Summary

This chapter presented AmaaS - a new model for *amcast* protocols that utilises a dedicated set of nodes to provide *amcast* as a service to distributed transactional systems. We then presented a new protocol SCast that provides fault-tolerant *amcast*ing in such an environment. Lastly, we outlined the shortcomings of existing *abcast* solutions and the need for a new protocol in order for the AmaaS approach to be fully realised.

Chapter 4

ABcast

In this chapter we introduce a hybrid *abcast* protocol, called ABcast, which provides non-blocking message delivery in the presence of node failures and low-latency message delivery in their absence. This protocol was designed for use amongst $N_s1 \ldots N_sn$ nodes within the AmaaS system model.

The remainder of this chapter is structured as follows: First we introduce the rationale behind utilising a Hybrid protocol and our design approach for ABcast, before detailing the protocol's requirements and assumptions. This is followed by an in-depth look at the components required by ABcast, and how they have been implemented. We then explore the two protocols used to create the hybrid solution in detail, outlining each protocol's delivery and rejection criteria for *abcast* messages. Finally we describe a new flow-control protocol, AFC, which has been designed specifically for use with ABcast.

4.1 Rationale

In the previous chapter we introduce AmaaS, a model that aims to increase the transactional throughput of distributed in-memory transactional systems. This model depends on an *abcast* protocol to maintain the replicated state between the service nodes which provide multicast ordering to client nodes; with each multicast request requiring a state change between service nodes. For an AmaaS service to be viable it is vital that it provides low-latency responses to the requesting client nodes, as well as being able to handle an increasing number of client requests as the transactional system scales. Furthermore, it is essential that such a service maintains

high-availability, even in the presence of node failures, as an entire cluster of client nodes are dependent on the service. Thus, it is essential that the underlying *abcast* protocol utilised by the service can provide both non-blocking and low-latency message delivery in order to satisfy the clients requirements of highly-available and low-latency requests respectively.

4.1.1 Existing Atomic Broadcast Solutions

The FLP impossibility [27] dictates that in an asynchronous environment *abcast* protocols must either admit blocking to meet its atomic guarantees or permit a likelihood of its termination guarantees not being met. As previously stated, known blocking protocols are of two types: GM dependent and Quorum based, both of which admit blocking in order to remain atomic. The quorum based protocols block mildly due to false/valid suspicions of the leader node and GM protocols block severely but only in the presence of slow or crashed nodes.

Quorum based protocols provide non-blocking message delivery, however they only provide low levels of throughput as they are typically leader based, which ultimately limits the scalability of the system. Furthermore, there is also a non-zero probability that such protocols get stuck indefinitely in a cycle of leader elections after the previous leader node is falsely suspected of crashing[1]. On the contrary, leaderless GM based protocols typically allow for increased levels of throughput, however the blocking inherent in GM protocols would critically undermine an AmaaS service's availability in the event of a service node crash.

From the disadvantages stated above, it is clear that the aforementioned protocols are not ideal when utilised within AmaaS. Therefore it is necessary for a non-blocking approach to be utilised, that allows for the possibility that guarantees G1-G4 (§ 2.2.2) will not always be met in order to overcome the limitations of the FLP impossibility. Utilising probabilistic guarantees on message delivery is an established technique for increasing the scalability of network multicasting systems[43], which has also been applied to *abcast* protocols.

Felber *et al.* [25] propose an *abcast* protocol, PABCast, that provides probabilistic guarantees on both message *safety* and *liveness*. With some non-zero probability, it is possible for only a subset of the destination set to receive a broadcast, or for all destinations to deliver the broadcast but in an inconsistent ordering. The aim of the PABCast protocol is to provide increased

[1] This is unlikely to occur in practice with adaptive or sufficiently long timeouts used for crash-suspicion.

scalability for atomic broadcasts across large numbers of destinations, not a small subset of nodes as required by AmaaS. As such the protocol does not consider throughput a primary concern. The protocol uses *rounds* to regulate when a node can initiate a broadcast and a node cannot initiate a new broadcast until all broadcasts in the current round have been delivered locally. Ultimately this protocol structure limits a sending node to a single broadcast, which clearly limits the protocol's throughput capabilities.

In the literature, the performance of PABCast is evaluated using a simulation that focuses on the scalability of the system in terms of message cost as well as the likelihood of a broadcast's *safety* and *liveness* being violated due to the probabilistic guarantees not being met. The performance evaluation presented in the paper does not consider the throughput or latency of the PABCast protocol, and the protocol is only evaluated using a simulation so it is not possible to ascertain how such a protocol will function in a live asynchronous system. It is our view that PABCast is not suitable for use in the *AmaaS* system model.

4.1.2 Existing Hybrid Solution

Bezerra *et al.*[7] propose a hybrid *amcast* protocol (Optimistic Atomic Multicast) that combines a deterministic consensus protocol (Paxos), with a probabilistic protocol in order to reduce *amcast* latency. The probabilistic protocol is utilised for faster message delivery (*optimisitic*), whereas Paxos is utilised to ensure correctness (determinism). Consequently, when the probabilistic protocol delivers messages outside of the total order, it is followed up by a second, correctly-ordered delivery, which enables applications to take retrospective action to correct the ordering mistakes of the *optimistic* protocol. Hence, this approach requires both the deterministic and probabilistic protocols to finish executing in order for an *amcast* to fulfil all of its guarantees.

Informally and assuming that there exists only a single group of destinations (*abcast*), Optimistic Atomic Multicast works as follows:

Optimistic Delivery

1 A *abcast m*, is timestamped with the sending processes, p, clock time and broadcast to all $m.dst$.

2 Upon receiving m, another process p', computes a wait period specific to the sender p, denoted as $w(p)$. Where $w(p)$ is a delay calculated utilising the estimated clock difference between p' and p, combined with the largest observed network latency between p and p'.

3 p' must then wait until its local clock reads $m.ts + w(p)$ before optimistically delivering m ($opt\_del(m)$).

Note: A G4 violation occurs, if $m.ts$ is received at p' after m' has already been $opt\_del(m')$ and $m.ts < m'.ts$.

Deterministic Delivery

4 After $opt\_del(m)$ has been completed, it is necessary for a Paxos instance to propose m to all $d \in m.dst$ to decide upon a deterministic ordering for m.

5 Once a consensus has been reached by Paxos for m, it is necessary for m to be conservatively delivered to the application in its established total order ($con\_del(m)$).

Application Level

6 The application processes $opt\_del(m)$ messages as if the total order is correct. However, if m is later delivered in a different total order via $con\_del(m)$, then the application knows that G4 was violated by $opt\_del(m)$ and it must take retrospective action.

In the context of *abcast* protocols, a key disadvantage of this approach is that the deterministic protocol must always be executed in its entirety to ensure that incorrect message orderings are eventually detected. For a message m, the delay between a G4 violation occurring ($opt\_del(m)$) and the Paxos execution completing ($con\_del(m)$) could be very large, as $con\_del(m)$ cannot occur until at least $m.ts + w(p) + px$; where px is the time taken for Paxos to reach a consensus for m. Furthermore, as Paxos is a leader based protocol, all m must be processed by the leader node for a consensus to be achieved, therefore the leader node will become a bottleneck for $con\_del(m)$ under high levels of load, hence the delay between $opt\_del(m)$ and $con\_del(m)$ could become even larger.

At the application layer, a key disadvantage of Optimistic Atomic Multicast, is that two independent delivery channels must be processed by the application at anyone time ($con\_del(m)$ and $opt\_del(m)$), with the application being responsible for both detecting and recovering from G4 violations by contrasting the output of both channels. Hence the application has to allocate additional resources for processing messages.

4.1.3 Our Approach

Our approach is to create a hybrid protocol that combines the leaderless GM-based protocol described in section 2.2.4, with a custom designed probabilistic *abcast* protocol that is also leaderless. The deterministic GM protocol is utilised in the absence of crashed/suspected nodes, however in their presence the probabilistic protocol is utilised in order to overcome the blocking inherent in GM based protocols. We refer to the probabilistic protocol as Aramis, and the deterministic protocol as Base, which when combined creates the hybrid Atomic Broadcast protocol - ABcast.

It should be noted that both Aramis and Base work in parallel and additional overhead is minimal as both protocols are leaderless in nature. That is, both protocols attempt to deliver each *abcast* in parallel; when the faster one succeeds in delivering a given *abcast*, the slower one's attempt on that *abcast* terminates. Consequently, a given *abcast* is never delivered more than once. Typically, Base succeeds when there are no crashes and Aramis succeeds when Base is blocked.

Aramis[18–20] is a non-blocking *abcast* protocol that guarantees uniform total order (G4 § 2.2) with a probability close to 1. Aramis utilises the probabilistic synchronous model (§ 2.1.3), in conjunction with closely synchronised clocks, to calculate a probabilistic upper bound on *abcast* delivery times; we refer to this upper bound as a message's delivery delay, Δ_m.

Aramis: An Informal Description

Upon receiving an *abcast* message, a destination node waits for the calculated delivery delay to expire before delivering the message to the application. If a message m does not reach one of its destination, say $N_s i$, before Δ_m, then it is possible for $N_s i$ to deliver a subsequent message m' if $\Delta_{m'}$ expires before m is received by $N_s i$. When such a scenario occurs the *abcast* guarantees

G4 will not be met and therefore the broadcast cannot be considered to be atomic.

A key advantage of the Aramis approach is that no message acknowledgements are required for a message to be delivered, instead it depends entirely on the calculated delivery delay Δ_m. Relying solely on Δ_m ensures that faulty nodes have no effect on the delivery of a message at correct nodes and it is therefore impossible for a message's delivery to become blocked. Furthermore, as no quorums or acknowledgements are required, it is possible for Aramis to tolerate at most $(n-1)$ destination crashes when n nodes are involved in an *abcast*.

The Aramis protocol was developed to be risk adverse, with all probabilistic calculations carried out pessimistically in order to ensure that Δ_m is rarely exceeded. Furthermore, Δ_m always assumes the worst case scenario will happen when the protocol is executing (e.g. the originator node crashes during every broadcast) to ensure that such situations are catered for. A consequence of this pessimism, is that the latency of a *abcast* message can be very large, typically 100-1000ms. Note that these potentially large latencies, though not desirable, do not undermine Aramis from offering high throughput.

Aramis and Base: An Informal Description

To counteract the large delivery latencies of Aramis it is necessary to operate a low-latency *abcast* protocol, Base, alongside Aramis. The Base protocol is a GM based deterministic protocol, similar to NewTop[23], that provides low-latency high throughput *abcast*s at the expense of blocking when node failures occur. In the context of an AmaaS ordering service, Base works as follows: A message's orginator, say $m.o = N_s i$, broadcasts m to every $N_s j$, which in turn broadcasts an $ack_j(m)$ to every node in the service. Once a s-node has received $ack_j(m)$ from all $N_s j$, $N_s j \neq m.o$, m becomes deliverable. Note that if one $N_s j$ crashes during the *abcast*ing of m, the delivery of m will be blocked if the crashed node had not sent $ack_j(m)$ before crashing and the protocol must wait for the GM service to detect the crash so that it can unblock m.

In order to hone the advantages of both protocols it was necessary to create the hybrid protocol ABcast, where an abcast m becomes deliverable either when Δ_m has elapsed (Aramis) or when $ack_j(m)$ is received from every $N_s j$ (Base). This approach provides the application with the low-latency of Base for the majority of message deliveries, whilst ensuring that a missing acknowledgement is not waited upon for more than Δ_m time.

In the event of a node failure the Base protocol has to wait for the GM service to detect a crash before message delivery becomes unblocked, however messages will be delivered by Aramis after Δ_m expires. Therefore, when node failures are present the ABcast protocol will always allow for a greater throughput of delivered messages than a traditional GM based protocol, assuming that Δ_m remains smaller than the time it takes for GM to detect a node failure and construct a new service. In the worse case, if the GM delay is smaller than Δ_m, then the Base protocol can simply unblock its message buffer and continue to deliver messages without the use of Aramis. Finally, in normal working conditions, the ABcast protocol should have similar performance to a traditional GM based protocol as, in the majority of cases, Aramis is not used for message delivery.

As the ABcast protocol utilises the probabilistic protocol Aramis, it is possible for a node to not receive an *abcast m* before Δ_m, resulting in that node delivering a subsequent *abcast* m', via Aramis, ahead of m in the total order. However, as previously stated, Aramis is carefully designed to keep the probability of meeting G4 close to 1. Furthermore, as Aramis is only used when Base is slow or node failures occur, the probability of an *abcast* message m being missed in the total order is the product of two very small probabilities; Base not being able to deliver m and Aramis failing m. Therefore, in reality the occurrence of a node not delivering m in the correct total order is rare.

4.1.4 ABcast Guarantees

Below, we state the guarantees provided by the ABcast protocol.

G1 - *Validity*: If the source of m_i does not crash until it *abcast*s m_i, then all operative destinations of m_i deliver m_i.

G2 - *Uniform Agreement*: If the source of m_i crashes while *abcast*ing m_i, and if any destination delivers m_i, then all operative destinations of m_i must deliver m_i.

G3 - *Uniform Integrity*: If m_i has already been delivered by a destination d, then d cannot deliver m_i again.

G4-P - *Probabilistic Total Order*: For any two abcast*s*, m_i and m_j, destinations that deliver both m_i and m_j, will deliver them in an identical order with a probability $> R$. Typically

R is close to 1 ($R \rightarrow 1$).

4.2 Assumptions

This section first defines the four key assumptions made when designing the Aramis protocol.

Assumptions:

A1 - Fault Tolerance

At most $(n-1)$ of n nodes involved in a broadcast can crash. However, 2 or more nodes cannot crash within an interval of some finite duration Δ_m that is smaller than a few seconds.

A2 - Synchronised Clocks

At any moment, clocks of any two operative nodes utilising ABcast are synchronised within 2ε with a probability at least as large as $(1-10^{-5})$.

We meet **A2** by implementing a well known probabilistic clock synchronisation algorithm [15]. The details of our implementation and the parameters used are explored in § 4.3.1.

A3 - Reliable Communication

When an operative node broadcasts m to all $m.dst$, all operative destinations $d \in m.dst$ will eventually receive m.

We use reliable UDP protocol to guarantee that all operative nodes receive m in crash-free scenarios. However, when a broadcasting node crashes, the use of reliable UDP alone is not enough to ensure that all of the operative destinations receive m. Therefore, a reliable broadcast, *rbcast*, protocol will be required. The Reliable UDP and *rbcast* protocol we use are explored in detail in § 4.3.3 and § 4.3.4, respectively.

A4 - Probabilistically Synchronous

Let x_{mx} be the maximum delay estimated at time t by observing NT_P transmissions in the recent past: The delay x_{mx} will not be exceeded in any of NT_F, $NT_F \leq NT_P$, transmissions to unfold after t with probability $(1-q)$; where q can be estimated with reasonable accuracy. The measurement of x_{mx} and q are presented in section 4.3.5.

A4 is motivated by previous research conducted by Ezhilchelvan *et al.* [22] into PSM, which proposes that the challenges of designing asynchronous distributed systems, namely the FLP impossibility, can be avoided by assuming that the underlying network communication is synchronous to a given probability. This assumption is crucial to Aramis's efforts in minimising the probability of G4-P not being met. Informally, the larger the estimated q, the more intensive the efforts made by Aramis to preserve these guarantees and *vice versa*.

A consequence of A4, is that Aramis is not suitable for use over the Internet, or similar networks that are susceptible to large fluctuations in network delays over a short period of time. This is because frequent occurrences of such fluctuations in NT_F can lead to q being underestimated, *i.e.* more violations of x_{mx} occur than indicated by q.

4.3 ABcast Components

In this section we detail the individual components required by the ABcast protocol. For each component, we explain its purpose and design; with important implementation details highlighted where appropriate. All of the protocols presented in this thesis are implemented in Java using the JGroups framework.

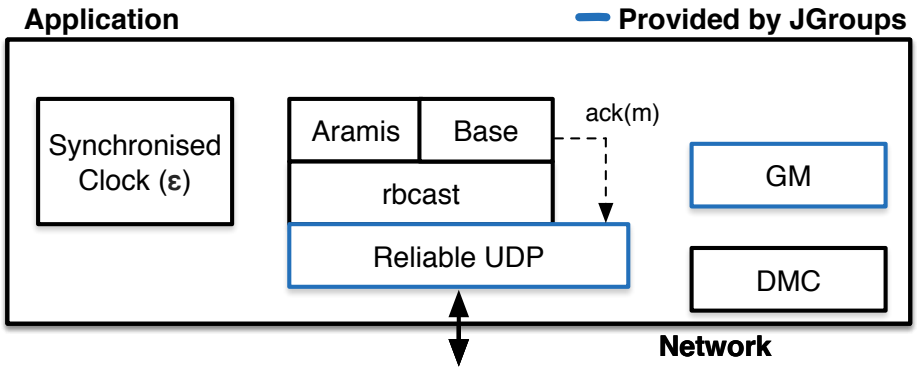

Fig. 4.1: ABcast Protocol Components

Figure 4.1 provides an overview of all of the components required by the ABcast protocol; where GM is the Group Membership service provided by JGroups, DMC is the Delay Measurement Component (4.3.5) and *rbcast* is the Reliable broadcast Component (4.3.4).

4.3.1 Clock Synchronisation

In order to provide synchronised clocks between nodes executing ABcast, we implemented the probabilistic clock synchronisation algorithm presented in [15] as a dedicated protocol in JGroups. Cristian's algorithm is a master/slave protocol, that utilises a single master node's clock time to synchronise all of the slave nodes; with each slave periodically issuing a clock synchronisation request to the master in order to synchronise their clock.

At any moment a slave's clock value is synchronised with the master node with a maximum error rate of ε, with probability $\mathscr{P}_{\varepsilon} \geq (1 - 10^{-5})$. All of the experiments presented in this thesis utilise clock synchronisation with ε estimated as 1 millisecond (ms). A major consideration when estimating ε is the worst-case rate of clock drift between successive synchronisations. Ultimately, the longer the synchronisation interval, the larger the drift rate between clocks. Estimation of $\varepsilon = 1$ usually assumes an interval of 45 minutes between synchronisations, however we use a shorter 15 minute interval in order to increase $\mathscr{P}_{\varepsilon}$.

As each slave node synchronises its clock value with that of the master, it is possible for any two slave nodes to have a maximum error rate of 2ε. This is because a slave N_i could synchronise its clock behind the master's clock value by ε time. Whereas, another slave N_j could synchronise its clock ahead of the master by ε. Hence, it is possible that $N_j.clockValue - N_i.clockValue = 2\varepsilon$.

4.3.2 Group Membership

JGroups provides a GM service, called GMS which simply stands for Group Membership Service. GMS works as follows: upon discovering that a new node has joined the group or a node failure has occurred, GMS issues a new view to all of the protocols in the JGroups stack. It is then the responsibility of the individual protocols to take the appropriate action when a new view is issued. For example, unblocking message delivery if the local node was waiting for an acknowledgement from a node that is no longer present in the newly issued view.

4.3.3 Reliable UDP

JGroups provides a reliable UDP protocol, UNICAST3, which guarantees that all UDP messages sent by a protocol higher in the network stack arrive at their destinations when node crashes do

not occur. This reliable UDP layer is placed below ABcast in the network stack to ensure that when messages are broadcast they are received by all destinations; where a broadcast consists of m being unicast via UNICAST3 to each of its intended recipients.

As well as providing reliable UDP unicasts, the UNICAST3 protocol provides *node-to-node* ordering as default for each message sent. This ordering means that if a node N_i sends two consecutive unicast messages, m_1 followed by m_2, to N_j, then N_j will not deliver m_2 until it has first delivered m_1. This behaviour is not always appropriate, therefore UNICAST3 allows for messages to be sent Out-Of-Band (OOB), which simply means that messages will be sent reliably but they will be delivered at a destination as soon as they are received, regardless of the messages that have (or have not) been delivered before it. Unless stated otherwise, our explanations assume that a unicast is sent using the default UNICAST3 behaviour *i.e.* not OOB.

4.3.4 Reliable Broadcast

In the event of a node failure reliable UDP alone is not sufficient to ensure that assumption A3 holds. This is because it is possible for a messages originator, $m.o$, to crash during the unicasting of m. Assume that $m.dst = \{N_i, N_j, N_k\}$ and $m.o = N_i$, if N_i crashes after unicasting m to N_j only, then N_k will never receive m. Similarly, if N_i crashes during the unicasting of m to N_j it is possible that N_i managed to send m before crashing, in which case m may eventually be received by N_j. Both scenarios highlight that an additional protocol is required to ensure that all $m.dst$ receive m in the event of $m.o$ crashing.

To overcome the limitations of Reliable UDP we have implemented a Reliable Broadcast protocol, called *rbcast*, that sits above the Reliable UDP layer in the network stack. This protocol is inspired by the work of Ezhilchelvan *et al.* [21, 24], as it utilises redundant broadcasts in collaboration with PSM, to ensure that all destinations receive a broadcast. Our *rbcast* protocol has been designed specifically for use with PSM based protocols and consequently utilises some of the values from the DMC (§ 4.3.5) as protocol parameters.

Below, we state the guarantees provided by the *rbcast* protocol. In stating them, we assume two primitives $rbcast(m)$ and $rb.deliver(m)$, which are described in detail later on in this section.

Probabilistic Guarantees of *rbcast*

RB1 - *RB-Validity with Probabilistic Timeliness*: If the source of m does not crash until it completes *rbcasting* m, then all operative s-nodes eventually $rb.deliver(m)$, and do so within D of $rbcast(m)$ with a probability $> R$. Typically $R \rightarrow 1$.

RB2 - *Agreement with Probabilistic Timeliness*: If $m.o$ crashes during $rbcast(m)$ and if any s-node *rb.delivers* m, then all operative s-nodes eventually $rb.deliver(m)$, and do so within D of $rbcast(m)$ with probability $> R$. Typically $R \rightarrow 1$.

RB3 - *Uniform Integrity*: Any s-node $rb.delivers(m)$ at most once.

RB1, RB2 and RB3 are directly used when implementing Aramis, to provide G1-G3 and G4-P, as detailed in section 4.4.2. The remainder of this section describes the basic *rbcast* protocol, whilst the calculations of D are presented in § 4.3.5.

The *rbcast* protocol

All messages broadcast via *rbcast* include a tuple $\{m.o, m.seq\#, m.ts\}$ that uniquely identifies the broadcast. Where $m.o$, short for message originator, is the address of the node that initiates a broadcast message; $m.seq\#$ is a sequence number unique to each $m.o$ that is incremented after each broadcast and $m.ts$ is a timestamp of $m.o$'s synchronised clock. Note that the first two values of the tuple are sufficient to uniquely identify a broadcast.

The *rbcast* protocol supports two primitives: $rbcast(m)$ and $rb.deliver(m)$. The protocol uses a set of parameters provided by the DMC, these are:

$\boldsymbol{x_{mx}}$ and $\boldsymbol{q}$ **-** As described in our assumptions.

$\boldsymbol{\eta}$ **-** The delay observed between redundant broadcasts of m, to ensure successive broadcasts remain independent as they are passed down the network stack. [2]

$\boldsymbol{\rho}$ **-** The number of redundant wait periods to be included into D for a given m; the larger the value of ρ, the closer R is to 1.

$\boldsymbol{\omega}$ **-** A node's estimate of the networks Packet Delay Variation (PDV).

[2] Utilising η ensures that lower protocols in the network stack do not bundle $m.copy = 0$ and $m.copy = 1$ into a single network packet, as this bundling would undermine the broadcasting of redundant message copies.

***rbcast*(*m*)**

The primitive *rbcast*(*m*) works as follows:

i The message to be broadcast, *m*, is created with the parameters defined above added as fields *e.g.* $m.\rho, m.\eta, \ldots$

ii The *m* to be *rbcast*(*m*) is then broadcast two times by the originator, with the second transmission occurring η time after the first one.

- Redundant transmissions of *m* are distinguished by the field *m.copy*, where $m.copy = 0 \vee 1$.

***rb.deliver*(*m*)**

As soon as a copy of *m* is received by a node, it is delivered up the stack to the higher level protocol (ABcast); the later copy of *m* is not delivered up the network stack, but is used as per the *rbcast* protocol.

A *rbcast*(*m*) is considered a success if every operative $d \in m.dst$ performs *rb.deliver*(*m*). Figure 4.2 shows the *rbcast* primitives and their relationship with the DMC and the underlying reliable UDP layer.

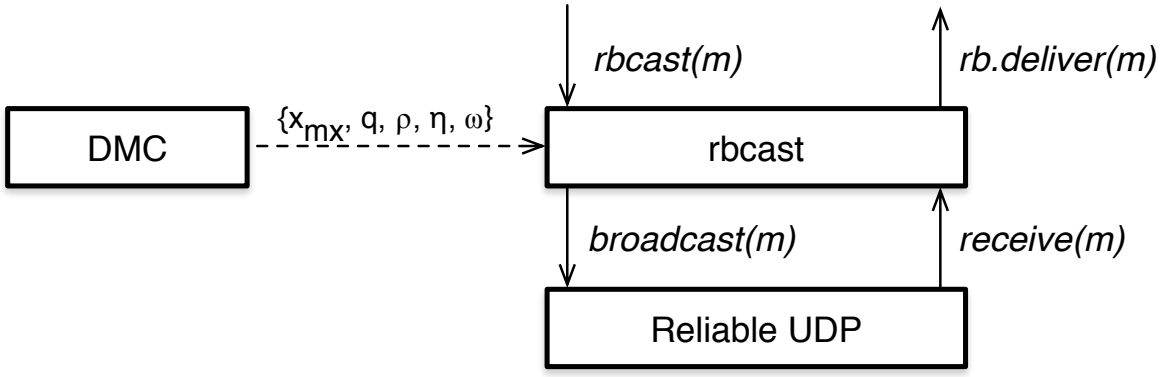

Fig. 4.2: Reliable Broadcast Interactions

After *rb.deliver*(*m*) has been executed, any destination N_j that receives $m.copy = 0 \vee 1$ cooperates to ensure *m* is successfully delivered by all non-crashed nodes. Upon receiving $m.copy = 0$, N_j waits to receive $m.copy = 1$ for t_1 time; where η and ω are included in *m*'s metadata by $m.o$ (N_i) and $t_1 = \eta + \omega$. If t_1 expires before receiving a subsequent copy of *m*, N_j

assumes that N_i has crashed and starts broadcasting the largest value of $m.copy$ it has received, until $m.copy = 1$ has been broadcast. Note, N_j rebroadcasts the largest copy of m that it has received, as it does not know if all other $d \in m.dst$ have also received this copy of m.

To reduce the probability that multiple nodes are re-broadcasting m simultaneously, N_j waits a further ζ time to receive subsequent copy of m from $m.o$, or another node in $m.dst$, before broadcasting the latest $m.copy$; where ζ is uniformly distributed on $(0, \eta)$. This process continues until all $d \in m.dst$ have received or broadcast m with $m.copy = 1$.

Note that if N_i does not crash, or if it crashes and an operative node receives m, then m is broadcast *at least* twice. Conversely, if N_i crashes during the initial broadcast of m, and no members of $m.dst - \{m.o\}$ receive a copy of m, then the broadcast has failed and m is lost. This is acceptable for ABcast, because if no $d \in m.dst - \{m.o\}$ receive m, then its not possible for any node to $rb.deliver(m)$, therefore it is not possible for *abcast* guarantees G1 or G2 to be violated.

Implementation Optimisation

It is worth noting that in our implementations of *rbcast*, every broadcast where $m.copy = 0$ is treated differently to subsequent broadcasts of m. Copy 0 of m is broadcast to all destinations using the default settings of Reliable UDP, i.e. messages are delivered in the same order that they were originally unicast from their source address. This means that if node N_i broadcasts m, followed by m', it is not possible for any of the destinations to receive copy 0 of m' before it has received m. This also implies that if the transmission of copy 0 of m is slow or becomes lost, than copy 0 of m' cannot be forwarded up the network stack to *rbcast* protocol until m has been received. To overcome this issue all messages with $m.copy > 0$ are sent OOB to ensure that they are forwarded to the *rbcast* protocol as soon as they are received at the destination node. Therefore, if $m'.copy = 0$ has been received but not been forwarded to *rbcast*, then $m'.copy = 1$ is forwarded as soon as it arrives, bypassing the backlog of messages.

The calculations used to produce x_{mx}, η, ρ and ω are discussed in detail in § 4.3.5.

4.3.5 Delay Measurement Component (DMC)

For the sake of clarity, assumption A4 is repeated below:

Let x_{mx} be the maximum delay estimated at time t by observing NT_P transmissions in the recent past: The delay x_{mx} will not be exceeded in any of NT_F, $NT_F \leq NT_P$, transmissions to unfold after t with probability $(1-q)$; where q can be estimated with reasonable accuracy.

The delay measurement component is responsible for monitoring and observing the network latency of NT_P transmissions from the recent past. These latencies are then used to calculate various parameters that are required by *rbcast* (and by Aramis) for executing *abcasts* in the near future. Being conservative, we use $NT_F = 10\%$ of NT_P and $NT_P = 1000$; so, a ABcast node freshly estimates x_{mx} for every 100 new delays it observes. Each fresh estimation of x_{mx} results in the recalculation of the following parameters: η, ρ, q and ω.

Latencies are measured by the DMC based upon the timestamp $m.ts$, which is included in every message m that is broadcast via the *rbcast* protocol. As the clocks of all nodes executing the ABcast protocol are synchronised, it is possible to measure the one-way latency of each message that is received by a node. For example, a node N_i sends an *rbcast* m to N_j, upon receiving m, N_j immediately records the latency x:

$$x = (N_j.clockValue - m.ts) + 2\varepsilon \tag{4.1}$$

It is necessary to add 2ε to each latency to ensure that if $N_j.clockValue$ is behind $N_i.clockValue$ by the maximum error of 2ε, a positive latency value is still recorded.

The remainder of this section explores each of the parameters provided by the DMC, explaining what they represent and how they are calculated, before providing a proof for *rbcast* guarantees RB1 and RB2.

x_{mx}

x_{mx} is simply the largest latency out of the NT_P latencies observed in the recent past.

q

The parameter q is the estimated probability that a transmission delay observed in the near future will exceed x_{mx}. We estimate q by assuming that each x in NT_p increases by 5% in the near future. So, the estimated set of transmission delays that exceed x_{mx} in the

near future is:

$$\{x \text{ in } NT_p \quad | \quad 1.05 \times x \quad > \quad x_{mx}\} \tag{4.2}$$

Note that:

$$1.05 \times x \quad > \quad x_{mx} \quad \Rightarrow \quad \frac{x_{mx}}{1+0.05} \approx 0.95 x_{mx} \tag{4.3}$$

So,

$$q = \frac{\text{Number of Transmissions in } NT_p \text{ that exceeds } 95\% x_{mx}}{|NT_P|} \tag{4.4}$$

It is possible that q is calculated close to 1 if many latencies in the recent past are within 95% of x_{mx}. As $q \to 1, \rho \to \infty$ so that R is close to 1. To deal with such extreme cases, we fix an upper bound ρ_{mx}, $\rho_{mx} > \rho_{mn}$. Therefore we find q to be the largest value that satisfies:

$$q \quad < \quad \left(1 - \tilde{R}\right)^{\left(\frac{1}{\rho_{mx}+1}\right)}, \text{ where } \tilde{R} = R^{\frac{1}{n-1}} \tag{4.5}$$

η

η is the parameter used by *rbcast* to determine the amount of time to wait between each broadcast of a message copy and is calculated as the maximum of x_{mx} and the largest delay incurred with probability R, when a given copy of m is sent to $(n-1)$ destinations by reliable UDP. The latter is calculated by assuming that delays are exponentially distributed with mean $\bar{x}$. Thus,

$$\eta = maximum\left\{x_{mx}, -\bar{x}[\ln(1-\tilde{R})]\right\} \tag{4.6}$$

where $\tilde{R} = R^{\frac{1}{n-1}}$ and $\bar{x}$ is the $67th$ percentile of delays observed in NT_p.

The second term in Equation 4.6 turns out to be $5.3\bar{x}$ when $R = 0.99$ and $n = 3$. In general, as $R \to 1$ (*e.g.* $R = 99.99\%$), $\tilde{R} \to 1$ and $\ln(1-\tilde{R}) \to -\infty$. Therefore one may wish to

define η simply as:

$$\eta = -\bar{x}[\ln(1-\tilde{R})] \tag{4.7}$$

Equation 4.7 leads to η being immune to large outliers in NT_p which exceed $-\bar{x} \times \ln(1-\tilde{R})$ and thus inflate x_{mx}. It may be unnecessary to delay the second, redundant transmission so excessively, solely in response to such large outliers.

ρ

ρ is a parameter that defines the number of redundant wait periods incorporated into the calculation of D so that RB1 and RB2 are met with a probability $> R$; where R is a configuration parameter specified before runtime.

The probability that a given operative destination receives at least one of $m.copy = 0 \vee 1$, before $m.ts + x_{mx}$ is:

$$1 - q^{(2)} \tag{4.8}$$

Given that there can be $(n-1)$ operative destinations at any time, we require:

$$\left[1 - q^{(2)}\right]^{n-1} \quad > \quad R \tag{4.9}$$

As we cannot change n, it is necessary for us to introduce the variable ρ which determines the number of additional η length wait periods should be incorporated into D. Therefore, the previous equation becomes:

$$\begin{aligned} \left[1 - q^{(\rho+1)}\right]^{n-1} \quad &> \quad R \\ 1 - q^{(\rho+1)} \quad &> \quad R^{\frac{1}{n-1}} = \tilde{R} \end{aligned} \tag{4.10}$$

Rearranging, taking in both sides and accounting for the fact that $\ln(a) < 0$ for $0 < a < 1$,

we set:

$$(\rho+1) \quad > \quad \frac{\ln\left(1-\tilde{R}\right)}{\ln(q)}, \quad \forall \quad q \neq 1 \tag{4.11}$$

i.e:

$$\rho \quad > \quad \frac{\ln\left(1-\tilde{R}\right)}{\ln(q)} - 1, \quad \forall \quad q < 1 \tag{4.12}$$

In theory, it is possible for $\rho = 0$ to be a valid parameter as it is possible for $(1-q) > R$. However, as all of our calculations have been defined pessimistically, we define ρ to be the smallest integer that satisfies:

$$\rho \quad > \quad maximum\left\{\rho_{min}, \quad \frac{\ln\left(1-\tilde{R}\right)}{\ln(q)} - 1\right\} \tag{4.13}$$

Where $\rho_{min} \geq 1$ is a configuration parameter specified before runtime; unless otherwise stated we utilise $\rho_{min} = 1$. Observe that, for a given R, an integer $I = \rho_{min}, \rho_{min}+1, \ldots$, satisfies:

$$I \quad < \quad \frac{ln\left(1-\tilde{R}\right)}{ln(q)} - 1 \quad < \quad I+1 \tag{4.14}$$

for a wide range of q values; e.g., for $R \approx 0.9999$

$$\frac{ln(1-\tilde{R})}{ln(q)} - 1 < 1 \quad \forall \quad q < 0.01 = 1\% \tag{4.15}$$

This implies that small inaccuracies in estimating q may not adversely affect ρ estimates.

ω

ω is the parameter utilised by *rbcast* to approximate the PDV encountered by the network. ω is simply calculated as:

$$\omega = \eta - \bar{x} \tag{4.16}$$

Again, we assume exponential distribution and that $\bar{x}$ is the exponential mean of NT_P observed delays.

Guarantees of *rbcast*

Let m be *rbcast* by N_i, as per its clock time $m.ts$. Let:

i $D_1 = \rho\eta + x_{mx}$

ii $D = x_{mx} + 2\eta + \omega + D_1$

iii $D_m = D + 2\varepsilon$

Let x, as before, denote a delay (observed) and X the (random) delay variable when m is being *rbcast*. Finally, assume for now that $\varepsilon = 0$.

RB1 - RB-Validity with Probabilistic Timeliness

Figure 4.3 shows two multicasts by a correct N_i, with $n = 3$.

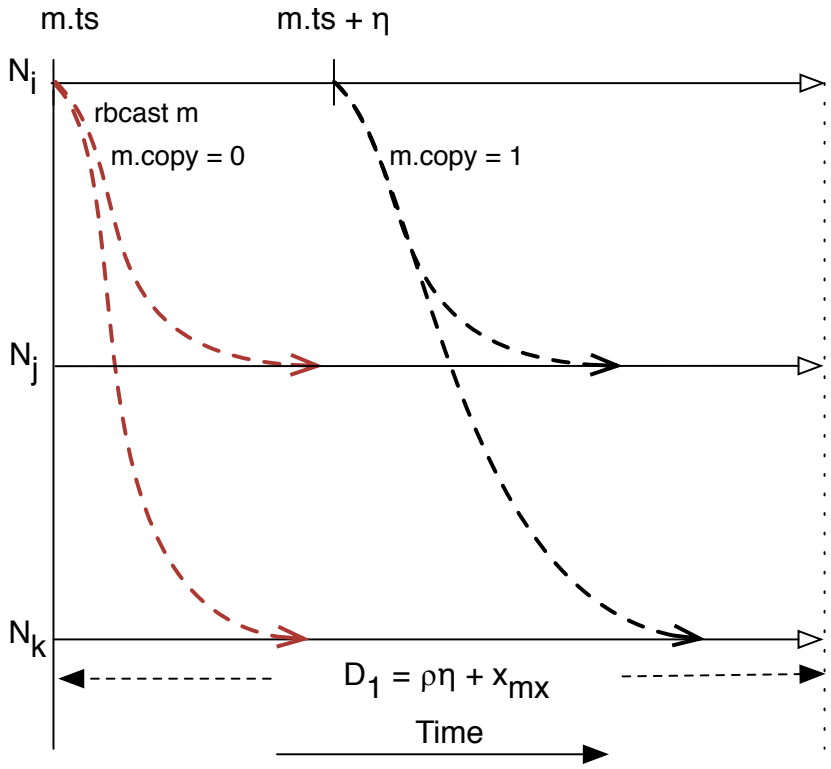

Fig. 4.3: RB1 (RB-Validity) Calculations

A Single Correct Destination Receives m

To start with, let us focus on $m.copy = 0$ and a given correct destination, say N_j. The probability that N_j receives $m.copy = 0$ by its clock time $m.ts + x_{mx}$ is $(1-q)$ by assumption A4. *i.e.* $\mathscr{P}(N_j \text{ receives } m.copy = 0 \text{ by } m.ts + x_{mx}) = (1-q)$.

Assuming that X is exponentially distributed with parameter λ,

$$\mathscr{P}(N_j \text{ receives } m.copy = 0 \text{ by } m.ts + x_{mx}) = \left(1 - e^{-\lambda x_{mx}}\right) \tag{4.17}$$

Therefore,

$$1 - e^{-\lambda x_{mx}} = 1 - q \implies e^{-\lambda x_{mx}} = q \tag{4.18}$$

Let us add one redundant wait period of x_{mx}. By the property of exponential distribution,

$$\begin{aligned} \mathscr{P}(N_j \text{ receives } m.copy = 0 \text{ by } m.ts + 2x_{mx}) &= \left(1 - e^{-\lambda 2x_{mx}}\right) \\ &= \left(1 - \left(e^{-\lambda x_{mx}}\right)^2\right) \\ &= 1 - q^2 \end{aligned} \tag{4.19}$$

Adding 2 redundant wait periods, leads to:

$$\mathscr{P}(N_j \text{ receives } m.copy = 0 \text{ by } m.ts + 3x_{mx}) = 1 - q^3. \tag{4.20}$$

In general, with ρ, $\rho \geq 1$, redundant wait periods:

$$\mathscr{P}(N_j \text{ receives } m.copy = 0 \text{ by } m.ts + (\rho+1)x_{mx}) = 1 - q^{(\rho+1)} \tag{4.21}$$

All Correct Destinations Receive m

$$\begin{aligned} \mathscr{P}(\text{all } (n-1) \text{ correct destinations receive } m.copy = 0 \text{ by} \\ m.ts + (\rho+1)x_{mx}) = \left[1 - q^{(\rho+1)}\right]^{n-1} > R \end{aligned} \tag{4.22}$$

If there are fewer correct destinations then $(n-1)$, say $(n-2)$, then the above probability

is:

$$\left[1-q^{(\rho+1)}\right]^{n-2} > \left[1-q^{(\rho+1)}\right]^{n-1} > R \tag{4.23}$$

When $\eta \geq x_{mx}$ (see Equation 4.6):

$$D = \eta\rho + x_{mx} \geq \rho x_{mx} + x_{mx} = (\rho+1)x_{mx} \tag{4.24}$$

Moreover, $D > D_1$, so:

$$\begin{aligned}\mathscr{P}(\text{all } (n-1) \text{ correct destinations receive } m.copy = 0 \text{ by} \\ m.ts + D) > \left[1-q^{(\rho+1)}\right]^{n-1} > R\end{aligned} \tag{4.25}$$

Accounting for $\varepsilon > 0$ and recalling $D_m = D + 2\varepsilon$:

$$\mathscr{P}(\text{each correct destinations receives } m.copy = 0 \text{ by its clock time } m.ts + D_m) > R \tag{4.26}$$

Note: The RB-Validity property is met when a correct N_j receives either $m.copy = 0$ or $m.copy = 1$ by its cock time $m.ts + D_m$. Since, $\rho \geq 1$,

$$\begin{aligned}\mathscr{P}(\text{RB-Validity is met within } D_m) > \\ \mathscr{P}(\text{Each correct } N_j \text{ receives } m.copy = 0 \text{ by its clock time } m.ts + D_m) \\ > R\end{aligned} \tag{4.27}$$

RB2 - Agreement with Probabilistic Timeliness

Suppose now that N_i crashes before completing the redundant transmissions of m and $n > 2$. Suppose also that only one node, N_j, has m with $m.copy = 0$. This is the worst case to be considered because if N_j $m.copy > 0$, then N_i crashed only after it completed broadcasting $m.copy \geq 0$, therefore some node other than N_j also has m; the more destinations which receive some copy of m, the more likely it is that $rbcast(m)$ is completed. On the other hand, if no destination receives any copy of m from the crashed N_i, then the case for discussion does not exist.

If copy $m.copy = 0$ takes at most x_{mx} to reach N_j (which occurs with probability $(1-q)$), N_j would start disseminating on behalf of N_i at or before time:

$$m.ts + x_{mx} + \eta + \omega + \zeta \tag{4.28}$$

Recall that ζ is the random wait that all disseminating nodes must observe before disseminating m, with ζ uniformly distributed on $(0, \eta)$. Therefore, in our calculations we assume that the observed ζ is the largest value possible, η and ζ in equation 4.28 is replaced by η in the discussions below. Figure 4.4 shows the worst case scenario when $n = 3$.

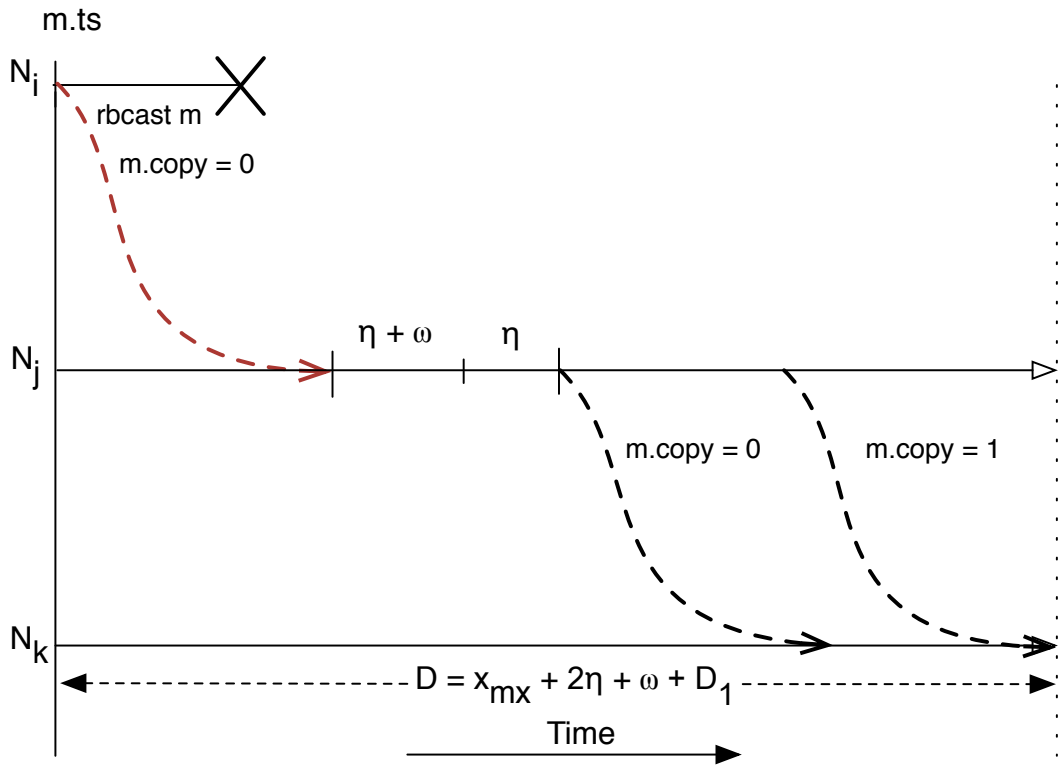

Fig. 4.4: RB2 Calculations - Worse Case

Simplified Scenario

As before, we assume that propagation delays are exponentially distributed with parameter λ. Also suppose, for now, that $\varepsilon = 0$ and N_j instantaneously transmits its $m.copy = 0$ without waiting for $(\eta + \omega)$ and a random wait [3].

Of interest, is the probability, P_{D2}, that a given node, N_k receives $m.copy = 0$ from N_j by $m.ts + D_2$; where $D_2 = (\rho + 2)x_{mx}$. Recall that $q = e^{-\lambda x_{mx}}$ by the exponential assumption. The

[3] Both of these simplicity assumptions will be removed shortly.

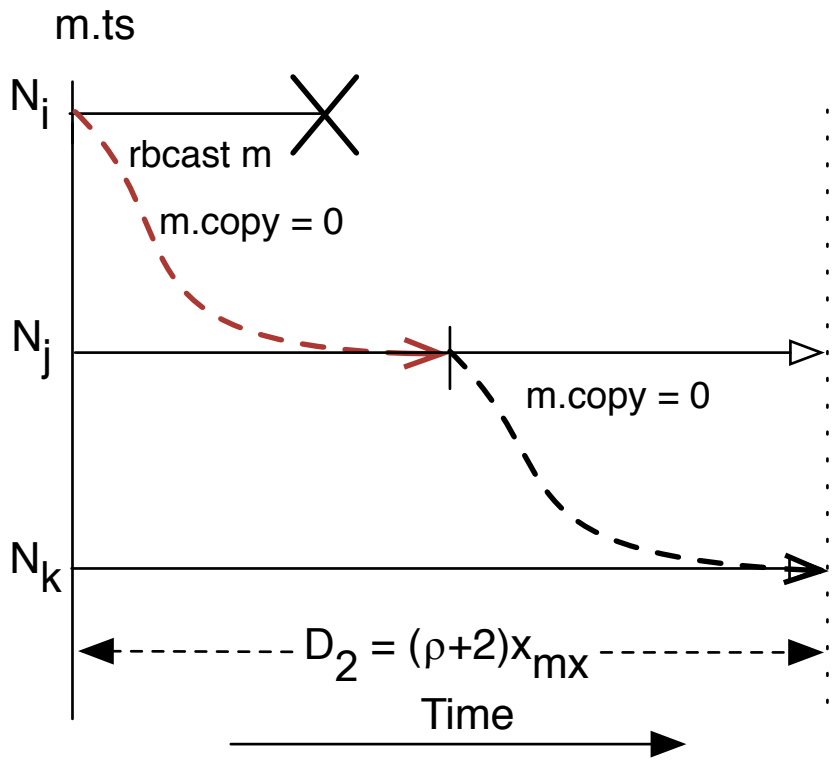

Fig. 4.5: RB2 Calculations - Simplified Scenario

time when N_k receives $m.copy = 0$ is determined by the sum of two independent exponential delays (see Figure 4.5):

i N_i to N_j

ii N_j to N_k

This sum follows Erlang distribution with parameter λ and number of *hops*, k, $k = 2$. So,

$$\begin{aligned} P_{D2} &= 1 - \sum_{n=0}^{k-1} \left[\left(\frac{1}{n!} \right) e^{-\lambda D_2} (\lambda D_2)^n \right] \\ &= \left[e^{-\lambda D_2} + e^{-\lambda D_2} (\lambda D_2) \right] \\ &= \left[e^{-\lambda D_2} (1 + \lambda D_2) \right] \end{aligned} \tag{4.29}$$

Recall that $e^{-\lambda x_{mx}} = q$ (see equation 4.18),

$$\begin{aligned} -\lambda x_{mx} &= \ln(q) \\ e^{-\lambda D_2} &= e^{\lambda(\rho+2)x_{mx}} = q^{(\rho+2)} \\ \lambda D_2 &= \lambda x_{mx}(\rho+2) = -(\rho+2)\ln(q) \end{aligned} \tag{4.30}$$

From equation 4.29:

$$\begin{aligned} P_{D2} &= 1 - \left[q^{(\rho+2)}\left(1-(\rho+2)\ln q\right)\right] \\ &= 1 - q^{(\rho+1)}\left[q\left(1-(\rho+2)\ln q\right)\right] \\ &= 1 - q^{(\rho+1)}\left[q-(\rho+2)q\ln q\right] \end{aligned} \tag{4.31}$$

Let $q\ln q = 0$. As q values are $q \to 0_+$ typically small, we approximate:

$$q-(\rho+2)q\ln q \lesssim 1 \tag{4.32}$$

So, applied to Equation 4.31 we get:

$$P_{D2} = 1 - q^{(\rho+1)}\left(q-(\rho+2)q\ln q\right) \gtrsim 1 - q^{(\rho+1)} \tag{4.33}$$

Therefore, the probability that all $(n-2)$ correct N_k receive $m.copy = 0$ from N_j by $m.ts + D_2$ is:

$$[P_{D2}]^{n-2} \gtrsim \left[1-q^{(\rho+1)}\right]^{n-2} > \left[1-q^{(\rho+1)}\right]^{n-1} > R \tag{4.34}$$

When $\eta \geq x_{mx}$,

$$D-(2\eta+\omega) = \rho\eta + 2x_{mx} \geq (\rho+2)x_{mx} \tag{4.35}$$

Thus, the probability that every correct N_k receives $m.copy = 0$ from N_j, by $m.ts + D - (2\eta+\omega)$, is larger than $[P_{D2}]^{(n-2)} > R$.

Simplifications Removed

Accounting for the fact that N_j *rbcasts* m within at most $(2\eta+\omega)$ time after receiving $m.copy = 0$ from N_i and $\varepsilon \neq 0$:

$$\mathscr{P}(\text{each correct } N_k \text{ receives } m.copy = 0 \text{ by } \underline{\text{its}} \text{ clock time } m.ts + D + 2\varepsilon) > R \tag{4.36}$$

Thus, the agreement property is met for the delay of $D_m = D + 2\varepsilon$, with probability $> R$.

Remarks

The simplifying approximation $(q-(\rho+2)q\ln q)$ is not unreasonable; it evaluates to be 0.80 and 0.95 when $q = 5\%$ and $\rho = 3$ and $\rho = 4$, respectively. Other than this approximation, the agreement anaylsis is pessimistic by considering the worse-case crash scenario. When N_i crashes during a multicast, several scenarios are possible, with just some or all destinations receiving $m.copy = 0$. In the former case, several N_j may multicast $m.copy = 0$, increasing the chances of the other destinations receiving $m.copy = 0$ within D_m. In the latter case the analysis is similar to that presented for RB-Validity.

4.4 Atomic Broadcast Protocol

Hitherto this chapter has focused on the underlying assumptions made when designing the ABcast protocol and the components it requires to function. This section focuses on how the hybrid protocol functions, detailing the specifics of both the Aramis and Base protocols. First we explore the Base protocol, as this protocol will be responsible for the majority of message deliveries and is the more conventional of the two *abcast* protocols. We then explore the Aramis protocol, detailing how it utilises the guarantees of *rbcast* to calculate Δ_m. This followed by a formal definition of ABcast's delivery conditions. Finally, we discuss how the ABcast's reliance on the DMC requires an initialisation period to be observed by the protocol before *abcast*ing can begin.

4.4.1 Base

The Base protocol is based upon the NewTop [23] algorithm discussed in § 2.2.4, with a few key differences motivated by our use of synchronised clocks and PSM.

The Base protocol works as follows when a node N_i sends an *abcast* m: N_i *rbcast*s m, and as per the *rbcast* protocol m is assigned an id tuple $\{m.o, m.seq\#, m.ts\}$. This tuple is utilised by Base to specify the total order of m, with all destinations in $m.dst$ ordering messages in ascending order based upon their timestamp. In the event of any two messages having the same $m.ts$ value, the address specified in $m.o$ is used for tie-breaking to ensure a total order; note this is highly unlikely in practice as $m.ts$ is recorded in nanoseconds. Similarly, in our implementation it is not possible for the same node to *rbcast* two messages with the same $m.ts$, as a single thread called the *sender* thread, is used for sending all m with $m.copy = 0$.

Like NewTop, delivery of m is blocked until each $d' \in m.dst - \{m.o\}$ has acknowledged m by sending $ack_{d'}(m)$ to every $d \in m.dst$ and all $d \in m.dst$ have received $ack_{d'}(m) \forall (m.dst \setminus \{m.o, d\})$, thus C1 (§ 2.2.2) is met. With each acknowledgement consisting of the id tuple that belongs to the message being acknowledged.

The use of synchronised clocks to uniquely timestamp each message, removes the need for tentative timestamps to be shared between destinations. Instead the ordering of a message is dictated from its inception based upon its timestamp. However, a message's final place in the total order is not known by a destination until it has received an acknowledgement from all

other $d' \in (m.dst - \{m.o\})$. Figure 4.6 shows the message flow required by the base protocol in order for an *abcast* to be delivered.

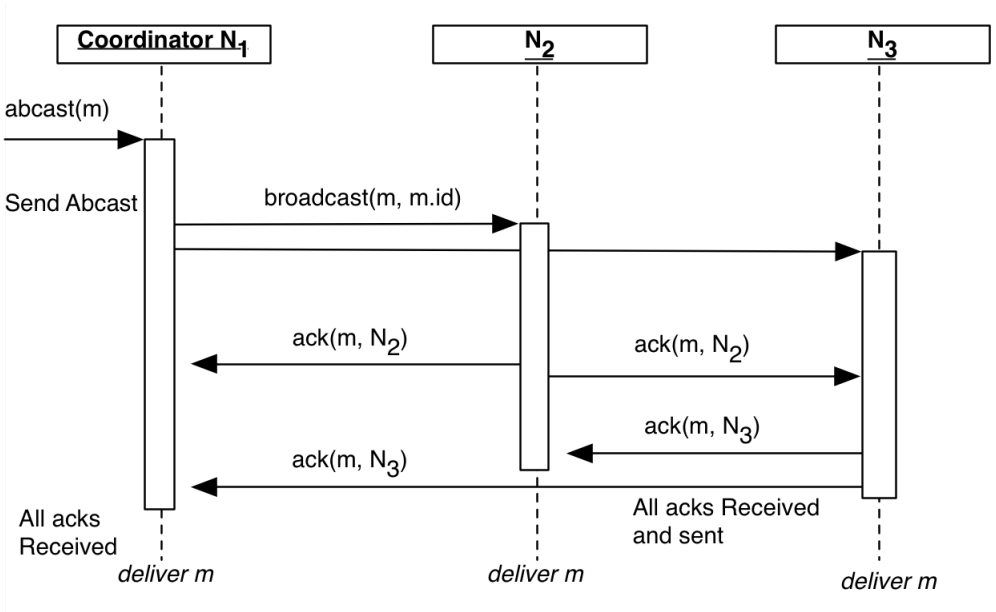

Fig. 4.6: Base Atomic Broadcast protocol

In order to overcome C2, as described in 2.2, and ensure that G3 and G4 are respected, it is necessary for each $d \in m.dst$ to maintain a vector clock [26, 55]; with each d's vector clock stating the last $rbcast(m)$ sent by d as well as the the latest m to be *rb.delivered*, by d, that originated from each $d' \in (m.dst - d)$ [4]. This vector clock is then included in every m and $ack_{d'}(m)$ sent by a node.

For any message m, regardless of whether it originated at the current node, it is necessary for the associated acknowledgements and vector clocks to be checked to see if a message is missing from the total order before m can be delivered. If a node N_i has not received m, but has learnt of its existence via a *vector clock* or an acknowledgement, then we consider m to be *known* by N_i. The Base delivery conditions are formalised below:

Base Delivery Rule:

A node, N_j delivers any m, via Base, only after $D1_B$ and $D2$ stated below are satisfied:

$D1_B$ - m is acknowledged by all nodes other than $m.o$.

$D2$ - all *known* m', with $m'.ts < m.ts$ have been delivered.

[4] Where the latest m, is defined as the message containing the largest timestamp from a given d'

Acknowledgement Piggybacking

In the explanation of Base we assume that each acknowledgement is explicitly sent as dedicated message, however in practice this is an expensive operation in terms of both latency and bandwidth. Therefore in our implementation of Base we piggyback message acknowledgements on subsequent *rbcast*s sent by an the acknowledging node; acknowledgements are piggybacked onto all copies of m, i.e. all $m.copy = 0, 1$, or not at all. Of course this is only appropriate if there is a message waiting to be *rbcast*, otherwise deadlock will occur at all $d \in m.dst$ as an acknowledgement will never be sent. Consider, node N_j is attempting to send $ack_{N_j}(m)$ to N_i, if N_j does not receive an *abcast* request within $\mathscr{A}_d$ time, then an explicit acknowledgement message is sent to N_i containing $ack_{N_j}(m)$, as well as any other pending acknowledgements. We define $\mathscr{A}_d$ as $\mathscr{A}_d = 2\eta + \omega$ and an explicit acknowledgement as being a dedicated message m_{ack} that is unicast to all $d \in m.dst$ and is not assigned a *rbcast* id; hence only a single copy of m_{ack} is broadcast via reliable UDP. [5]

4.4.2 Aramis

As previously stated, Aramis is a non-blocking probabilistic *abcast* protocol that utilises a calculated delivery delay Δ_m to place an upper bound on message deliveries. The Aramis protocol works in conjunction with Base to ensure that message delivery does not become blocked in the event of slow or crashed nodes. However, the Aramis protocol does not simply deliver each received message after Δ_m has expired, as this could cause a *known* message to be missed in the total order. Instead, it utilises the acknowledgements and *vector clocks* that are integral to Base to ensure that *known* messages are not missed from the total order. Therefore, if a message m's Δ_m delay expires, the message can only be delivered after all *known* messages that precede m in the total order have been delivered. The Aramis delivery conditions are formalised below:

Aramis Delivery Rule:

A node, N_i delivers any m, via Aramis, only after $D1_A$ and $D2$ stated below are satisfied:

$D1_A$ - The clock of $N_i > m.ts + \Delta_m$, where $\Delta_m = 2(D + \varepsilon) + \mathscr{A}_d$.

$D2$ - All *known* m', with $m'.ts < m.ts$ have been delivered.

[5] Explicit acknowledgements are not *rbcast* in order to further minimise the bandwidth cost of sending m_{ack}.

Calculating Δ_m

Let us, for simplicity, assume that $\varepsilon = 0$. The explanation is twofold. First, recall that *rbcast* guarantees on agreement: if N_i *rbcast*s m and if any operative s-node (be it N_i or otherwise) *rb.delivers* m, all destination nodes *rb.deliver* m within $m.ts + D$ with probability $> R$.

Secondly, the aim of Aramis is to aid the delivery of *abcast* messages when Base is blocked due to node crashes. So, delivery by Aramis is delayed until all acknowledgements are *rb.delivered* when there are no crashes. Since an operative node can acknowledge at most $\mathscr{A}_d$ time after *rb.delivering* m, we have:

$$\Delta_m \geq D + \mathscr{A}_d + D \tag{4.37}$$

Accounting for the clock synchronisation error rate of 2ε, we calculate Δ_m as:

$$\Delta_m = 2(D + \varepsilon) + \mathscr{A}_d \tag{4.38}$$

Note: By delivering m at $m.ts + \Delta_m$, $\Delta_m > 2 \times D$, Aramis indeed meets the total order (G4-P) guarantee with a probability much larger than the user specified value of R.

4.4.3 Aramis and Base - ABcast

The ABcast protocol, is a hybrid solution that combines the delivery conditions of the Aramis and Base protocols. Here we present a concise formalisation of the delivery rule for the entire ABcast protocol. A node, N_j delivers any m via ABcast, only after both $D1$ and $D2$ stated below are satisfied:

$D1$ - The clock of $N_i > m.ts + \Delta_m$ ($D1_A$) or m is acknowledged by all nodes other than $m.o$ ($D1_B$).

$D2$ - All *known* m', with $m'.ts < m.ts$ have been delivered.

Extremely Delayed *rbcast* Messages

As Aramis is a probabilistic protocol it is possible for a message m to be *rb.delivered* at destination d, after its proceeding message m' in the total order has already been *ab.delivered*; where

ab.delivered refers to a *m* being delivered by ABcast. In such a case, there is one of two actions possible:

- Discard *m* when it becomes known to *d*; resulting in a violation of G1 and G2.
- Deliver *m* to the application via an exception; resulting in a violation of G4.

Aramis takes the second option and throws an exception when a *rb.delivered* message is not *ab.delivered*; we refer to this process as a message being *rejected*. Explicitly *rejecting* a message from the *ab.delivery* allows for higher levels in the network stack (e.g. SCast) to initiate an appropriate recovery mechanism to mitigate the effects of G4 ordering violations on the system's state. Furthermore, as the *rejected* message is still being delivered to the application, albeit via an exception, it is possible for the payload of the *rejected* message to be utilised by the application as part of its recovery mechanism for ordering violations.

Message Rejections

A message, *m*, sent by node N_i, can only be *rejected* by another recipient, N_j, when a message m' from N_j has been incorrectly delivered before *m* in the total order; where $m.ts < m'.ts$ but m' is incorrectly *ab.delivered* first. In order for m' to be *ab.delivered* ahead of *m*, resulting in *m* being *rejected* by N_j, it is necessary for both of the statements below to be true:

1. N_j *rbcasts* m' with $m'.ts > m.ts$, but *m* is not *rb.delivered* by N_j before $m'.ts + \Delta_{m'}$.
2. N_j does not receive an acknowledgement for *m*, or any *rbcasts* sent after *m* by N_i, before $m'.ts + \Delta_{m'}$.

If condition one is not true, i.e. N_j *rb.delivers m* before $m'.ts + \Delta_{m'}$, then N_j has received *m* before m''s delivery time and therefore N_j will not miss *m* in the total order. Similarly, if condition two is not true, then N_j will know that a message sent by N_i is missing as soon as it inspects the received acknowledgement, attached *vector clock* or the *seq#* of the *rb.delivered* message. When both conditions hold, it is guaranteed that m' will be *ab.delivered* via Aramis, as it is impossible for Base to *ab.deliver* a message if condition two is true.

The rules presented above can be extended to cater for when $n > 2$. Assuming the same scenario described above, we introduce N_k which represents all nodes involved in an *abcast* that

are not N_i or N_j. A message is *rejected* by a node N_j if conditions 1 and 2 are true, as well as the condition stated below:

3. N_j does not *rb.deliver* an acknowledgement of m from any N_k, and it does not *rb.deliver* a *rbcast* from any N_k, that has *rb.delivered* m, before $m'.ts + \Delta_{m'}$ [6].

If conditions $1, 2$ and 3 are true it is not possible for m to be known by N_j before $m'.ts + \Delta_{m'}$. This is because m has not been *rb.delivered* by N_j and N_j does not know of m, as m has not been acknowledged, or described in a vector clock, by any N_k, before m' is *ab.delivered* by N_j.

4.4.4 Initialisation Period

The ABcast protocol requires a 'warm-up' period before *abcast*s can be sent between nodes. This period is required in order to:

i Synchronise the clocks of all participating nodes in the view.

ii Ensure that each node's DMC has recorded at least NT_p latencies

Synchronisation must be performed first as the DMC is dependent on this assumption. Our solution to recording NT_p latencies, is to incorporate a mandatory *probing* period that must be observed by all nodes in the current view after their clocks have been synchronised and before *abcast*ing can begin.

The probing period required during initialisation utilises 'empty' probe messages to record NT_p latencies at each node's DMC. An empty probe consists of a message, with a payload the size of those expected during *abcast*ing, being unicast to all n nodes in the current view. This requires each node in the view to send at least $\frac{NT_P}{n}$ probes, however in reality the number should be higher to take into account that nodes will start the initialisation process at different times. With each subsequent probe being broadcast x time apart; with x being a value determined before run-time that should be an approximation of the expected frequency of *abcast*s. Once all nodes in the view have sent their probes, and recorded at least NT_p latencies, it is possible for this node to start executing *abcast*s.

A disadvantage of utilising empty probes is the risk of network latencies being over-or underestimated respectively, due to unknown load conditions that an application would experience

[6] Where $m'.ts + \Delta_{m'}$ is based upon N_j's local clock.

later. Such inaccuracies are corrected as the application progresses and will not be an issue if crashes do not occur at the start of the application.

4.4.5 Initialising a Newly Joined Node

When a new view is issued containing a new node, N_i, it is necessary for N_i to undergo an initialisation period similar to that described in the previous section. Clock synchronisation is simple, as N_i can just contact the designated master node (as per [15]) and initiate the synchronisation protocol.

Recording the required NT_p latencies is slightly trickier, as utilising a probing period similar to the initialisation period could have an adverse effect on all other nodes in the view. This is because the existing nodes will most likely be heavily loaded from application requests, hence the need for an additional *s*-node, and adding additional load over a short period of time would be detrimental to performance. Therefore, we propose that a better solution would be for new nodes to be *silent watchers* until NT_p latencies have been recorded. A silent watcher, is a node that receives *abcast*s from all other nodes in the view, but is unable to initiate its own *abcast*s until after it has received NT_p *abcast*s and hence recorded NT_p latencies. When a new view is issued by the GM service, existing nodes include N_i in the destination set of subsequent *abcast*s, resulting in N_i eventually receiving NT_p messages.

Note: While N_i is considered a silent watcher, it is still possible for it to participate in the redundant *rbcast*s of message copies as the required timeout values are transmitted along with the message itself.

4.5 Flow Control

The ABcast protocol described in this section functions as expected when each node's throughput is low. However, the ABcast protocol discussed thus far has no flow-control, therefore as the number of requests per second increases, the protocol starts to become saturated by requests and performance deteriorates. If the broadcast rate of a node is not restricted in any network protocol, it is possible for a *congestive collapse* [36, 57] to occur. A congestive collapse is a situation whereby the current load on the network has saturated the underlying network, result-

ing in little to no throughput due to the rate of packet loss and the overall delay encountered by packets increasing. The increase in network delays is typically caused by the need for data packets to be queued in a buffer at both the sending and receiving node, whilst the increase in packet loss is attributed to the overflowing of said buffers; packet loss exasperates the problem as additional network traffic is required when the lost packet is retransmitted. For a congestive collapse to occur, its necessary for a network's input rate to exceed its output rate over an extended period of time, therefore in order to avoid such a collapse, it is necessary for the input rate of each node in the network to be throttled so that the network's average rate of input is approximately equal to its output rate.

Congestive collapse is the worse case scenario for the network, however it is possible for similar symptoms to temporarily manifest themselves if the input rate of one, or more, nodes' sporadically increases. Sporadic increases in a node's input rate is likely to cause large *bursts* of messages to be flooded into the network over a very short period of time, placing more stress on the buffers of both the sending and receiving nodes. This burstiness can cause the network to become temporarily congested, which does not lead to a total congestive collapse, however it is liable to cause increased packet loss and delays, resulting in a loss of throughput. Therefore it is important for a flow control protocol to not only ensure that a congestive collapse does not occur, but also to ensure that each node transmits data at a consistent and stable rate.

Due to its reliance on the PSM model and assumption **A4**, the ABcast protocol is more susceptible to the effects of network congestion than traditional deterministic protocols. As the underlying network starts to become congested, then it becomes harder for the DMC to produce even reasonably accurate estimates of *future* performance due to the variations in network delays and increased likelihood of packet losses. Whilst assumption **A3** ensures that all nodes will eventually receive a lost packet, it is still possible for an increase in packet loss to have an adverse effect on the overall system performance, as it is still necessary for the missing packets to be retransmitted by the reliable UDP protocol. Therefore additional packets are sent across the already congested network and the latencies recorded by the DMC will become larger. The unpredictability of a congested network can cause the DMC to underestimate the latencies that will be encountered by future transmissions, leading to assumption **A4** being compromised. Such an underestimation can result in more messages being delivered by Aramis, which in turn,

increases the probability that the *abcast* guarantess of ABcast will not be met due to messages being *rejected* from the total order.

Ultimately, a flow-control mechanism is required by ABcast nodes to ensure that the number of requests issued by a node, per second, does not adversely effect the performance of the underlying communication network. The P2P *abcast* protocol currently used by Infinispan, TOA, utilises a flow control scheme provided by JGroups, called UFC [68], to control each node's broadcast rate. UFC is based upon the *sliding-window* approach to flow control [6], however in the JGroups literature they describe the sliding window concept in terms of a finite number of *credits* that are maintained by each node; for completeness we utilise the same terminology when referring to UFC.

The UFC protocol is based on the premise that a node's credits are expended when a new broadcast is sent and reimbursed when a destination node confirms receipt of the original broadcast. Informally, the UFC protocol works as follows: A receiving node, N_j reimburses the sending node N_i's credits, by sending a response message to N_i with x amount of credits; where x is equal to the number of bytes received by N_j. If a sending node attempts to broadcast a message equal to y bytes, but its remaining credits $rc < y$ then the sending of a message m becomes blocked until a a receiving node reimburses the sender for its earlier broadcasts or a configurable timeout period expires.

The UFC approach works well for deterministic *abcast* protocols such as TOA, however it is not well suited for use with ABcast, due Aramis's reliance on the DMC's calculations for generating its probabilistic guarantees. As previously stated, the Aramis protocol is heavily reliant on assumption **A4**, consequently, it is necessary to ensure that the DMC's observed latencies do not fluctuate unpredictably in a manner that would undermine **A4**.

The UFC's independence from the DMC, ultimately means that it cannot determine whether the current load on the network is having an adverse effect on the latencies been measured by the DMC, and hence, UFC cannot take action in the event of the DMC's measurements deteriorating. Similarly, the UFC approach can become overly-restrictive when utilised by ABcast, as it is possible for a node to block the broadcasting of a message due to insufficient credits even if the DMC is operating as expected and no large increases in network latencies have been observed. Finally, the UFC protocol requires additional messages to reimburse each node's credit

when a broadcast has been received by a destination. This additional bandwidth requirement could alternatively be utilised by the underlying *abast* protocol to increase its throughput if it were not required by UFC.

Ultimately, a bespoke flow-control scheme is required by ABcast to ensure that optimal levels of throughput are maintained under heavy loads. Such a protocol should utilise the DMC's measurements to control the send rate of *abcast*s and to preserve the validity of assumption A4 where possible. The remainder of this chapter details the design and implementation of such a protocol.

4.5.1 AFC Design

In contrast to the UFC approach to flow control, our approach does not require additional messages, or the concept of a finite number of credits to restrict a node's transmission rate. Instead, our solution depends entirely on the latencies measured by the DMC and its associated calculations. Consequently, our flow control protocol is tightly coupled with the ABcast protocol and is not applicable for more traditional based *abcast* protocol such as TOA. The remainder of this section describes the rational behind our approach.

Assumption **A4** states:

> Let x_{mx} be the maximum delay estimated at time t by observing NT_P transmissions in the recent past: The delay x_{mx} will not be exceeded in any of NT_F, $NT_F \leq NT_P$, transmissions to unfold after t with probability $(1-q)$; where q can be estimated with reasonable accuracy.

Note: That $(1-q)$ can be estimated is a major assumption. For example, if actual q is 5% and is estimated as 25%, then this is not admitted by A4.

As previously stated, when ABcast nodes start to become overwhelmed by broadcasts it is possible for **A4** to be undermined, resulting in future transmissions exceeding x_{mx}. Therefore, we propose a new flow control protocol that utilises a *rate-based* scheme [6] that reduces a node's current broadcast rate if it starts to observe latencies greater than the last x_{mx} value calculated. We call this protocol ABcast Flow Control (AFC), and the basic design concept is as follows:

A node is sending and receiving broadcast messages between a fixed destination set of nodes and the latencies encountered by each broadcast is recorded by all recipient nodes[7]. In the event that one or more of these latencies exceed the current x_{mx} value, it is the responsibility of AFC to ensure that the local node adopts a lower broadcasting rate until a new x_{mx} value is calculated. At which time, the newly calculated x_{mx} takes into account the large latencies observed in the recent past, reducing the probability of assumption **A4** being compromised by the latencies encountered in the near future. If latencies continue to exceed x_{mx} then the node's broadcast rate will become increasingly restricted, whereas if no violations of x_{mx} occur, then no restrictions are placed on the node's broadcast rate. Thus a node's broadcast rate is restricted only when actual delays violate A4.

Unlike UFC, our approach restricts the sending rate of a node, N_i, based upon the messages it receives, not the rate at which N_i's messages are received at other nodes in the network. This may seem counterintuitive, but it is appropriate because the ABcast protocol is based upon the assumption that the latencies observed by a given node, N_i, are representative of the latencies which N_i's broadcasts experience at destinations, therefore the AFC protocol simply utilises this assumption and applies it to flow control. The AFC protocol is designed upon the assumption that if N_i's observed latencies repeatedly exceed x_{mx}, then it is highly probable that N_i's message buffer or the underlying network is approaching saturation. In which case, it is very likely that another node, N_j, will also be observing increased-latencies due to similar circumstances, therefore it is necessary for N_i's broadcast rate to be lowered in order to reduce the load on N_j. This assumption is especially apt in the *AmaaS* model when the SCast protocol is used, as each *c*-node randomly selects an *s*-node when sending a multicast request, resulting in client requests being evenly distributed between *s*-nodes. Therefore each *s*-node is likely to issue approximately the same number of *abcast*s, and thus, each node receives approximately the same number of messages.

Figure 4.7 shows the new ABcast components diagram with the AFC protocol included. Note that the AFC protocol is the highest in the stack, i.e. closest to the application, as all *abcast* messages must be sent via the AFC protocol.

The remainder of this section focuses on the calculations used by a node to regulate its broadcast rate.

[7] As required by the ABcast protocol.

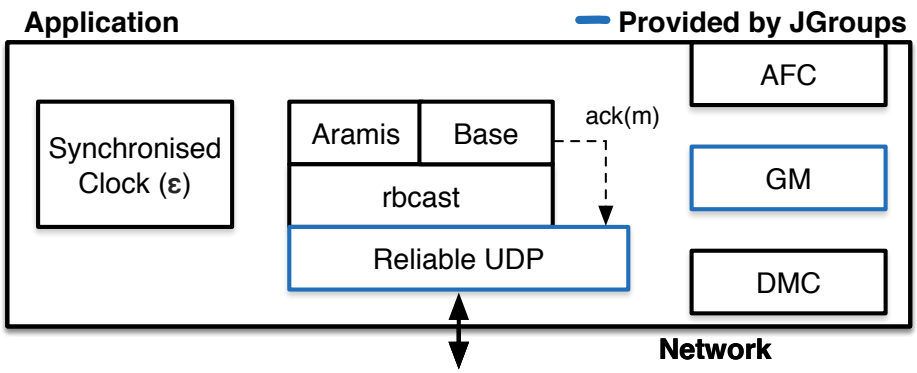

Fig. 4.7: ABcast Protocol Components with AFC

4.5.2 AFC Protocol

This section introduces the variables utilised by AFC, explaining their significance and why they are required, before detailing the calculations that utilise these variables to alter a node's broadcast rate. The calculations presented in this section assume that a single message *m* is being broadcast.

The steps required by AFC to initiate a broadcast of *m* are as follows: An application thread sends *m* down the protocol stack, and upon receipt of *m*, the AFC protocol performs all of the calculations presented in this section in order to calculate a flow control delay. This delay must be observed between the time of a node's previous broadcast and the broadcasting of *m*, with neither this thread or another application thread able to broadcast a subsequent message until *m* has observed its delay. Upon expiration of this delay, *m* can be sent to ABcast for broadcasting. Once *m* has been broadcast, the applications request has been completed and it is possible for the previously engaged thread to initiate a new broadcast. Note, it is possible for other application threads to submit messages to the AFC protocol whilst *m* is being handled, or waiting for its delay to expire; these messages cannot be processed until *m* has spent its required delay.

A common technique for implementing rate-based flow control, is the use of the *leaky bucket scheme* [6, 37] to regulate the sending of many messages. In this scheme, messages are placed into a 'bucket' until they are assigned a permit allowing them to be transmitted. The rate at which permits are dispensed to messages, determines the broadcast rate of the node and is equivalent to the estimated flow-control delay. It is common for leaky bucket implementations to utilise buckets that consists of several messages, with the transmission of the bucket being

delayed until all of its messages have acquired a permit; hence a flow-control delay has been observed by all nodes in the bucket. This leads to messages being sent in batches from each node, with a large delay observed between subsequent buckets, opposed to individual messages being sent at a consistent rate.

As implied earlier, the AFC protocol only utilises buckets consisting of one message, with each bucket observing a flow-control delay. AFC does not utilise buckets consisting of multiple messages due to ABcast's reliance on acknowledgement piggybacking and vector clocks. Sending messages in batches, means that ABcast is more likely to resort to sending explicit acknowledgement messages as *abcast*s will not be initiated frequently by a node. Instead large amounts of messages will be initiated over a short period of time, and in the interim period between consecutive buckets there will be no *abcast*s for acknowledgements to be piggybacked on.

Similarly, ABcast is reliant on regularly receiving vector clocks attached to *abcast*s from other nodes, to ensure that no messages in the total order have been missed. If *abcast*s were sent in batches, these clocks would not be received at a constant rate, rather they would be received in large batches semi-frequently. In this interim period a node would be more likely to violate the *abcast* total order as the node may not have received a vector clock from a given node for a relatively large period of time, and hence it would be unaware of some messages that were broadcast in the recent past.

Protocol Parameters and Calculations

When the AFC protocol receives m from the application, it polls the DMC to determine the number of latencies that have exceeded x_{mx} at the present time. These latencies are used to calculate the flow control delay for m, which ensures that m and subsequent messages, are broadcast at the newly calculated rate. Recall that the DMC utilises a $NT_F = 10\%$ of NT_P, where $NT_P = 1000$, and a new x_{mx} value is calculated after every NT_F latency has been recorded; in this case a new x_{mx} value is calculated after every 100 latencies observed. We consider a latency x to have exceeded x_{mx} if $x > x_{mx}$ and x is recorded after current x_{mx} is computed. When NT_F latencies have been recorded since the last calculation of x_{mx}, it is necessary for a new x'_{mx} value to be calculated that incorporates the latency values that previously exceeded x_{mx}.

When a latency x exceeds x_{mx}, we refer to this as a Marginal Peak Mp, as x_{mx} is the boundary (margin) value and Mp a latency that has peaked beyond the margin; we record Mp as the difference between x and x_{mx}, thus $Mp = x - x_{mx}$. It is possible that multiple x values will exceed x_{mx}, in which case we record all Mp values and refer to the total number of Mp values as $\#Mp$. Once all Mp values have been recorded, we calculate the variable μ; which along with the current x_{mx} value determines the amount that a node's broadcast rate should be restricted.

Let $\Sigma Mp = Mp + Mp' + , \ldots, + Mp''$, and μ be calculated as:

$$\mu = \frac{\Sigma Mp}{\#Mp} \tag{4.39}$$

However, if $\#Mp < 10$, μ is calculated as follows:

$$\mu = \frac{\Sigma Mp}{10} \tag{4.40}$$

Equation 4.40 is necessary, in order to reduce the effects of a small number of Mp values from severely restricting a node's broadcast rate. For example, consider $x_{mx} = 2ms$ and a single Mp occurs where latency x, $x = 4ms$, this would result in $Mp = 2ms$. If equation 6.1 was utilised, then $\mu = 2$, which would produce a large flow-control delay, which in turn would result in the broadcast rate being reduced significantly and the flow-control becoming overly restrictive. However, when we utilise Equation 4.40, the influence of a small number of latencies (< 10) on the calculated μ is reduced. In this example, and our implementations of AFC, we have utilised 10 to define the minimum divisor required when calculating μ, however this value can be configured depending on a system's requirements.

The μ variable is used alongside x_{mx} to calculate γ, where γ is calculated as:

$$\gamma \quad = \quad \frac{x_{mx} + \mu}{x_{mx}} \quad = \quad 1 + \frac{\mu}{x_{mx}} \geq 1 \quad as \quad \mu \geq 0 \tag{4.41}$$

Calculating a New Broadcast Rate

Let λ_1 represent the current broadcast rate of a given node. It is used alongside γ to calculate the new broadcast rate and the duration of the flow-control delay that needs to be observed by m.

Let λ_2 denote the new rate which is calculated as:

$$\lambda_2 = \lambda_1 e^{\left(\frac{1-\gamma}{C}\right)} \tag{4.42}$$

Where $C > 0$ is an input parameter for AFC.

$\gamma = 1$

When $\gamma = 1$, no Mp value has been observed so far; so, $\mu = 0$ as per Equation 4.40 and $\lambda_2 = \lambda_1$. Thus, m will be handed over to ABcast at the old rate of λ_1.

$\gamma > 1$

When $\gamma > 1$, x_{mx} has been observed to have been violated and hence the broadcast rate must be slowed down. Moreover $(1-\gamma) < 0$, $e^{\left(\frac{1-\gamma}{C}\right)}$ and $\lambda_2 < \lambda_1$.

Note that $e^{\bar{z}}$ decays rapidly as $\bar{z}$, $\bar{z} > 0$, decreases even by a small amount (see Figure 4.8). To control λ_2 becoming extremely small, we choose C to be a large positive constant.

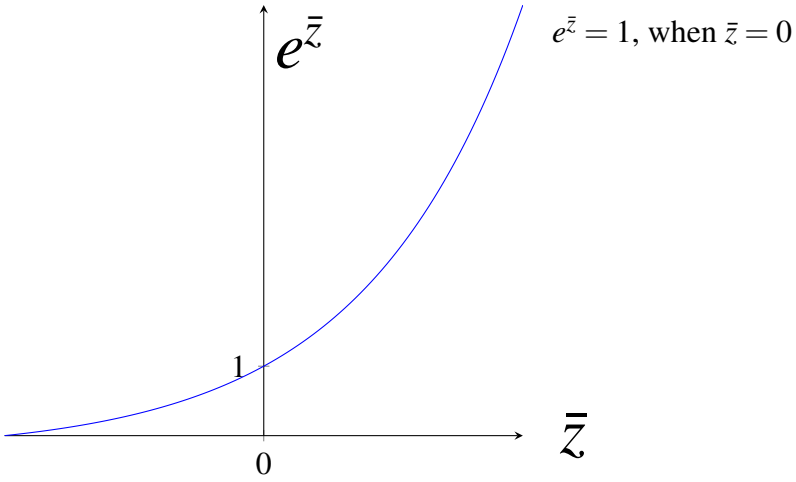

Fig. 4.8: The effect of $\bar{z}$ decreasing on $e^{\bar{z}}$

Honouring λ_2 for m, means that AFC must maintain the time at which it handed over a message to ABcast most recently. Let $\tilde{m}$ be that message, and $\tilde{m}.t$ be the time $\tilde{m}$ was handed over to ABcast. Note that $\tilde{m}.t$ is registered as per a node's unsynchronised clocks and bears no relevance to $\tilde{m}.ts$ that ABcast would timestamp $\tilde{m}$ with. Let $m.t = \tilde{m}.t + \frac{1}{\lambda_2}$ and AFC hands over m to ABcast when the (unsynchronised) clock reads $m.t$ and retains $m.t$ instead of $\tilde{m}.t$.

The additional delay m experiences over $\tilde{m}$ is given by δ:

$$\frac{1}{\lambda_1} + \delta = \frac{1}{\lambda_2} \tag{4.43}$$

Note: $\delta = 0$ if $\lambda_1 = \lambda_2$.

A Practical Observation and Remedy

AFC presented above is very dynamic and can result in a node's δ value fluctuating dramatically over a period of time. Although a dynamic flow-control solution that reacts to the changing conditions of the network is desirable, early experiments showed that calculating δ as above resulted in very small delays being calculated for the majority of broadcasts. This resulted in the broadcast rate of sending nodes not being restricted sufficiently over an extended period of time, causing larger numbers of Mp values to suddenly appear. This sudden emergence of Mp values results in the calculated δ value being extraordinarily high and the system's throughput becoming excessively restricted. Over time, as new x_{mx} values were calculated and Mp values stopped appearing, the flow-control began to increase the node's broadcast rate, however we found that δ eventually become too small again, causing a cycle to occur that consistently repeated itself. Ultimately, solely utilising δ caused AFC to retrospectively react to a congested network, when the purpose of AFC is to be proactive and stop congestion from occurring.

Our solution, was to propose two new constants δ_{min} and δ_{max}, which set a lower and upper bound on δ. The new calculations for λ_2 are presented below:

δ_{min}

If

$$\frac{1}{\lambda_1} - \frac{1}{\lambda_2} < \delta_{min}$$

then

$$\lambda_2 = \lambda_1$$

δ_{max}

If

$$\frac{1}{\lambda_1} - \frac{1}{\lambda_2} > \delta_{max}$$

then λ_2 is recomputed as

$$\frac{1}{\lambda_2} = \frac{1}{\lambda_1} + \delta_{max}$$

The purpose of the lower bound δ_{min} is to ensure that all broadcasts are sent at a constant, predictable rate, in order to stabilise x_{mx} and ensure that the system does not become excessively *bursty*. Conversely, the upper bound δ_{max} ensures that if a large number of Mps occur between x_{mx} calculations, the calculated δ value will not be excessively large and thus wont overly restrict a node's broadcast rate. Like the other constants used by AFC, δ_{min} and δ_{max} are determined before runtime, and therefore appropriate values can be set for each depending on the network environment and the expected throughput of data.

4.5.3 Limitations

The AFC protocol detailed in this section has proved to be an effective flow-control protocol for use with ABcast, with the broadcast rates of nodes in the cluster being restricted sufficiently to prevent congestive collapse and minimise Aramis rejections (§ 6). However, the AFC has two key limitations:

i The protocol is a rate-based scheme and therefore the limitations inherent with such an approach can be attributed to AFC by default. Specifically, such an approach does not guarantee that no buffer overflows will occur as it is possible that the calculated rate of broadcast is too large. This is especially true if the conditions of the network change dramatically over a short period, and in the case of ABcast this will also effect our Δ_m calculations.

ii The use of δ_{min} and δ_{max} as upper and lower bounds on delays. These bounds are required to ensure that the system is not overly restrictive, or permissive, of *abcast* requests, however in the current design these values are specified before run-time and do not change based upon the networks current condition. Therefore, it is possible that δ_{min} may be overly restrictive when the system is lightly loaded, whereas δ_{max} may permit too many broadcasts to be sent when the system is heavily loaded.

4.6 Summary

This chapter presented ABcast - a new hybrid protocol that utilises both a deterministic (Base) and probabilistic (Aramis) protocol in order to create a non-blocking *abcast* solution. We detailed the protocol's assumptions and required components, before detailing the delivery and rejection criteria of the two protocols. Finally, we presented a new flow-control protocol designed specifically for use with ABcast.

Chapter 5

Probabilistic SCast

This chapter explores the consequences of utilising the ABcast protocol for state machine replication between *s*-nodes in the AmaaS model. Throughout our explanations we assume that the SCast protocol and its system model are utilised for all interactions between *c*-nodes and the ordering service. We refer to this approach as Probabilistic SCast, or PSCast for short.

The remainder of this chapter is structured as follows: First we present the new *amcast* guarantees provided by PSCast. We then explore the ramifications of these new guarantees and discuss how they can cause *amcast* messages to be delivered out of the total order at one or more destinations. Finally, we describe the consequences of such missorderings, within the context of Infinispan, and explore potential solutions that can be employed by Infinispan to tolerate these missorderings.

5.1 PSCast Guarantees

Below, we state the *amcast* guarantees provided by PSCast.

G1 - *Validity*: If the source of m_i does not crash until it *abcast*s m_i, then all operative destinations of m_i deliver m_i.

G2 - *Uniform Agreement*: If the source of m_i crashes while *abcast*ing m_i, and if any destination delivers m_i, then all operative destinations of m_i must deliver m_i.

G3 - *Uniform Integrity*: If m_i has already been delivered by a destination d, then d cannot deliver m_i again.

G4-PSCast - *Probabilistic Total Order*: If two amcasts, m_i and m_j, have common destinations, then all such destinations that deliver both m_i and m_j, will deliver them in an identical order with a probability *Rmc*; where $(1 - Rmc)$ is linearly proportional to $(1 - R)$ and R is the order probability of ABcast.

Note: The guarantees of PSCast are identical to those of the underlying *abcast* protocol, ABcast. This is because the guarantees of the *abcast* protocol, and any violations of these guarantees, directly impacts how each *s*-node maintains its $order\_history[]$ and generates $m.history[]$; with $m.history[]$ dictating the order in which a *c*-node must deliver *amcast*s (§ 3.4).

5.2 G4-PSCast Implications

Recall that the SCast protocol utilises the final timestamp of the *abcast* message, which contains a clients ordering request, to determine the total order of the *amcast*.

The SCast protocol, described in chapter 3, assumes that the underlying *abcast* protocol provides deterministic guarantees for all G1-G4 (§ 2.2). Hence, the SCast protocol assumes that all *s*-nodes will eventually deliver an *abcast* and that all *abcast*s are delivered in the same order at each destination. This assumption means that the SCast protocol is able to guarantee that all *amcast*s sent via the ordering service will respect G4 as it is guaranteed that all *s*-nodes will always maintain a consistent $order\_history[]$. Therefore, all ordering responses, $rsp(Tx)$, from the service will contain the correct $m.history[]$ data.

Conversely, PSCast utilises a probabilistic *abcast* protocol, ABcast, that only guarantees G4 with probability *R* (G4-P). Therefore, for an *abcast m* sent between *s*-nodes, there is a small probability $(1 - R)$ that a destination, N_s, will neither receive nor *know* of *m* before Δ_m time. In this case, it is possible for another *abcast*, m', $m.ts < m'.ts$, sent from a different *s*-node, $m.o \neq m'.o$, to have been delivered at N_s when Δ'_m expired. Resulting in a violation of G4, as m' has preceded *m*, which causes N_s to *reject m* from the total order and deliver it to the application (SCast), via an exception, when it eventually arrives.

5.2.1 An Abstraction Based Explanation

The mechanisms that give rise to probabilistic total order of *amcasts* can be abstracted by modelling *abcast* total order failures as random choices occurring with probability $\leq (1-R)$. Suppose, to the contrary, that ABcast meets G4 deterministically; *i.e.* all *abcast*s are delivered only by Base. Note that this supposition allows us to use $\lll$ and $\prec$ defined earlier. Suppose also that each service node Ns_i, upon delivering an *abcast* m, $m.o \neq Ns_i$, generates a random number RV, which is distributed uniformly on $(0,1)$ and carries out one of the two actions.

Case 1: $RV \leq R$ or $m.o = Ns_i$

This represents the case that G4 is met and m is processed exactly as explained in the deterministic SCast protocol (§ 3.4.3). In particular, m is added to a node's $order\_history[]$ instantly as described in SCast.

Case 2: $RV > R$

This represents the case where G4 was not met. In this case, m is not inserted in $order\_history[]$ until after x time elapses. With x being a random value that represents the unknown time at which m would be delivered late by Aramis. Consequently, subsequent *abcast* messages, m', $m'.ts > m.ts$, that are delivered by Ns_i without any additional random wait of x would be inserted into $order\_history[]$ ahead of m. When the random wait period expires, Ns_i "detects" m to have disrupted the ideal $\lll$ supposed earlier and handles it in an exceptional manner.

Remark

Had $m.o = Ns_i$, then Ns_i cannot deliver its own *abcast* late, hence Case 1 would apply. This also means that when $m.o \neq Ns_i$ and Ns_i applies Case 2 for m, there will be another service node $Ns_j = m.o$ which, unless crashed, applies Case 1 on m and converts it into its corresponding *amcast*. In other words, that $Ns_i \neq m.o$ chooses Case 2 for m does not necessarily imply that the corresponding *amcast* to client nodes is unduly delayed. Next, we discuss the consequences of $Ns_i \neq m.o$ not promptly entering m into its $order\_history[]$.

Consider a scenario where Ns_i delivers four *abcast*s, $\{m_1, m_2, m_3, m_4\}$; each m corresponds to a transaction with the same value, *e.g.* $m_1 \rightarrow Tx_1$, and $Tx_1.dst = \{N_c1, N_c2, N_c3\}$; $Tx_2.dst = \{N_c1, N_c3\}$; $Tx_3.dst = \{N_c1, N_c2\}$; $Tx_4 = \{N_c2, N_c3\}$, and $m_4.o = Ns_i$. Furthermore, each

$mcast(Tx_1)$ message sent as per the SCast protocol, is denoted as mc_1 for short, *e.g.* mc_1 is associated with the same *amcast* as Tx_1 and m_1.

If Ns_i delivers all four messages in the correct order, all messages are inserted into the $order\_history[]$ in the correct order. Therefore, a correct $m_4.history[]$ is produced by Ns_i; the resulting $order\_history[]$ and $m_4.history[]$ generated upon delivery of m_4 are shown in Figure 5.1.

Alternatively, consider that a single *abcast*, m_3,$m_3.o \neq Ns_i$, is allocated a value $RV > R$. This causes m_3 to be delayed for x time. Assuming that m_4 is delivered before x time, m_4 will take m_3's place in the total order, resulting in an erroneous $order\_history[]$ and $m_4.history[]$. This is shown in Figure 5.2.

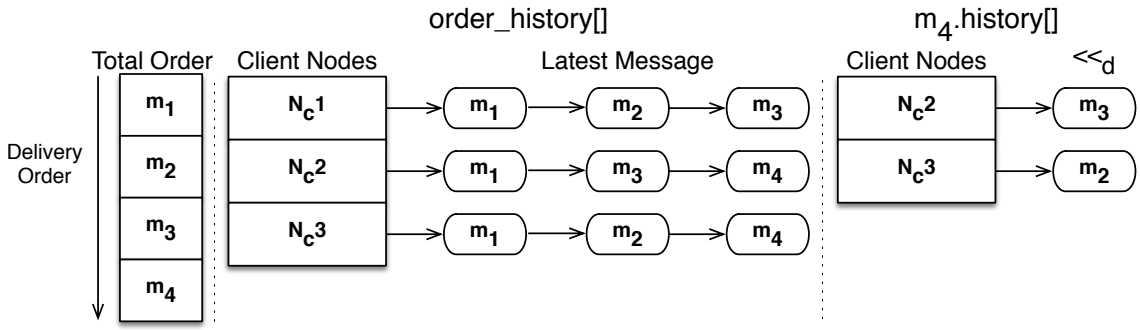

Fig. 5.1: $order\_history[]$ with Correct ABcast Ordering (G4)

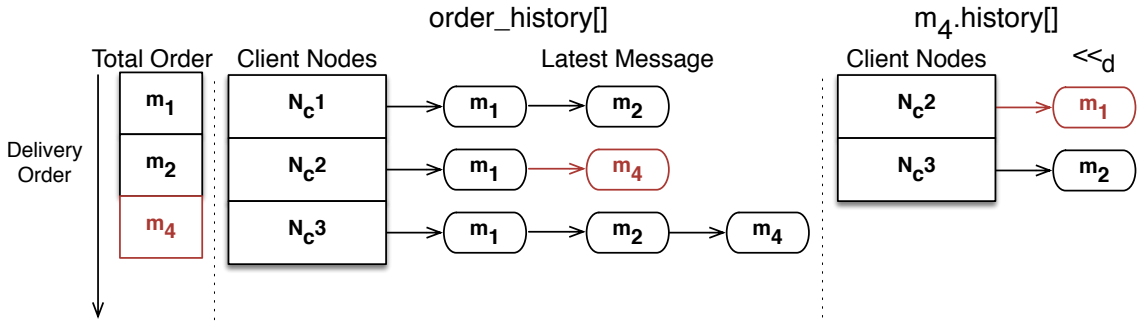

Fig. 5.2: $order\_history[]$ with ABcast Order Failure (G4 Violation)

The consequences of the incorrect $m_4.history[]$, as shown in Figure 5.2, are that it is now possible for the c-node, N_c2, to miss mc_3 in its total order, even though $mc_3 \lessdot_{N_c2} mc_4$. In order for such a miss-ordering to occur, both of the following must be true:

i N_c2 ***receives*** mc_4 before mc_3.

ii N_c2 ***delivers*** the message stated in $mc_4.history[N_c2]$ before receiving mc_3.

If condition (i) is true, then N_c2 will not know of mc_3 as it is not stated in $mc_4.history[]$ and if (ii) is also true, then mc_4 will be delivered immediately. Note, that if N_c2 was to receive mc_3, before delivering $mc_4.history[N_c2]$, then mc_3 will be *known* by Nc_2 and the correct delivery order of $mc_1 \lessdot_{N_c2} mc_3 \lessdot_{N_c2} mc_4$ will be preserved.

5.3 Service Node - Coping with ABcast Order Violations

We now explore how ABcast order violations can be handled by *s*-nodes in order to reduce the chances of SCast violations from occurring.

Let m and m' be *abcast*s such that $m \lessdot_d m'$ for all $d \in tx.dst \cap tx'.dst$; where tx and tx' are the transactions associated with *abcast*s m and m' respectively. Now consider that m is assigned a value $RV > R$ and m' $RV \leq R$ at an *s*-node N_s. This results in N_s delivering m' before delivering m via an exception.

In the above scenario, N_s's $order\_history[]$ will be missing m when $m'.history[]$ is calculated, therefore it is *possible* for one or more d to suffer from an SCast ordering violation. Upon delivering m, it is not possible for N_s to update $order\_history[d]$, as $order\_history[d]$ contains $m'.ts$ which is more recent than $m.ts$, $m.ts < m'.ts$, and $order\_history[]$ must only store the latest timestamp encountered for a given d. However, for all $d' \in tx.dst \setminus tx'.dst$, it is necessary for $order\_history[d']$ to be updated if $order\_history[d']$'s current entry precedes m in the total order. Updating all $order\_history[d']$ entries ensures that the latest timestamps are recorded where possible, therefore reducing the chances of further SCast ordering violations.

Crash-Tolerance

In the scenario (i) described above, it is not possible for m to create a correct $m.history[]$ as m' will be present in $order\_history[d] \forall d \in m.dst \cap m'.dst$. However, an accurate $m.history[]$ needs to be generated in order to store a local copy of m, at N_s, as it may be required in the event of a crashed node (§ 3.4.4 ($C3$ and $S1-S4$)). A possible solution to this problem, is to utilise a partially-persistent version of $order\_history[]$, $\bar{order\_history}[]$, that allows

past values associated with a given c-node to be queried [1]. This would enable N_s to query $\bar{order\_history}[d] \forall d \in m.dst$ and find the message that actually precedes m in the total order, not m', so that an accurate $m.history[]$ can be generated.

A limitation of such an approach would be the amount of memory consumed by the $\bar{order\_history}[]$ data structure, however this is not a major concern as the amount of data required for each message is very small. Therefore, utilising an eviction scheme alongside reusable data objects would allow such a scheme to function indefinitely whilst storing thousands of message records at any one time.

5.4 Client Nodes - Detecting SCast Order Violations

Recall that SCast utilises a single *delivery* thread to process the AWQ and *am_deliver*() all *amcast* messages (*mcast*()) to the application. In addition to these threads, we also assume there exists several *worker* threads which are responsible for delivering and processing unicast messages to the SCast protocol. It is these threads that process unicast messages and determine whether they are *mcast*() messages which must be added to the AWQ or if they are unicast messages required by higher-level protocols in the stack; in which case, the *worker* threads simply forwards the messages up the stack [2].

To detect order violations, we modify the behaviour of the original SCast protocol. Instead of placing all received *mcast*() messages into the AWQ, it is now necessary for the *worker* thread to inspect the *mcast*() message, m, and determine whether m has missed its place in the total order. This is achieved by comparing $m.order$ with $last\_amcast.order$; where $last\_amcast$ is the last *mcast*() message that invoked *am_deliver*(). If $last\_amcast.order \prec m.order$, then m can be added to the AWQ and processed as normal. However, if $m.order \prec last\_amcast.order$ then we know that m has been missed in this node's total order, therefore it is necessary for the primitive *am_deliver_with_exception*() to be invoked.

The purpose of *am_deliver_with_exception*() is to enable an the application to still receive m, whilst being aware that an order violation has occurred. Assuming that the primitive has been invoked for a *mcast*() m, *am_deliver_with_exception*() works as follows: m is assigned

[1] This could be implemented by maintaining a partially-persistent linked list for each client, with each subsequent message associated with a client simply appended to the end of the list.

[2] The *delivery* thread is not required as normal unicast messages do not adhere to a total order.

an error bit to identify it as an order violation and it is delivered to the application via a *worker* thread, *i.e.* as a normal unicast message. The *delivery* thread is not utilised as this node's total order has already been violated and it is necessary for the higher level application to receive m as soon as possible.

For example, let *amcast*'s $m_1 \prec_d m_2$ for a c-node destination d. Now assume that an ABcast order violation occurred at an s-node when processing m_1, which results in $m_2.history[]$ missing m_1 for $m_2.history[d]$. Let d receive and deliver m_2, before eventually receiving m_1; hence $last\_amcast = m_2$. Upon receiving m_1, d will realise that $m_1 \prec last\_amcast$ and will therefore invoke $am\_deliver\_with\_exception(m_1)$. Assuming that the application is still executing m_2 using the *delivery* thread, it is possible for the application to receive m_1 via a *worker* thread and initiate a recovery mechanism as soon as possible, in order to counteract m_1's SCast order violation.

The pseudocode for receving a message and determining whether a $mcast()$ has suffered a SCast order violation at a destination d is presented in Algorithm 4, whilst the pseudocode for processing the AWQ remains the same as Algorithm 3 presented in section 3.4.

Algorithm 4 PSCast Receive Message

```
1:  m ← receive_message();
2:  if m is mcast() then
3:      if last_amcast ≺_d m.order then
4:          add_to_AWQ(m);
5:      else if m.order ≺_d last_amcast then
6:          am_deliver_with_exception(m);
7:      end if
8:  else
9:      forward_to_application(m);
10: end if
```

5.5 Infinispan (Client Node) - Coping with SCast Order Violations

In the event that one or more order violations occur, it is necessary for the client application to be able to recover from such events. In this subsection, we present a recovery mechanism that can be employed by the Infinispan transaction manager at each *c*-node in the event of a violation occurring. The exact methods of the recovery mechanism are determined by the isolation level chosen by the Infinispan clients before runtime; with one method required for *Repeatable Read*, *RR* and *Read Committed*, *RC* transactions, whilst an alternative method is required for *RR* with WSC due to its dependence on a voting phase to commit/abort transactions. Therefore, we first explore the method required for *RR* and *RC* before detailing the provisions for *RR* with WSC.

5.5.1 Transaction Manager Assumptions

Our solutions for recovering from ordering violations assume that the following data structures are implemented by the Transaction Manager (TM):

tx_entry

For a transaction, Tx, the following data must be stored:

- The duration of the transaction from its inception. ($\boldsymbol{Tx.duration}$).
- A boolean flag, which is set to true if Tx was delivered to the TM via an exception. ($\boldsymbol{Tx.exceptional}$).
- The value of all (k, v) pairs in this Tx at the time Tx starts to be processed. ($\boldsymbol{Tx.start\_pairs}$).
- The *m.order* value associated with the *amcast* which delivered Tx's $prepare(Tx)$ message. ($\boldsymbol{Tx.order}$).
- The final decision, whether to *abort* or *commit* the transaction. ($\boldsymbol{Tx.decision}$).
- An associative array of all $commit()$ or $abort()$ votes received for this transaction, with the source node of a vote being used for indexing (*RR* with *WSC only*). ($Tx.votes[]$). **Note:** votes received for transactions that have not yet been processed by a TM are stored in a temporary buffer until the associated transaction becomes active; at which point the votes are *received* and added to $Tx.votes[]$.

Henceforth, all references to a transaction, or Tx, assumes that a *tx_entry* is maintained along with all of the above values.

tx_queue

The transaction queue, *tx_queue*, is a priority queue which stores the TM's currently executing transaction, Tx, as its head entry; a Tx is dequeued when it has finished executing and pushed onto *tx_history*. Subsequent entries in the queue are additional transactions, Tx' that succeed Tx in the total order. **Note:** When no order violations have occurred, the *tx_queue* should only contain Tx.

tx_history

The transaction history, *tx_history*, is a stack which stores the transactions which have already been committed by the TM, with the Tx stored at the top of the stack representing the last transaction to have been committed by the TM.

Delivery Thread and TM

As the *delivery* thread utilised by PSCast is responsible for delivering all *mcast*() messages that do not suffer an order violation, we assume that this thread is also responsible for processing the *tx_queue* upon message delivery. Therefore, transaction processing occurs as follows: At the PSCast level when *am_deliver*() is invoked, the *delivery* thread passes the $prepare(Tx)$ message upto the TM, adds the Tx to *tx_queue* and starts processing *tx_queue*; hence Tx. Once all messages in the *tx_queue* have been processed, the *delivery* thread returns to the PSCast level and starts processing the next message in SCast's AWQ.

Worker Threads and TM

We assume that all unicast messages sent between TMs are delivered up their destination's network stack and to their local TM via a *worker* thread. This enables the TM to receive unicast messages from other TMs, *e.g.* votes when WSC is enabled, whilst enabling the *delivery* thread to be utilised solely for *amcast*s that require processing by the TM.

5.5.2 Repeatable Read and Read Committed

When *RR* or *RC* isolation is utilised the outcome of a transaction is determined in a single phase as each $d \in Tx.dst$ deterministically decides whether to commit or abort a transaction without requiring any additional communication between destinations. Consequently, when a TM receives a $prepare(Tx)$ message via PSCast, assuming that no ordering violations occur, the TM is able to commit the transaction immediately.

Now consider that an *amcast* $prepare(Tx)$ message, m_i has suffered an order violation and a subsequent *amcast* m_j has been delivered to the TM before m_i. In such a scenario, the TM remains unaware of m_i until $am\_deliver\_with\_exception(m_i)$ has been called by the PSCast layer below and m_i is received via an exception. Therefore, until TM receives such an exception, the transaction associated with m_j and any other m delivered ahead of m_i will be processed as if no ordering violation has occurred.

When m_i is delivered to the TM via an exception, the only course of action is for a roll-back procedure, similar to that utilised in compensating transactions [44], to be initiated; with all transactions that should have been executed after m_i's transaction being reversed. Once all of the offending transactions have been reversed it is possible for m_i's transaction to be executed and the previously rolled back transactions reapplied in the correct total order. Hence, the data maintained by this *c*-node eventually becomes the same as if no ordering violation had occurred.

For example, if the original value of a key k was $v = 5$, the transaction associated with m_i was $update(k, 10)$ and m_j's transaction was $update(k, v+1)$. When m_i is missing, the outcome of m_j's transaction would be $v = 6$, however after m_i is delivered via an exception and the transactions are re-executed, the result of m_j's transaction will be $v = 11$.

Note: When using *RR* or *RC* isolation it is not possible for order violations to be resolved before a transaction commits as both of these isolation levels utilise a single phase commit protocol. In order for order violations to be resolved before committing a transaction it is necessary for multiple phases to be used, therefore if an application cannot tolerate commits being rolledback, *RR* with *WSC* isolation should be utilised (§2.5.3.2 and §5.5.3).

TM roll-back Implementation

Assume the same scenario as described in the previous section, with m_j being incorrectly delivered to the TM ahead of m_i; where m_j and m_i are associated with transactions Tx_j and Tx_i respectively. When a destination eventually receives m_i, it invokes $am\_deliver\_with\_exception(m_i)$ and delivers m_i to the TM using the *worker* thread which delivered m_i to the PSCast layer. Upon receiving m_i, and hence Tx_i, the TM takes the following actions:

1. Tx_i is stored within the TM as a *tx_entry* and inserted into the *tx_queue*. An interrupt is sent to the *delivery* thread so that it is aware that at least one Tx exists in the queue.

2. The *delivery* thread finishes executing its current transaction before processing the next Tx in the *tx_queue*; which will most likely be Tx_i [3].

3. During the execution of Tx_i, past transactions stored in *tx_history* are *popped* from *tx_history* and inserted into the *tx_queue* (in order), until no transactions exist in *tx_history* which succeed Tx_i in the total order.

4. Once all of the required transactions have been (re-)inserted into *tx_queue*, the TM will execute Tx_i and then push Tx_i onto *tx_history*. This process continues until the *tx_queue* becomes empty, at which point all transactions that are known by the TM will have been executed in the order dictated by PSCast.

5. Once the *tx_queue* becomes empty, the roll-back procedure is considered complete and the *delivery* thread is able to return to the PSCast level to start processing the next *amcast* message in the AWQ.

Figure 5.3, shows the process of a transaction tx_3 being executed and then pushed onto *tx_history*. Note how tx_2 is missing from *tx_history* as it was missed in the total order. Figure 5.4 shows the roll-back procedure being executed once tx_2 is delivered to the TM via an exception, with tx_3 being popped from *tx_history*, as it was erroneously executed ahead of tx_2, and reinserted into the *tx_queue* in its correct order behind tx_2.

Algorithm 5 outlines the steps required by the TM to process the *tx_queue* when Repeatable Read or Read Committed isolation is utilised by transactions. In this algorithm, we assume that the transaction associated with the interrupting exception has already been added to *tx_queue*.

[3] Assuming only one ordering violation occurs, Tx_i will always be inserted at the head of the queue.

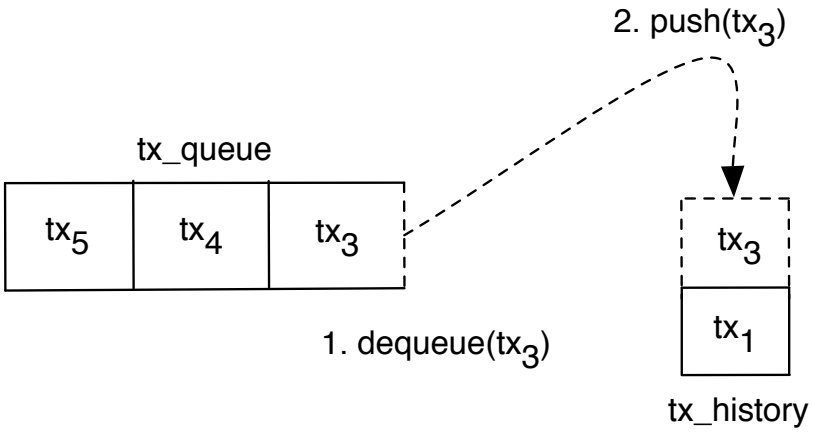

Fig. 5.3: PSCast: TM *tx_queue* and *tx_history*

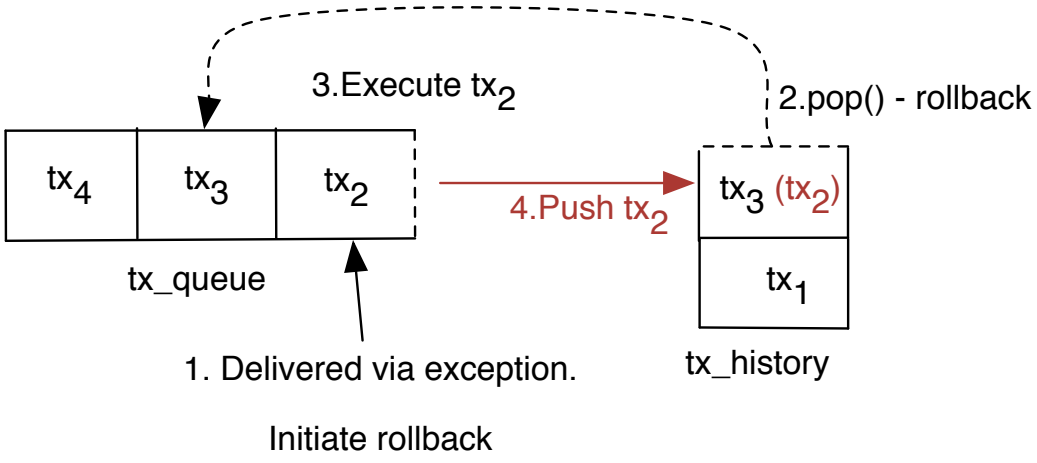

Fig. 5.4: PSCast: TM *tx_queue* and *tx_history* Executing a Roll-back

Limitations

A consequence of utilising a roll-back mechanism, with *RR* or *RC* isolation, is that 'stale' reads can occur as it is possible for a subsequent transaction, to read $v = 6$ when requesting the value of k, ultimately leading to *write-skew*. We consider this an acceptable risk for three reasons:

i Ordering violations are already the product of several small probabilities, therefore the probability of *write-skew* occurring is even smaller.

ii The window of opportunity for a *write-skew* to occur is very small. We anticipate that the time between an ordering violation occurring and being detected to be in the order of milliseconds.

iii The Infinispan store is already susceptible to *write-skew*s when the WSC is not utilised,

Algorithm 5 TM *tx_queue* Processing for RR and RC Isolation

```
1:  while tx_queue is not empty do
2:      tx ← dequeue(tx_queue);
3:      previous_tx ← pop(tx_history);
4:      while tx ≺ previous_tx do
5:          insert_into_tx_queue(previous_tx);
6:          previous_tx ← pop(tx_history);
7:          revert_commit(previous_tx);
8:      end while
9:      if previous_tx is not null ∧ previous_tx ≺ tx then
10:         push_to_tx_history(previous_tx);
11:     end if
12:     tx.decision ← process(tx);
13:     if tx.decision ≠ abort(tx) then
14:         push_to_tx_history(tx);
15:     end if
16:     execute(tx);
17: end while
```

therefore the business logic of the application utilising Infinispan should already be tolerant of such phenomena.

5.5.3 Repeatable Read with WSC

When utilising transactions with *RR* and WSC it is not possible to just utilise the roll-back procedure outlined in section 5.5.2, as the outcome of each transaction affects the second voting phase that is required to avoid *write-skews*. However, the additional voting stage required by the WSC can be used to our advantage to prevent ordering violations from affecting the consistency of Infinispan's key/values.

Recall that for every key stored in the infinispan system there exists a single *primary* replica and at least one *backup* replica. Thus, the WSC requires that a transaction coordinator, $Tx.c$, receives at least one commit vote for each of the distinct keys involved in a Tx in order for it to be able to send a final $commit(Tx)$ decision to all $Tx.dst$. Whereas, $Tx.c$ only requires a single abort vote from any of the $Tx.dst$ members in order to disseminate an $abort(Tx)$ decision. The solutions presented in this section require a slight modification to this existing WSC behaviour. Our solutions require that:

> For all transactions, votes to commit or abort a transaction can only be sent by the client node which contains the *primary* replica of a key.

Again consider that an *amcast* $prepare(Tx)$ message, m_i has suffered an order violation and a subsequent *amcast* m_j has been delivered to the TM before m_i. As is the case for RR and RC isolation, the TM only becomes aware of m_i once $am\_deliver\_with\_exception(m_i)$ has been called by the PSCast layer. Upon receiving m_i, its associated transaction, tx_i is inserted into *tx_queue*. In the event that the *delivery* thread is currently processing another Tx, tx_i is not processed until Tx has finished executing.

TM Classification

How each Tx stored in *tx_queue* is processed, depends entirely on the (k,v) pairs involved in an individual Tx and the relationship of the TM with each of these pairs. Below we define three classifications of TM with respect to an individual Tx:

Primary TM

We classify a TM as being a *primary* for a given Tx, if this node hosts a *primary* replica of at least one (k,v) pair that is modified by this transaction. Read requests do not count, as these actions have already been performed before the $prepare(Tx)$ message was *amcast*.

Backup TM

A TM is classified as a *backup*, if it only hosts *backup* replicas of the (k,v) pairs involved in a Tx. A TM cannot be a *backup* if it hosts a *primary* replica of any of the (k,v) in the Tx.

Coordinator TM

A TM is classifed as the *coordinator*, if the node hosting the TM was the original node who initiated Tx, *i.e.* $Tx.c$, or if this node is the currently active coordinator for the Tx, *i.e.* $Tx.\tilde{c}$.

Note: It is possible for a TM to be both a *coordinator* and a *primary* or a *coordinator* and *backup*, simultaneously. However, by definition it is not possible for a TM to be both a *primary* and a *backup*.

Processing a Transaction

Once a Tx becomes the head of *tx_queue*, and hence the active transaction, it is passed to the function $process(Tx)$ which determines whether Tx should be committed or aborted. The actions taken by the TM when executing $process(Tx)$ depends on its classification, therefore we present the actions of each classification below:

***Primary* TM**

A *primary* TM takes the following actions for a Tx:

1a. If Tx was delivered via exception, *i.e.* $tx.exceptional$ is true, then send an abort $vote(Tx)$ to $Tx.c$

1b. Otherwise, the outcome of the write skew check determines the vote to be sent to $Tx.c$, either $commit(tx)$ or $abort(tx)$.

2. Wait to receive the final decision from the $Tx.c$.

***Backup* TM**

As *backup* TM does not vote towards the progress of a transaction, it simply waits to receive a final decision for Tx from the $Tx.c$.

***Coordinator* TM**

If a *coordinator* is also a *primary* then it is necessary for the *coordinator* to first cast a vote as required by the *primary* TM role before executing the actions required by a *coordinator*. A *coordinator* must take the following actions:

1. Wait to receive a vote from all *primary* TMs associated with Tx.

2a. If a single $abort(Tx)$ vote is received, then stop waiting and send an final $abort(Tx)$ decision to all $d \in Tx.dst$.

2b. Otherwise, if a $commit(Tx)$ is received from all *primaries*, send a final $commit(Tx)$ decision to all $d \in Tx.dst$.

Note: If a single $abort(Tx)$ vote, or all $commit()$ votes, are not received after a pre-determined timeout period then the *coordinator* aborts Tx. This abort timeout is necessary

to overcome a deadlock scenario which, although unlikely, is possible with our solution. This deadlock scenario is explored in detail later on in this section.

Once a Tx has been processed and a final decision established, it is necessary for it to be removed from *tx_queue* and the decision enacted. If the Tx is to be committed, it must also be pushed to *tx_history*. At this point the execution of Tx at the local TM is complete, and the next Tx' in *tx_queue* can be processed.

Algorithms 6 and 7 present the pseudocode for processing *tx_queue* and executing $process(tx)$, respectively, when WSC is enabled. The purpose of lines 5-15 in Algorithm 7 is described in the next section.

Algorithm 6 Transaction Manager RR with WSC *tx_queue* Processing

```
1:  while tx_queue is not empty do
2:      tx ← peek(tx_queue);
3:      tx.decision ← process(tx);
4:      if tx.decision ≠ abort(tx) then
5:          if Not a primary for any (k, v) in tx then
6:              previous_tx ← pop(tx_history);
7:              while tx ≺ previous_tx do
8:                  insert_into_tx_queue(previous_tx);
9:                  previous_tx ← pop(tx_history);
10:                 revert_commit(previous_tx);
11:             end while
12:             if previous_tx ≺ tx ∧ previous_tx is not null then
13:                 push_to_tx_history(previous_tx);
14:             end if
15:         end if
16:         push_to_tx_history(tx);
17:     end if
18:     dequeue(tx_queue);
19:     execute(tx);
20: end while
```

Backup TM Roll-back Required

Previously we stated that it was not possible to utilise the roll-back procedure utilised by RR and RC isolation when the WSC was enabled. This is true when a TM is classified as a *primary*, however in the case of a *backup* TM it is possible, and sometimes required, for a roll-back mechanism to be utilised. The roll-back mechanism detailed on lines 5-15 of Algorithm 6 functions in much the same way as when RR or RC isolation is utilised and is required in the following scenario:

Algorithm 7 TM *process*(tx) with WSC

```
1:  if TM hosts primary replica of any (k,v) in tx then
2:      if tx.exceptional is true then
3:          v ← abort(tx);
4:      else
5:          v ← write_skew_check();
6:      end if
7:      send_vote(v);
8:  end if
9:  if TM is tx Coordinator then
10:     votes_rec ← size(tx.votes[]);
11:     while votes_rec < #primaries in tx.dst do
12:         if tx.duration > timeout then
13:             decision ← abort(tx);
14:             break;
15:         else
16:             v ← receive_vote();
17:             tx.votes[source_address(v)] ← v;
18:             if v = abort(tx) then
19:                 decision ← abort(tx);
20:             end if
21:         end if
22:     end while
23:     if descision is not set then
24:         decision ← commit(tx);
25:     end if
26:     send decision to all d ∈ tx.dst;
27: end if
28: tx.decision ← receive_final_decision();
29: return tx.decision
```

Let m_i and m_j, contain the transactions tx_i and tx_j respectively, where $m_i \prec m_j$. Assume that both tx_i and tx_j consist of a single write operation, $update(k1,x)$ [4], where nodes N_c1 and N_c2 host the *primary* and *backup* replica of $k1$, respectively. Finally, assume that node N_c3 is the $Tx.c$ for tx_i and N_c4 is the $Tx.c$ for tx_j.

Let m_i and m_j be successfully delivered at N_c1, resulting in the *primary* TM (N_c1) processing Tx_i and Tx_j in the correct order. Let, N_c3 and N_c4 receive m_i and m_j respectively, without any order violations. Now consider that m_i suffers an order violation at its backup node, N_c2, and m_j is delivered ahead of it in the total order. This results in N_c2's active transaction being tx_j.

Consequently, tx_i is the active transaction for its coordinator and all of its *primary* TMs, therefore it is guaranteed that a final decision for tx_i will be reached and eventually sent to N_c2's TM. As N_c2's active Tx is tx_j, N_c2 will not process tx_i's final decision until tx_j has finished executing. Therefore, N_c2's TM will execute tx_j before tx_i, whereas k's *primary* TM N_c1 will have executed tx_i before tx_j. Thus, the *backup* replica of k has become inconsistent from the *primary* replica. **Note:** An inconsistency only occurs at the *backup* TM if the final decision for both tx_i and tx_j is commit.

Figure 5.5 shows the message flow required by all parties in order for the above scenario to occur.

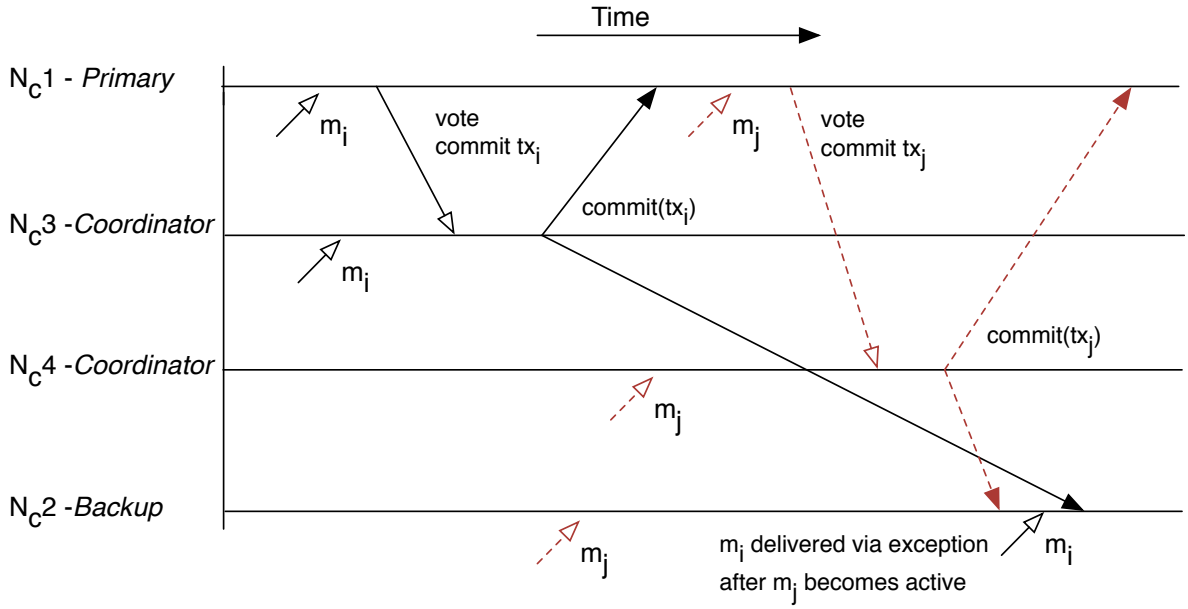

Fig. 5.5: PSCast: Roll-back Scenario with WSC

[4] We assume that the value of x is different for each Tx

When an inconsistency occurs between a *backup* and *primary* replica as described above, it is possible to perform a roll-back operation at the *backup* node as the *backup* node does not influence the coordinator's decision to commit/abort a transaction. Hence, performing a roll-back operation on a *backup* node is the same as when RR or RC isolation is utilised.

Deadlock Scenario

Let N_c1 and N_c2 be the *primary* replica for $k1$ and $k2$, respectively. Assume that *amcast*s m_i and m_j correspond to transactions tx_i and tx_j, with both transactions having N_c3 as their coordinator and the following operations included in their payload:

```
Tx.begin();
   update(k1, x);
   update(k2, y);
Tx.commit();
```

Assume that m_i and m_j are both successfully delivered at N_c1 and N_c3, resulting in both of their TM's setting tx_i as their active transaction. Now assume that m_j is delivered ahead of m_i at N_c2 due to an ordering violation; resulting in N_c2's TM having tx_j as its active transaction.

Such a scenario results in a deadlock occurring between the two transactions, as the *coordinator*, who is executing tx_i will be waiting indefinitely for a vote from N_c2. This vote will never arrive from N_c2 as its current transaction is tx_j, hence only a vote for tx_j will be sent to $tx_j.c$. Figure 5.6 shows the message flow required for deadlock to occur.

To prevent such a deadlock from halting progress indefinitely we introduce an abort timeout as per the Two-phase commit protocol (§ 2.5.3.3). Therefore, in the case described above, tx_i will eventually abort as no progress can be made until tx_i timesout at N_c3.

Note: Although we have reintroduced deadlock into the total order commit protocol, such occurrences are very rare. The probability of a transaction being aborted is equal to the probability of an ordering violation occurring at a *primary* TM when a Tx involves multiple keys whose *primary* replicas exist across several nodes.

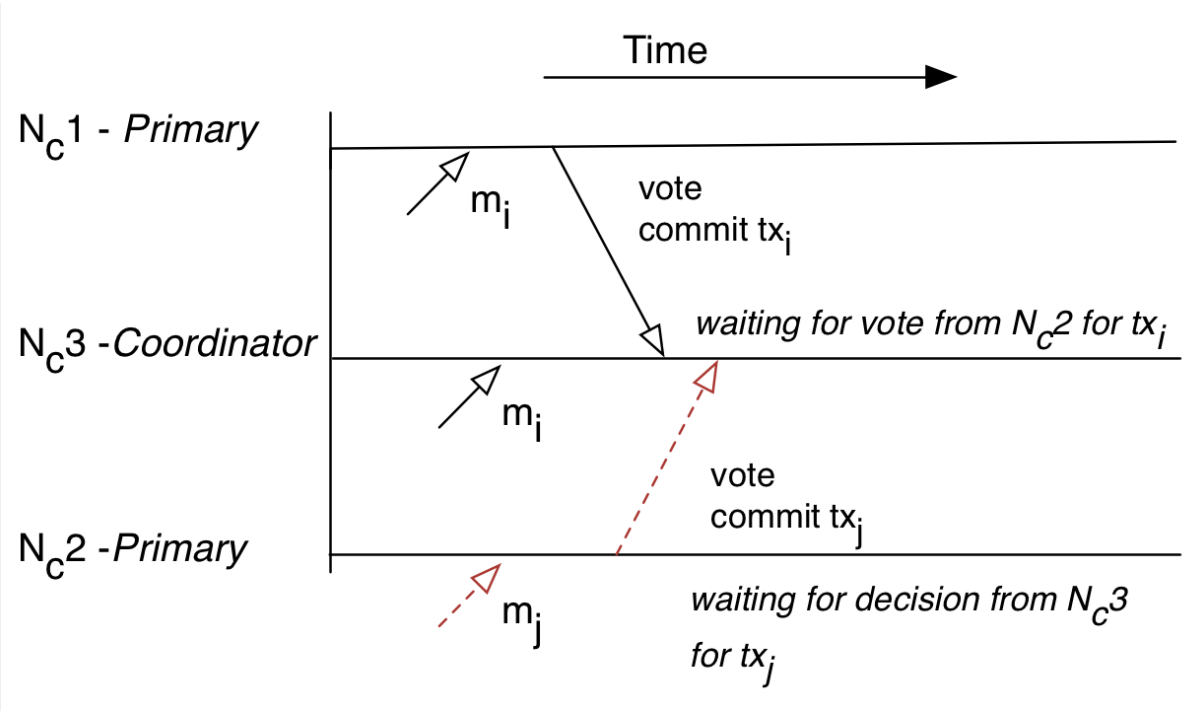

Fig. 5.6: PSCast: Deadlock Scenario with WSC

5.6 Summary

In this chapter we have explored the consequences of utilising the ABcast protocol for state machine replication between *s*-nodes in the AmaaS model; which we refer to as PSCast. We explore the consequences of ABcast's probabilistic guarantees not being met between *s*-nodes, specifically ordering violations at client nodes, and potential strategies for overcoming such violations. More specifically, we have provided an in-depth exploration of potential strategies that could be adopted by Infinispan's transaction manager to ensure that the consistency of its key/value pairs are maintained.

Chapter 6

Performance Evaluation

This chapter provides a comprehensive performance evaluation of the key concepts introduced in this thesis. Evaluation focusses on five issues:

i The scalability of the AmaaS model over P2P.

ii A performance comparison between ABcast and TOA when utilised in the context of AmaaS.

iii The performance of ABcast when requests are frequent over a long period of time.

iv The probability *R* of ordering correctness by ABcast, more specifically Aramis.

v The effectiveness of ABcast's non-blocking message delivery in the interim period between a node crashing and the GM protocol publishing a new view.

The remainder of this chapter is structured as follows: First we detail an experiment that emulates Infinispan's distributed transactions, and is used to evaluate (i) and (ii) (§ 6.1). This is followed by an experiment that replicates the inner workings of the SCast protocol, in order to simulate an AmaaS service operating at maximum capacity, which is used to evaluate (iii) and (iv) (§ 6.2). Finally, we introduce an experiment that evaluates (iv) and (v) by crashing a node while *abcast*s are sent between nodes (§ 6.3).

6.1 AmaaS

To test our hypothesis that the AmaaS model can improve the scalability of Infinispan's distributed transactions, we developed an experiment that emulates the workflow of these transactions by replicating the *amcast* messages sent by Infinispan when executing total order transactions (§ 2.5.3.4). This experiment does not utilise Infinispan, or implement a basic transaction manager, rather it focuses purely on replicating the underlying communication stages required by Infinispan transactions.

Existing research [64] has already shown the benefits of utilising a total order protocol instead of 2PC, therefore our experiments concentrate on the performance of the underlying *amcast* protocol used to coordinate these transactions.

In our experiments, if a new *amcast* protocol can demonstrably increase throughput and reduce latency of *amcast* messages as the number of destinations increase, then we can infer that the scalability of the Infinispan system will be improved by adopting this protocol. Therefore, if our experiments show that the AmaaS model consistently outperforms P2P, then we assume our hypothesis to be true.

In order to compare and contrast the performance of the AmaaS and P2P approach, it was necessary for two experiments to be created.

The first experiment was designed to evaluate the latency and throughput of both a SCast and PSCast service. This experiment allows the performance of the AmaaS model to be evaluated, whilst also enabling the performance of the underlying *abcast* protocols, which are utilised by the services for state machine replication, to be contrasted. Both the SCast and PSCast services utilise a simplified version of the SCast Protocol (§ 3.4) to coordinate interactions between *c*-nodes and *s*-nodes.

The second experiment was designed to measure the performance of *amcast* requests when utilising the P2P approach. This experiment utilises the same workloads and parameters as the first experiment, however, as per the P2P model, no *s*-nodes are present and consequently there is no need for the SCast protocol. Instead, the TOA protocol is executed directly between *c*-nodes when emulating transactions.

Utilising the same experiment structure and workloads across both sets of experiments allows us to compare the performance of the two system models across a consistent environment.

This consistency enables us to contrast the performance of the TOA protocol, when utilised in both the P2P model and the SCast service, with the ABcast protocol utilised by the PSCast service.

6.1.1 Experimentation

SCast and PSCast Services

We implemented our AmaaS services using the JGroups[38] framework with $n = 2$ and $n = 3$ *s*-nodes. All nodes in the experiment utilised commodity PCs of *3.4GHz Intel Core i7-3770* CPU and 8GB of RAM, running *Fedora 20* and communicating over Gigabit Ethernet. The *s*-nodes and *c*-nodes utilised in our experiments are a part of a large university cluster, hence communication delays between nodes can be quite volatile as they are influenced by other network traffic and processes launched by other users.

Our experiments are based upon a heavily modified version of an existing performance test available in the JGroups[38] framework, which mimics the partial replication of key/values in Infinispan[35]. In these experiments we utilise ten *c*-nodes in the same cluster, each of which emulates a transaction system which is reliant on an AmaaS service for transaction ordering. Each *c*-node operates 25 concurrent threads to initiate and coordinate transactions, and a transaction Tx involves a set $|Tx.dst| = 3,4,\ldots,10$ *c*-nodes; where $|Tx.dst|$ includes $Tx.c$. A thread coordinating a transaction starts its next transaction, Tx', as soon as it executes a commit/abort decision for the currently active Tx. Thus, at any moment, 250 transactions are in different stages of execution.

All of the emulated transactions consist purely of key/value write operations and thus require *amcast* messages for coordination. Infinispan's read requests ($get(k)$) are not emulated, as the retrieval of key/values occurs before $Tx.c$ *amcast*s its $prepare(k)$ message, hence read operations have no baring on the performance of the underlying *amcast* protocol.

Both the SCast and PSCast services utilise a modified version of the SCast protocol defined in section 3.4 to dictate the interactions between *c*-nodes and *s*-nodes. In our implementation *s*-nodes utilise message bundling to reduce the total number of *abcast* messages required.

Omission: Stage 1 of the SCast protocol has been omitted from this implementation because we only compare the performance of the two approaches in a crash-free scenario. Our rationale

for removing this stage, was that the fault-tolerance provisions described in SCast is only one possible solution for ensuring that the *amcast* protocol can continue to execute in the event of the original coordinator crashing during a multicast and alternative solutions are not obliged to utilise this additional communication stage. Furthermore, in our experiments we are comparing SCast to the TOA protocol which does not implement any mechanism to cope with a crashed message originator, therefore removing Stage 1 of SCast protocol makes for a fairer comparison of the two protocols.

Experiment Workflow: The workflow of a transaction in our experiments is as follows:

1. A coordinator thread submits its *amcast* request for Tx, denoted as $req(Tx)$, with some *s*-node; who stores the request in FIFO order within its ARP.

2. The *s*-nodes *Send* thread retrieves requests stored in its ARP and places them into a message bundle *mb*, which can have a maximum payload of $1kB$ [1], then *abcast*s *mb* to all other *s*-nodes.

   Note: If there exists no requests in the ARP, then the *Send* thread waits for it to become non-empty before initiating the next mb'. Hence, the number of requests bundled in any *mb* varies depending on the arrival rate of requests.

3. Once $req(Tx)$ has been *abcast* to all *s*-nodes, a response message, $Rsp(Tx)$ it is sent to $Tx.c$ who disseminates this message to $Tx.dst$ as $mcast(Tx)$.

4. When all $d \in Tx.dst$ have received and delivered $mcast(Tx)$, as per the delivery conditions of the SCast protocol, the transaction is considered complete and the coordinator thread can start executing Tx'.

In our experiments that utilise ABcast (PSCast service), an additional phase is required before the experiments can begin. Prior to accepting requests from *c* nodes, *s*-nodes must participate in an initialisation period that lasts approximately 1-2 seconds. During this period, the clocks of the *s*-nodes are synchronized and each *s*-node broadcasts 10^3 *probe* messages, with a payload of $1kB$, to all other *s*-nodes. The purpose of these probe messages is to record the NT_P latencies required by ABcast's DMC.

[1] In the experiments that utilise the ABcast protocol, we *pad* the contents of the message bundle to ensure that it is always equal to $1kB$. This ensures that all messages *abcast* by the protocol are approximately the same size, which increases the accuracy of the DMC's predictions at the expense of redundant bandwidth.

Finally, all of our experiments with ABcast (PSCast Service) utilise the the following constant values. The DMC utilises $R = 0.9999$ (§ 4.3.5), and AFC utilises δ_{min} and δ_{max} values equal to $1ms$ and $10ms$, respectively (§ 4.5.1).

P2P

In order to test the performance of P2P total order transactions we repeated the experiments detailed above, however, as per the P2P model, all *c*-nodes coordinate transactions between themselves without utilising any *s*-nodes. In these experiments, a transaction is considered complete when it has been successfully *amcast* to all $d \in Tx.dst$ by the P2P protocol; where success is defined as all correct destinations delivering the *amcast* message.

Note: The same cluster of machines were used for both the P2P and AmaaS experiments to ensure a fair comparison between protocols.

6.1.2 Results

Our performance evaluation focuses on the comparison of the TOA protocol, being utilised in a traditional P2P scenario (*TOA-P2P*), with two different AmaaS services that utilise the SCast protocol. The SCast service utilises the deterministic protocol TOA for state machine replication, whilst the *PSCast* service utilises the probabilistic protocol ABcast, hence we refer to these two services as the *TOA-Service* and *ABService*.

The performance of all three approaches is measured based upon the average transaction latency and throughput rate. In both the *TOA-Service* and *ABcast-Service*, latency is measured as the time elapsed between a *c*-node's initial transmission of $req(Tx)$ to some *s*-node, and **all** members of $Tx.dst$ delivering $mcast(Tx)$ to the experiment application. In TOA-P2P, latency is measured as the time taken for all $Tx.dst$ to deliver Tx to the experiment application. For both approaches, throughput is measured as the average number of *abcast*s delivered by the experiment application per second at each *c*-node.

Note: All of our experiments were conducted in isolation in order to prevent any side effects caused by simultaneously executing multiple experiments on the same cluster, however we conducted all experiments over approximately the same time period to ensure that the network was under similar loads for all of our experiments.

Figures 6.1 and 6.2 show the latency and throughput results for our experiments, with 2*N* and 3*N* representing an AmaaS service that consists of two and three, *s*-nodes respectively. Each plot on the graph is an average of three *crash-free* trials; where a trial consists of each *c*-node completing 10^4 transactions for a specific value of $|Tx.dst|$. Thus, in all three trials the *TOA-Service* and *ABService* each receive a total of 10^5 *amcast* requests. In TOA-P2P, each *c*-node initiates 10^4 TOA executions between its peers (10 *c*-nodes $\times 10^4 = 10^5$).

Concerning AmaaS performance, Table 6.1 shows the average number of client requests received, the average number of *abcast* messages sent and the average number of requests bundled into each *abcast*, based upon all of our experiments that utilised a AmaaS service. All of theses average values are calculated based upon the statistics recorded by each *s*-node that was utilised during our experiments.

Experiment	# Client Requests	# *abcast*s	Bundle Size
ABService-2N	50000	12763.4	4
TOA-Service-2N	50000	16632	3
ABService-3N	33333.3	13416.4	2.5
TOA-Service-3N	33333.3	13507.8	2.5

Table 6.1: Average Node Statistics for Emulated Transaction Experiments

Table 6.2 shows the performance of the ABcast protocol in both the ABService-2N and 3N experiments. It shows the average number of *abcast*s sent per node and the average number of these messages that were delivered by the Aramis protocol, as well as providing the total percentage of *abcast*s that were delivered via Aramis. Furthermore, this table details the ratio of *s*-nodes that delivered an *abcast* via Aramis compared to the total number of *s*-nodes utilised. For example, in the case of ABService-2N we performed 24 experiments, therefore we have statistics for 48 *s*-nodes and our records show that only 3 of these nodes utilised Aramis to deliver one or more *abcast* messages.

The very small number of Aramis deliveries is understandable as Δ_m of Aramis is estimated pessimistically and no crashes occur. In fact, it is surprising that some *abcast*s were indeed delivered by Aramis faster than Base and more discussions on this aspect are presented in subsection 6.1.3.

Experiment	# *abcasts*	Nodes Affected	Avg Aramis Deliveries	% Aramis Deliveries
ABService-2N	12763.4	3:48	10.8	0.085%
ABService-3N	13416.4	30:72	15.3	0.114%

Table 6.2: Average ABcast Statistics per Node

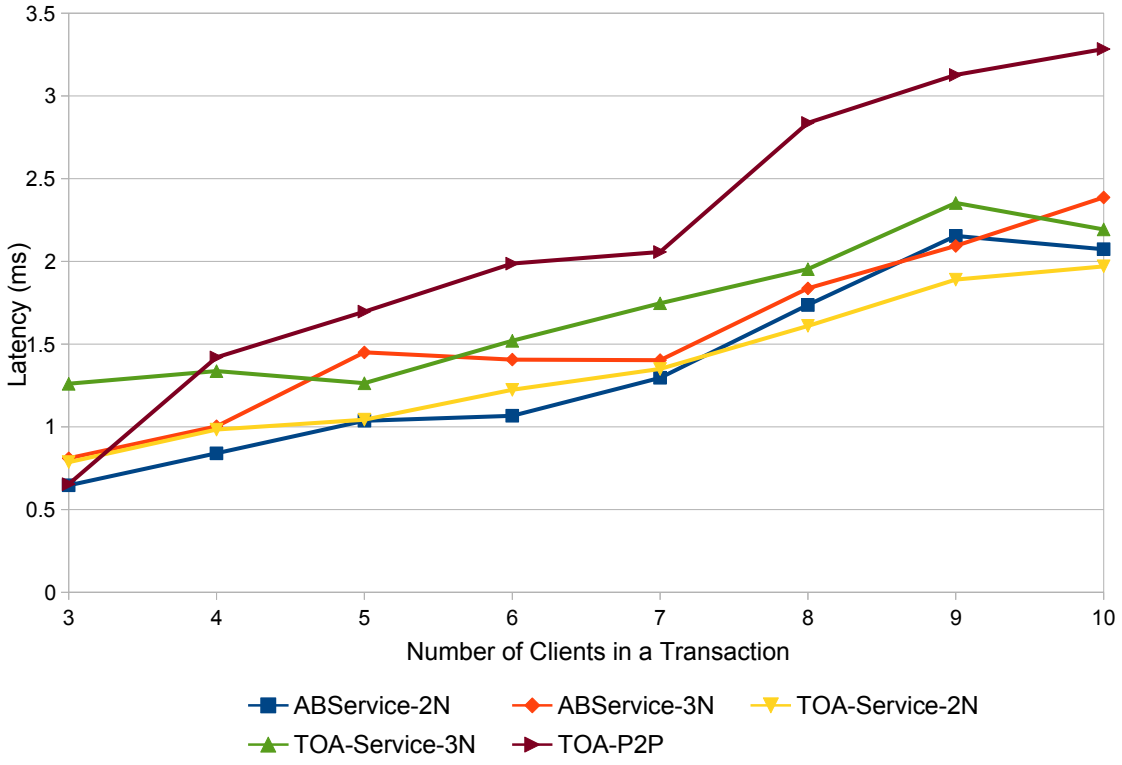

Fig. 6.1: AmaaS Latency Comparison

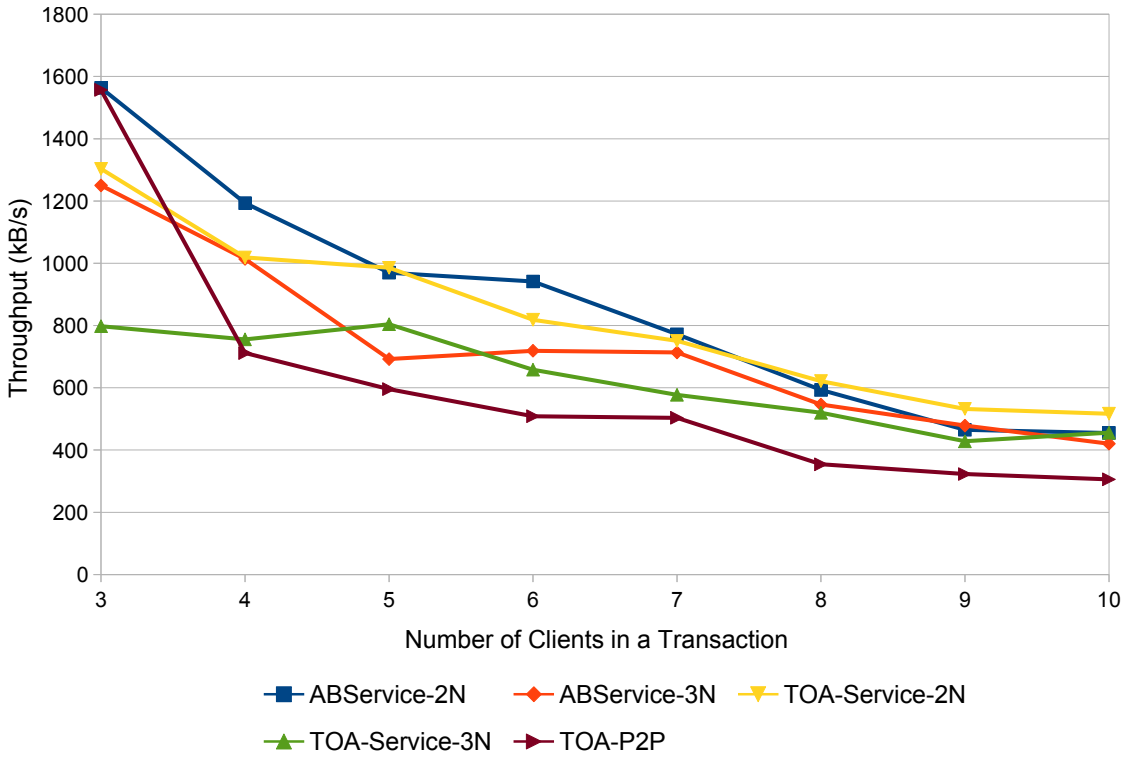

Fig. 6.2: AmaaS Throughput Comparison

6.1.3 Evaluation

This section is split into three distinct subsections. First we directly compare the performance of the AmaaS service and the P2P approach with both experiments utilising the same TOA protocol. We then evaluate the performance of the ABService in contrast with the previous two approaches, focusing on the differences between the performance of the ABcast and TOA based service. Finally, we evaluate the performance of ABcast, focusing on how often the Aramis protocol was utilised to deliver messages and its ability to maintain ordering correctness.

AmaaS vs P2P

In Figure 6.1 we can see that when $|Tx.dst| \geq 4$, TOA-P2P's *abcast* latencies increase considerably when compared to the two TOA-Service experiments. With TOA-P2P experiencing approximately a 25% and 50% increase in average latency when compared to TOA-Service-3N and TOA-Service-2N respectively. Thus, indicating that *amcast*ing is best provided as a service as the number of clients involved in a transaction increases. Comparing throughput in Figure 6.2 leads to similar conclusions, with the steady throughput observed as $|Tx.dst| \rightarrow 10$ also suggesting an absence of node saturation.

TOA-P2P's superior performance when $Tx.dst < 4$ can be attributed to the additional stages involved when utilising the *AmaaS* model. For example when TOA-Service utilises two *s*-nodes (2*N*) the following stages are required: $Tx.c$ sends a request, the *multicast service abcast*s it with $|m.dst| = 2$ to all *s*-nodes and returns it to $Tx.c$, who must then multicast $mcast(Tx)$ to $Tx.dst$. Ignoring the individual message cost of each stage the total number of stages is four, whereas in TOA-P2P the only step required is the *amcast*ing of Tx. So although $|m.dst|$ for each *amcast* is less in TOA-Service ($|m.dst| = 2$) than TOA-P2P ($|m.dst| = 3$), the overhead of sending a request to the *multicast service* and back is much greater than the savings offered by reducing $|m.dst|$ by one node. However, as $|Tx.dst|$ increases, the overhead of TOA-P2P's increased $|m.dst|$ becomes significant, to the point where TOA-Service's additional communication stages becomes less of an overhead than the cost of TOA-P2P *amcat*ing to a large $m.dst$.

ABService vs TOA-Service

In Figure 6.1 we can see that the latencies encountered by the ABService-2N and TOA-Service-2N experiments are very similar, regardless of the number of clients involved in a transaction, with the maximum difference between any two plots being no greater than 0.3 milliseconds. Interestingly, our experiments show that in the majority of experiments, the ABService outperforms TOA-Service. This superior performance can be attributed to a combination of two factors: the number of requests that are bundled on average per *abcast* and the overall message cost associated with the underlying *abcast* protocol.

The average number of client requests bundled into a single *abcast* can play a decisive role in the latency and throughput of a AmaaS service as the higher the average bundle rate, the lower the total number of *abcast*s required. As the *abcast*ing of requests between *s*-nodes is the most expensive operation, in terms of bandwidth and latency in the AmaaS model, it is self-evident that reducing their frequency will reduce the average latency encountered by client requests, therefore reducing the total duration of a transaction.

Table 6.1 shows that the average bundle rate for the ABService-2N was 4 messages, whilst it was only 3 for TOA-Service-2N. Therefore, on average a node in TOA-Service-2N sends ≈ 3869 more *abcast*s then its counterpart ABService-2N, which partially explains the difference in performance between the two approaches.

The difference in overall message cost between the two *abcast* protocols is a consequence of the two different approaches to solving *abcast* and the optimisations present in the ABcast protocol (§ 2.2.2 & 4.4.1). The ABcast protocol piggybacks any outstanding message acknowledgements on subsequent message broadcasts, enabling *abcast*s to be executed in a single phase when all nodes are frequently sending *abcast*s. Whereas, the JGroups implementation of the TOA protocol does not implement any optimisations, and thus, each broadcast always consists of two phases, therefore increasing the average latency encountered by transaction requests.

Correspondingly, it is possible to observe that the average and maximum difference between the latencies encountered in the ABService-3N and TOA-Service-3N experiments is greater than that observed when $N = 2$. The improving performance of the ABService can be attributed to the ABcast optimisations becoming more effective as the number of *s*-nodes increase. For example, if $N = 3$ and ABcast sends a broadcast, the total message cost for that single *abcast*

is only 2 unicasts when piggybacking occurs, whereas with TOA the total cost is always 6 unicasts. Clearly, such an optimisation will have a positive effect on the performance of the ABService implementation, especially when service requests are evenly distributed amongst *s*-nodes and are arriving frequently which is ideal for the ABcast optimisations.

Interestingly, in Table 6.1 we can see that the average bundle rate of ABService-3N and TOA-Service-3N are almost the same, yet the difference between the observed latencies in the two approaches has increased. This suggests that in these experiments the average bundle rate has no significant impact on the performance of the two approaches.

The large difference between the average bundle rate observed in ABService-2N and 3N, is a direct consequence of the DMC's calculations and how AFC (§ 4.5.1) manages broadcast rates. Recall that the delay imposed by AFC, for an *abcast* message, increases when latencies start to exceed the previously calculated x_{mx} value, and decreases to δ_{min} when no such latencies are observed. When 2 *s*-nodes are utilised, the observed x_{mx} is typically lower than 3 *s*-nodes, as the number of unicasts sent between *s*-nodes is less; hence the probability of large delays being observed is reduced. The smaller the average x_{mx} value, the more susceptible the system is to delays periodically exceeding x_{mx}. Therefore, when 2 *s*-nodes are utilised the probability of the calculated AFC delay regularly exceeding δ_{min} increases, which in turn reduces the node's broadcast rate. Consequently, the number of requests which can accumulate between *abcast*s will increase, and hence the average bundle rate also increases. When 3 *s*-nodes are utilised, the DMC's observations are typically more stable, resulting in less *outlier* latencies being recorded and the broadcast rate being more stable; hence an average bundle rate that is approximately the same as the TOA-Service-3N.

The throughput of the ABService and TOA-Service for both $2N$ and $3N$ follows a very similar pattern to that observed when analysing their latencies. This is not surprising as the average transaction latency has a direct impact on the average rate of throughput. Combining the results shown in Figures 6.1 and 6.2, it is clear to see that the ABService provides comparable performance to that of the TOA-Service and that both of these AmaaS solutions consistently outperform TOA-P2P when $Tx.dst > 3$.

Note: While the overall performance of ABServive and TOA-Service are similar, the ABcast protocol used by ABService provides non-blocking message delivery in the event of node-

failures, as well as stronger guarantees on message ordering than TOA. Recall that TOA does not provide *uniform agreement* in the event of a message originator crashing (§ 2.5.3.5), therefore it is not unreasonable to imagine that the performance gap between the two protocols would increase, in favour of ABService, if the TOA protocol was adapted to provide *uniform agreement*.

ABcast

In Table 6.2 we can see that only 3 of the 48 nodes utilised by ABService-2N delivered an *abcast* via the Aramis protocol, with the average number of messages being ≈ 11, only 0.085% of all messages. Hence, the Δ_m value calculated by the DMC was sufficient for 99.915% of *abcast*s. The results of the ABService-3N experiments shows that as the number of *s*-nodes increased, the total number of Aramis deliveries also increased. Almost 50% of nodes delivered at least one message via Aramis, with an overall average of ≈ 16 messages per node. Although this is a large increase in the number of nodes requiring Aramis, the protocol still only accounts for 0.114% of all *abcast*s sent.

The increase in Aramis deliveries as the number of *s*-nodes increase can be attributed to the DMC recording each latency anomalously (without regard for source of the message) and calculating Δ_m based upon these latencies. In the experiments where $n = 2$, we know that all of the latencies recorded by node Ns_1 will be from messages originating at Ns_2. Therefore, when node Ns_1 broadcasts message m, it is guaranteed that the calculated Δ_m has been calculated utilising latencies representative of Ns_2's past performance. Whereas, when $n = 3$, Ns_1 will have calculated Δ_m based upon latencies recorded from both Ns_2 and Ns_3, therefore it is possible that if Ns_3 is slower than Ns_2, the latencies calculated from Ns_2 will dilute the larger latencies recorded from messages originating at Ns_3. Thus, the calculated Δ_m could be smaller than the value required by the slower node Ns_3.

Note: None of the experiments that delivered a message via Aramis suffered an ABcast ordering violation and hence no SCast ordering violations occurred at any client nodes. Furthermore, we repeated our experiments with delivery condition $D1_B$ of the Base protocol disabled, which causes all *abcast*s to be delivered via Aramis, in order to evaluate the accuracy of Δ_m. We found that, for both $n = 2$ and $n = 3$, the calculated Δ_m was sufficient for all *s*-nodes to deliver

messages without a single ordering violation occurring. As expected, latencies were large, and they were so large that a single experiment (emulating 10^5 transactions) took several minutes to complete. Obviously such large latencies are not practical, however these experiments provide evidence of Δ_m's ability to prevent ordering violations.

6.1.4 Summary

When deploying a large-scale distributed transaction system that executes transactions which span several nodes ($|Tx.dst| > 3$), higher throughout and lower-latency can be achieved by utilising the AmaaS model for *amcast* messages. Furthermore, such a service can provide non-blocking *amcast*s when the ABcast protocol is utilised for state machine replication, whilst maintaining similar levels of performance to when a GM based protocol is utilised.

6.2 ABcast - Infinite Clients for Extreme Load Conditions

In the previous section, we tested the performance of the AmaaS approach whilst utilising the ABcast protocol. Our results showed, that the Aramis protocol was rarely required to deliver messages, accounting for only 0.015% and 0.114% of messages, when the number of *s*-nodes was two and three respectively. However, in these experiments the total number of *abcast* messages was, on average, relatively low for each node; typically less than 2×10^4. Furthermore, each *s*-node's rate of *abcast*s would vary depending on the restrictions of the AFC protocol and the rate at which requests were being received by *c*-nodes.

Due to the number of client nodes being relatively small, it is probable that at times an *s*-node's ARP could have been empty. Therefore, in order to test the performance of ABcast under extremely heavy loads, it was necessary for a new experiment to be developed. The purpose of these experiments are two fold. First, they allow us to measure how often Aramis is required to deliver messages and the frequency of order violations. Secondly, they allow us to monitor the values calculated by the DMC during high levels of network load and determine their effect on the resulting Δ_m.

In order to test the performance of ABcast under heavy loads, we could simply increase the number of client nodes that were used in our previous experiment, however this would require

a large amount of resources and would be cumbersome to orchestrate. Furthermore, such an approach does not guarantee that the ARP of a given *s*-node will always have a request to process.

We propose a new experiment, which we refer to as an *infinite client system* as it represents the performance of AmaaS ordering service if each *s*-node always has a full ARP. This experiment does not utilise client nodes at all, instead, it simply consists of n nodes initiating *abcast*s *as fast as* AFC permits. This is the same as the steps required by SCast for state machine replication, however we do not have the overhead of maintaining the data structures required by SCast at each node; *i.e.* $order\_history[]$. Therefore the delay between subsequent *abcast*s will be less in this experiment, hence the ABcast protocol will be under a heavier load in these experiments than is possible in a *complete* SCast implementation.

6.2.1 Experimentation

The infinite client experiment was implemented using the JGroups framework and the same ABcast implementations as the experiments detailed in § 6.1.1. Furthermore, our experiments utilised the same computer cluster and specification of machine as our previous experiments.

An individual *infinite client* experiment consists of 3 nodes sending 10^6 *abcast*s between themselves; with each individual node sending $\frac{10^6}{n}$ messages with a payload of $1kB$. The workflow of our experiments is as follows:

1. A node broadcasts its requests as fast as possible using a single thread, which represents the *sender* thread utilised in SCast

2. As soon as a message has been sent, another ABcast is initiated; where the sending of a message m consists of m being sent down the JGroups stack, processed and delayed by AFC, before being unicast to all n nodes.

3. An experiment is considered complete when each node has delivered 10^6 messages, or if one or more order violations ($\#violations$) has occurred, then $(10^6 - \#violations)$ [2].

For all of our experiments the ABcast protocol used the following constant values: $R =$

[2] As none of our experiments maintain a state at the application level, *abcast*s that cause order violations are not *delivered* to the application, instead their occurrence is simply recorded.

0.9999, $\delta_{min} = 1ms$ and $\delta_{max} = 10ms$. Furthermore, we utilise the same initialisation period from the AmaaS experiments.

6.2.2 Results

The experiment detailed in § 6.2.1 were executed a total of ten times, utilising the same machines for each experiment. Table 6.3 presents the results of each of these experiments based upon each node's individual performance as well as the performance of the cluster as a whole; where Ns_1, Ns_2, Ns_3 correspond to the values recorded by an individual node and we define the cluster as being the combined performance of $\{Ns_1, Ns_2, Ns_3\}$. For each node in an experiment, we show the total number of *abcast*s that were delivered by Aramis and in brackets the number of order violations. We also show the total number of *abcast*s delivered by Aramis across the cluster, and the percentage of all *abcast*s that are delivered by Aramis.

Table 6.4 presents the average delivery latency encountered by all *abcast*s sent via ABcast (including those delivered by Aramis), as well as the average Δ_m value calculated by each node. Each node records its delivery delay as $Dt_m - m.ts$, where $m.ts$ is the timestamp allocated to an *abcast* message m when an *abcast* is initiated and Dt_m is the time at which m is passed upto the application. The average Δ_m value is recorded using a given node's calculations of Δ not those recorded by others, hence Ns_1's average is calculated using only Δ values calculated by Ns_1's DMC. Therefore, the 'overall' entry in the table provides the average Δ value of all nodes in the cluster.

6.2.3 Evaluation

In Table 6.3 we can see that out of all 10^7 messages, only 1.36% of *abcast*s were delivered by Aramis. Furthermore, out of these 135684 Aramis deliveries there was not a single order violation, therefore *ABcast*'s guarantees were maintained even when the rate of requests was very high. This lack of order violations implies that the calculated Δ_m is sufficiently large to prevent messages being missed, whilst still being small enough for some *abcast*s (1.36%) to be delivered via Aramis before the Base protocol could complete. Thus, for 1.36% of *abcast*s the ABcast protocol reduces latency and prevents message blocking even in the absence of node failures, when compared to traditional GM based protocols. Finally, the lack of order

Experiment	Ns_1	Ns_2	Ns_3	Total	% of all *abcast*s
1	9220, (0)	7929, (0)	6434, (0)	23538	2.36
2	3348, (0)	4555, (0)	5008, (0)	12911	1.29
3	4496, (0)	4920, (0)	1952, (0)	11368	1.14
4	5832, (0)	6439, (0)	4801, (0)	17072	1.71
5	5320, (0)	5757, (0)	4066, (0)	15143	1.51
6	4181, (0)	3286, (0)	4157, (0)	11624	1.16
7	1743, (0)	2237, (0)	2235, (0)	6215	0.62
8	4188, (0)	1846, (0)	5421, (0)	11455	1.15
9	5621, (0)	4242, (0)	5291, (0)	15154	1.52
10	2953, (0)	5014, (0)	3192 , (0)	11159	1.12
Total	46902, (0)	46225, (0)	42557, (0)	135684	1.36

Table 6.3: Aramis deliveries (Order Violations) for infinite clients - $\rho_{min} = 1$

Node	Avg Delivery Latency (ms)	Avg Δ_m (ms)
Ns_1	21.48	710.34
Ns_2	23.47	687.29
Ns_2	25.45	767.74
Overall	23.47	721.79

Table 6.4: Average ABcast Latencies and Calculated Δ_m - $\rho_{min} = 1$

violations indicates that the protocol is able to handle a large number of *abcast* requests without compromising on message ordering.

Correspondingly, Table 6.4 shows that the average delivery latency encountered by *abcast* messages remains low even when the network is heavily loaded. Furthermore, it shows that the average Δ_m value remains below 800*ms* for each node.

ABcasts ability to provide to low-latency message delivery in such conditions is crucial, as the speed of the *abcast* protocol utilised in an AmaaS service ultimately determines the response time for each client request. More significantly, the low average Δ_m value shows that, even under the heaviest of loads, the DMC is able to calculate an average Δ_m that is sub 1 second and still deliver all messages without a single order violation. This is a ***vital*** result, because if the Δ_m value became increasingly large as the load increased, Δ_m would start to exceed the typical delay required by the GM service to publish a new view after a node crash, therefore rendering our hybrid approach redundant.

6.2.4 Summary

The ABcast protocol is capable of providing low-latency *abcast*s over a sustained period of time in conditions representative of those found in an AmaaS service. In such conditions, the DMC consistently calculates a Δ_m value that is small enough to outperform GM services, whilst being sufficiently large to ensure that no violations of *abcast* guarantees occur when messages are delivered by Aramis.

6.3 ABcast - Fault Tolerance

In our previous experiments with ABcast we have evaluated the performance of the protocol in the context of an AmaaS service where no node failures occur. However, as ABcast has been designed to compliment the low-latency performance of GM protocols, by allowing for non-blocking message delivery when node crashes occur, it is necessary to ensure that Δ_m is sufficiently small for messages to be delivered in the interim period between a node crash and the GM service publishing a new view.

Ultimately, if the GM service is able to publish a new view before any messages are de-

livered via the Aramis protocol, then the hybrid approach we have taken is unnecessary. In such a case, a traditional GM based protocol would be more suitable as order violations are not possible. Therefore in order to determine the effectiveness of ABcast's hybrid approach, it was necessary to create an experiment that monitors the number of messages, if any, that are delivered by ABcast in the interim period between a node crashing and the GM service publishing a new view.

Such an experiment also enable us to explore the impact of utilising different values for ABcast's configuration parameters, such as ρ_{min} and R, on the number of messages delivered in this interim period. More specifically, these experiments allow us to explore the impact of these configurations parameters on the average Δ_m value calculated by a node and how these variations impact the observed number of order violations.

6.3.1 Experimentation

In order to test the performance of ABcast when a node crashes, we reuse the experiments detailed in § 6.2.1. However, in these experiments, instead of all 3 nodes sending a total of 10^6 *abcast*s, only 2 of the nodes complete their broadcasts. The third node, Ns_3, is crashed after sending 50000 *abcast*s.

Note: As JGroups is implemented in the Java programming language, we crash Ns_3, by crashing the underlying Java Virtual Machine (JVM), not the physical machine.

In order to understand this experiment and why crashing the underlying JVM is necessary, it is important to recall the design of JGroup's GMS and associated *failure detection* protocols presented in section 2.6. Recall that the *failure detection* protocol `FD_SOCK` is particularly effective at detecting node crashes; with crashes typically detected within seconds. Therefore, in order for ABcast to deliver messages before the GMS protocol becomes aware of a node crash, the calculated Δ_m would need to remain relatively small ($\lesssim 2$ *seconds*) throughout our experiments.

Due to `FD_SOCK`'s use of Java shutdown hooks, it was not possible for the crashed node in our experiments to be exited as a normal Java application; as this would result in the terminating node sending a leaving message to all members in the view and alerting GMS almost instantly that the node was leaving the current view. Clearly such a leaving message cannot be sent when

a node is crashed unintentionally. Therefore, it was necessary for us to terminate the JVM in the most disruptive manner possible, in order to replicate the untimely occurrence of a real node crash. We achieved this by using reflection to access the *sun.misc.Unsafe* api and crash the JVM. The code used to crash the JVM is shown below:

```
Field theUnsafe = Unsafe.class.getDeclaredField("theUnsafe");
theUnsafe.setAccessible(true);
((Unsafe) theUnsafe.get(null)).getByte(0);
```

As previously stated, we crash the node Ns_3 after it has initiated 50000 *abcast* requests. Therefore, we consider each experiment to be complete when both Ns_1 and Ns_2 have delivered $(666666 + 50000 - \#violations)$ *abcast*s. For all of our experiments, we utilise the following AFC values $\delta_{min} = 1ms$ and $\delta_{max} = 10ms$. We execute our experiments utilising $\rho_{min} = 1,2,3$ and $R = 0.9999$ in order to determine the effect of increasing Δ_m on the number of messages delivered before the GM service publishes a new view and the number of order violations. Similarly, we also execute our experiments utilising $\rho_{min} = 1$ and $R = 0.99999$, to see the effect of increasing R on Δ_m and the number of ordering violations.

Once again, we utilise the same initialisation period for ABcast as in our previous experiments.

6.3.2 Results

Tables 6.6, 6.7 and 6.8 show the performance of the ABcast protocol in the experiments described in 6.3.1, when ρ_{min} is equal to $1,2$ and 3, respectively. With each table showing the results of ten experiments that were executed with the specified ρ_{min} value. Each of these tables, show the average Δ_m value calculated for messages originating at both Ns_1 and Ns_2, as well as the total number of *abcast*s, *#abcast*, delivered by Aramis in the interim period between node Ns_3 crashing and the GM publishing a new view [3]. Furthermore, the value in brackets next to this total represents the number of order violations that occurred in the interim period. Finally, *#abcast* shows the throughput gain provided by utilising the probabilistic Aramis protocol, as these *abcast*s would not have been delivered until GMS detected Ns_3's crash if a GM

[3] If a column contains $-$ it indicates that no Aramis deliveries occurred before GMS detected Ns_3's crash.

based protocol had been used for *abcast*ing.

Table 6.9 shows the performance of ABcast in the same experiments, but with $\rho_{min} = 1$ and the constant $R = 0.99999$. The fields and columns presented in this table are equivalent to those describe above.

ρ_{min}	R	# Violation Free Runs	# Violations	
			Ns_1	Ns_2
3	0.9999	10/10	–	–
2	0.9999	9/10	–	$\frac{1}{19485}$
1	0.9999	8/10	– –	$\frac{1}{12483}$ $\frac{1}{10544}$
1	0.99999	9/10	–	$\frac{1}{17016}$

Table 6.5: Summary of ρ_{min} and R when node crashes occur

Table 6.5 provides a summary of all of these previous tables, with each set of experiments represented as a single row and being uniquely identified by the combination of R and ρ_{min} values used in the experiments. For each experiment, we show the number of experiments that encountered no order violations over the total number of experiments, and in experiments where violations did occur we present the number of order violations over the number of successful Aramis deliveries that occurred before GMS detected Ns_3's crash[4].

6.3.3 Evaluation

From Tables 6.6, 6.7, 6.8 and 6.9 we can clearly see that the ABcast protocol allows for a large number of *abcast*s to be delivered in the interim period between a node crashing and the GMS protocol detecting it. With an individual node delivering, on average, greater than 10^4 *abcast*s and in one case more than double that amount. Furthermore, out of 40 experiments there was only two instances where there was no benefit to using the ABcast protocol, and this was when the protocol utilised more conservative values of $R = 0.99999$ or $\rho_{min} = 3$.

In 6.5, we can clearly see that increasing the size of ρ_{min} has a direct impact on the reliability of Aramis, as the number of order violations reaches zero when ρ_{min} is at its largest. This can be

[4]In Table 6.5, – indicates that no Aramis order violations occurred

Experiment	Ns_1		Ns_2	
	Δ_m	Aramis	Δ_m	Aramis
1	240	10544, (0)	212	10544, (1)
2	553	6874, (0)	527	6874, (0)
3	517	17452, (0)	402	17452, (0)
4	334	18487, (0)	274	18483, (0)
5	426	12483, (0)	322	12483, (1)
6	717	4723, (0)	429	4723, (0)
7	491	8936, (0)	816	8936, (0)
8	510	393, (0)	475	392, (0)
9	478	3798, (0)	931	3798, (0)
10	234	17341, (0)	290	17805, (0)
R_{ex}	1		0.9999803	

Table 6.6: Aramis deliveries (Order Violations) before GMS detects Ns_3 has crashed $R = 0.9999$, $\rho_{min} = 1$

Experiment	Ns_1		Ns_2	
	Δ_m	Aramis	Δ_m	Aramis
1	664	5509, (0)	580	5509, (0)
2	636	13697, (0)	555	13697, (0)
3	1020	2688, (0)	496	2688, (0)
4	320	19481, (0)	279	19485, (1)
5	331	19012, (0)	400	19106, (0)
6	456	2669, (0)	466	2669, (0)
7	432	10823, (0)	939	10823, (0)
8	271	18412, (0)	272	18414, (0)
9	498	5440, (0)	362	5449, (0)
10	716	3611, (0)	376	3611, (0)
R_{ex}	1		0.9999901	

Table 6.7: Aramis deliveries (Order Violations) before GMS detects Ns_3 has crashed $R = 0.9999$, $\rho_{min} = 2$

Experiment	Ns_1		Ns_2	
	Δ_m	Aramis	Δ_m	Aramis
1	452	17651, (0)	451	21064, (0)
2	475	–	679	-
3	754	3911, (0)	515	3911, (0)
4	355	16516, (0)	515	3911, (0)
5	214	17620, (0)	503	17619, (0)
6	386	12968, (0)	694	12968, (0)
7	453	7311, (0)	345	7311, (0)
8	632	12613, (0)	546	12613, (0)
9	356	18030, (0)	569	18034, (0)
10	695	13907, (0)	511	13907, (0)
R_{ex}	1		1	

Table 6.8: Aramis deliveries (Order Violations) before GMS detects Ns_3 has crashed $R = 0.9999$, $\rho_{min} = 3$

Experiment	Ns_1		Ns_2	
	Δ_m	Aramis	Δ_m	Aramis
1	387	17982, (0)	453	17016, (1)
2	7507	255, (0)	3804	742, (0)
3	2019	8117, (0)	1676	8117, (0)
4	3094	-	1899	-
5	264	10876, (0)	416	10880, (0)
6	683	9262, (0)	605	9262, (0)
7	244	18224, (0)	301	18222, (0)
8	1160	2207, (0)	830	2207, (0)
9	334	19058, (0)	278	19060, (0)
10	233	17588, (0)	421	17586, (0)
R_{ex}	1		0.9999903	

Table 6.9: Aramis deliveries (Order Violations) before GMS detects Ns_3 has crashed $R = 0.99999$, $\rho_{min} = 1$

explained by a larger ρ_{min} increasing the calculated Δ_m value for each *abcast* (as seen in Tables 6.6, 6.7, 6.8) [5].

Conversely, when we increase R from 0.9999 to 0.99999, with $\rho_{min} = 1$, the number of violations is reduced from two to one, at the expense of a greatly enlarged Δ_m (compared to $\rho_{min} = 1,2,3$ when $R = 0.9999$).

From its initial conception, ABcast has been designed with pessimistic assumptions in order to minimise the chances of Δ_m being exceeded by a given *abcast*. This pessimism is reflected in our experiments, with Tables (6.6, 6.7, 6.8, 6.9) all showing that the experienced R, denoted as R_{ex}, is greater than the user specified R. Where R_{ex} for a given set of experiments is calculated as

$$1 - R_{ex} = \frac{\sum_{1}^{10} \text{Number of Order Violations}}{\sum_{1}^{10} \text{Messages Delivered by Aramis}} \tag{6.1}$$

unless the number of messages delivered by Aramis is zero, in which case $R_{ex} = 1$.

Finally, while our experiments show that a larger number of *abcast*s are delivered in the interim period between node failures and detection, we believe that in the event of a 'real' crash this value could be much higher. In our experiments we crash the JVM instantly, which results in the TCP sockets utilised by the FD_SOCK protocol being closed immediately. This means that it is almost certainly the FD_SOCK protocol that detects the failure of Ns_3 each time. If a crash was preceded by a slowing down period where node responses become more staggered and the node was unresponsive, but still running and maintaining an open TCP socket, it is highly probable that the total number of *abcast*s sent in the interim period would be much larger, as the alternative failure detection protocol FD_ALL has a default timeout period of 40 seconds.

[5] The difference in calculated Δ_m values is not significant between $\rho_{min} = 1,2,3$ in our results, however this can be attributed to the varying state of the underlying network. Our experiments were conducted in sets based upon their constant values, e.g. all ten experiments that utilised $\rho_{min} = 1$ and $R = 0.9999$ were performed one after the other. Therefore, as these experiments take several minutes each, the time required to conduct all of the experiments was significant, and as a consequence these experiments were conducted over several days. Consequently, the load on the underlying network will have varied for each set of experiments. However, we can still attribute the reduced number of order violations to an increase in Δ_m, as this variable is calculated based upon latencies that represent the networks current state. Therefore, if a smaller ρ_{min} value was utilised under the exact same network conditions as the $\rho_{min} = 3$ experiments, we know that the calculated Δ_m value would have been significantly smaller.

6.3.4 Summary

We have found that utilising the ABcast protocol for *abcast*s allows for a significant number of messages ($> 10^4$) to be delivered in the interim period between a node crash and the GM protocol publishing a new view. Furthermore, we have found that increasing both ρ_{min} and R reduces the chances of order violations occurring in the presence of node crashes. When $\rho_{min} = 1$, R_{ex} is much larger than the specified R, with the difference increasing as ρ_{min} becomes larger. However, a larger ρ_{min} can occasionally risk Aramis not being able to deliver any *abcast*s before the GM publishes a new view, so it is recommended to keep $\rho_{min} = 1$.

6.4 Summary

In this chapter, we have presented a thorough performance evaluation of both the AmaaS model and the ABcast protocol. We have shown that, as the number of nodes involved in a transaction increases, the AmaaS model, coupled with the SCast protocol, can improve the average latency and throughput of distributed transactions when compared to the existing P2P approach.

Additionally, we have shown that the ABcast protocol can provide comparable performance to existing deterministic protocols, such as TOA, when utilised within a AmaaS service. Crucially, this performance does not come at the expense of ABcast's guarantees, as we have demonstrated that these guarantees can be met when handling large numbers of *abcast*s. Furthermore, we have shown that with the correct configuration parameters, it is possible to avoid order violations when node crashes occur. Finally, we have shown that ABcast's non-blocking message delivery enables a significant number of *abcast* messages to be delivered in the interim period between a node crashing and a GM protocol detecting it.

Chapter 7

Conclusions

This thesis explored the challenges of designing totally ordered multicast protocols for the coordination of distributed transactions in partially replicated environments.

Existing research [64] shows that utilising atomic multicast protocols in partially replicated environments to coordinate distributed transactions, opposed to the classic 2PC, can improve transaction throughput as it removes the need for distributed locks. However, a consequence of transactions being coordinated via *amcast* is that the transactional throughput of a system deteriorates linearly with the performance of the underlying *amcast* protocol; as it is the *amcast* protocol that ultimately determines the commit rate of transactions.

In this thesis we have shown that the performance of existing *amcast* protocols starts to deteriorate as the number of nodes involved in a multicast increases. This degradation occurs because the existing *amcast* protocols execute in a P2P manner, resulting in the number of nodes that require consensus increasing with the multicast's destination set. To overcome the limitations of the P2P *amcast* approach, we advocate the use of the AmaaS model, to dictate *amcast* message ordering to *amcast* participants. The key advantage of this approach being that the number of nodes required to reach consensus is constant regardless of the number of nodes participating in an *amcast*, hence as the number of nodes involved in a transaction increases the cost of consensus remains constant. Performance evaluations have shown that utilising the AmaaS approach can provide significant performance improvements, in terms of both latency and throughput, to *amcast*'s when the number of destinations becomes greater than three.

In addition to advocating the AmaaS model, another key contribution of our work has been the development of a non-blocking, low-latency, probabilistic atomic broadcast protocol called

ABcast, which was specifically designed to facilitate state machine replication within AmaaS ordering services. The development of a probabilistic consensus protocol was necessary to fully realise the benefits of the AmaaS model, as deterministic consensus protocols must all admit blocking in the event of nodes failures due to the well known FLP impossibility. Without such a probabilistic protocol, the availability characteristics of AmaaS would be poor during the reconciliation of node failures in the ordering service itself.

The remainder of this chapter is structured as follows: Section 7.1 provides a summary of the content and key findings of each chapter in this thesis. This is then followed by Section 7.2 which explores the limitations of our work. Lastly, we explore potential avenues for future research in Section 7.3.

7.1 Thesis Summary

In chapter 2 we began by examining related work in the field of atomic broadcast and multicast protocols, as well as the current state of existing coordination services, before providing an in-depth analysis of Red Hat's open source in-memory NoSQL database, Infinispan. This analysis of Infinispan is essential for understanding our work, as all of our proposed solutions have been designed within the context of Infinispan's distributed transactions, and consequently our performance evaluation was also based upon their semantics.

In chapter 3, we proposed that an alternative system model, AmaaS, should be utilised for coordinating distributed transactions in partially replicated environments. We advocate the use of an external ordering service to provide total ordering for multicast messages, as an alternative to Infinispan nodes executing atomic multicasts between themselves via the P2P model. This was followed by the introduction and analysis of the fault-tolerant atomic multicast protocol SCast. Performance evaluation in chapter 6 shows that, when the SCast protocol is utilised for coordinating transactions and the total number of destinations is greater than 3, the AmaaS model consistently outperforms the existing P2P protocols utilised by Infinispan.

Chapter 4 introduced the hybrid atomic broadcast protocol, ABcast, which was designed for state machine replication between service nodes in an *AmaaS* ordering service. ABcast leverage's a traditional deterministic protocol, Base, in conjunction with a new probabilistic protocol, Aramis, in order to create an *abcast* protocol that provides low-latency message delivery in the

absence of node failures and non-blocking message delivery in their presence.

Due to Aramis's probabilistic guarantees, the ABcast protocol can only guarantee a message's total order with a probability close to 1; hence it is possible for order violations to occur. This small probability of order violations motivated Chapter 5, which provides an analysis of potential strategies that can be adopted by the PSCast protocol and the Infinispan transaction manager, in order to mitigate the repercussions of ABcast and SCast order violations.

Finally, chapter 6 presented the results of our extensive performance evaluation and showed that the ABcast protocol is able to maintain similar levels of performance to traditional deterministic atomic broadcast protocols in the absence of node crashes, whilst providing non-blocking message delivery in the presence of such crashes. With, on average, $> 10^4$ messages being delivered in the interim period between a node crash and the group membership service publishing a new view. Furthermore, we have shown that the ABcast protocol is able to handle a large number of broadcasts (10^7) without a single order violation occurring, even when smaller values are utilised for R and ρ_{min}.

7.2 Limitations

In our performance evaluation we have shown that AmaaS is an effective model for improving the performance of atomic multicasts, however our performance evaluation only considers a case where ten clients issue requests to the service at any one time. It is inevitable that any AmaaS service will have an upper limit on the number of ordering requests that it can accommodate at anyone time. Therefore, a limitation of the AmaaS approach is that the service will eventually have to start rejecting client requests as the number of client requests becomes greater than the service's throughput capabilities. This is an inherent limitation of using a centralised service and is an acknowledged limitation of other coordination services such as Chubby [11].

Another key limitation of the ABcast protocol is the need for additional logic higher up the network, or application, stack in order to handle order violations. Examples of the additional logic required by SCast and Infinispan's transaction manager are presented in chapter 5. From these examples it is clear to see that this additional logic can require substantial redesigning of existing systems and is often far from trivial to implement. However, it is worth noting that this problem is not unique to ABCcast and similar additional logic is required if any other

probabilistic atomic broadcast or atomic multicast protocol is utilised.

7.3 Future Work

This section explores potential future research problems that have arisen from the work documented in this thesis.

7.3.1 Multiple AmaaS Services

As stated in § 7.2, it is inevitable that an AmaaS ordering service will eventually become saturated by requests if the number of clients in the system continues to increase. A possible solution to this problem is to utilise multiple ordering services simultaneously, with client nodes assigned a specific service that they must interact with. This would allow multicasts to be sent between clients, who are utilising the same service, in much the same manner as requests are handled in the existing SCast protocol. However, if a client wishes to send a multicast between clients which utilise distinct ordering services, then it is necessary for these services to share state between themselves in order to service the client's request and return accurate ordering data. Enabling two ordering services to interact in such a way is far from trivial and is an interesting research challenge. Furthermore, designing an algorithm to effectively partition clients in a way that minimises the frequency in which distinct ordering services need to interact is also non-trivial.

7.3.2 Utilising ABcast for State Replication in Zookeeper

At present, the AmaaS protocol SCast presented in chapter 3 has only been utilised in a proof of concept implementation. In order for it to be utilised in production systems, it would necessary for a standalone service that exposed an API to client nodes to be implemented. One solution to this problem would be to utilise the existing Zookeeper implementation, which is open source and released under the Apache License Version 2.0 [2], as a foundation for a new AmaaS service. The existing protocol used by Zookeeper for state machine replication, ZAB, could be replaced with ABcast. The SCast protocol could be then created as an additional application that resides on the Zookeeper nodes and utilises the underlying primitives exposed by Zookeeper for

state machine replication, in order to provide its own API for client multicasts. Such a solution would not only provide a solid foundation for the ordering service, it would also enable existing applications that depend on Zookeeper to utilise the new implementation as the Zookeeper API would remain the same [1].

7.3.3 Extending Infinispan to Support AmaaS with ABcast

The performance evaluation presented in chapter 6 compares the performance of an AmaaS service utilising PSCast, with the performance of Infinispan's existing P2P protocol TOA. In these experiments, we evaluate the performance of each approach by contrasting the performance of the underlying *amcast* protocol when executing emulated Infinispan transactions. This is an effective way of comparing *amcast* performance, however it does not take into account other factors relevant to Infinispan, such as the transaction abort rate [2], and it does not allow industry standard benchmarks such as TPC-C [67] to be evaluated.

In order to enable the measurement of such metrics, it is necessary for Infinispan to be extended to support the PSCast model. Such an undertaking would require a large engineering effort, which was unfortunately beyond the scope of this thesis, as it would require the Infinispan system to be adapted to utilise PSCast instead of TOA and for Infinispan's transaction manager be reimplemented so that it provides a recovery mechanism for handling order violations. Furthermore, as the PSCast protocol depends on an external ordering service, it would also be necessary for a standalone service to be implemented similar to the solution described in § 7.3.2.

[1] Where possible. The potential for state machine replication to fail because of order violations would require some slight changes to Zookeeper's API.

[2] Transactions can abort when the WSC fails or when a order violation has occurred in PSCast

References

[1] Aguilera, M. K. and Walfish, M. (2009). No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 3–3, Berkeley, CA, USA. USENIX Association.

[2] Apache License, Version 2.0 (2015). Apache license, version 2.0. https://www.apache.org/licenses/LICENSE-2.0. [Accessed: 18-March-2015].

[3] Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience.

[4] Bernstein, P. and Newcomer, E. (1997). *Principles of Transaction Processing: For the Systems Professional*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[5] Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221.

[6] Bertsekas, D. and Gallager, R. (1992). Flow control. In *Data Networks, Chapter 6 (2nd Ed.)*, pages 493–535. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[7] Bezerra, C., Pedone, F., Garbinato, B., and Geyer, C. (2013). Optimistic atomic multicast. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 380–389.

[8] Birman, K., Schiper, A., and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314.

[9] Brewer, E. (2012). Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29.

[10] Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA. ACM.

[11] Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA. USENIX Association.

[12] Cattell, R. (2011). Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27.

[13] Cerf, V. G. and Icahn, R. E. (2005). A protocol for packet network intercommunication. *SIGCOMM Comput. Commun. Rev.*, 35(2):71–82.

[14] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA. ACM.

[15] Cristian, F. (1996). Synchronous and asynchronous. *Commun. ACM*, 39(4):88–97.

[16] Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

[17] Di Sanzo, P., Antonacci, F., Ciciani, B., Palmieri, R., Pellegrini, A., Peluso, S., Quaglia, F., Rughetti, D., and Vitali, R. (2013). A framework for high performance simulation of transactional data grid platforms. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, pages 63–72, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[18] Emerson, R. and Ezhilchelvan, P. (2014a). A non-blocking atomic-multicast service for scalable in-memory transaction systems. Number CS-TR-1435.

[19] Emerson, R. and Ezhilchelvan, P. D. (2014b). An atomic-multicast service for scalable in-memory transaction systems. In *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014*, pages 743–746.

[20] Emerson, R. and Ezhilchelvan, P. D. (2014c). Faster transaction commit even when nodes crash. *CoRR*, abs/1404.7548.

[21] Ezhilchelvan, P., Clarke, D., and Di Ferdinando, A. (2011). Near certain multicast delivery guarantees amidst perturbations in computer clusters. Technical Report CS-TR-1267, School of Computing Science, Newcastle University.

[22] Ezhilchelvan, P. and Shrivastava, S. (2010). Learning from the past for resolving dilemmas of asynchrony. *SIGOPS Oper. Syst. Rev.*, 44(2):58–63.

[23] Ezhilchelvan, P. D., Macedo, R. A., and Shrivastava, S. K. (1995). Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, ICDCS '95, pages 296–, Washington, DC, USA. IEEE Computer Society.

[24] Ezhilchelvan, P. D. and Shrivastava, S. K. (1994). rel/rel: a family of reliable multicast protocols for distributed systems. *Distributed Systems Engineering*, 1(6):323.

[25] Felber, P. and Pedone, F. (2001). Probabilistic atomic broadcast. In *in Proceedings of the 12th International Symposium on Distributed Computing*, pages 318–332. Springer.

[26] Fidge, C. J. (1988). Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66.

[27] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.

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

[29] GridGain (2014). Gridgain: In-memory computing. http://www.gridgain.com/. [Accessed: 29-August-2014].

[30] Guerraoui, R. and Schiper, A. (2001). The generic consensus service. *IEEE Trans. Softw. Eng.*, 27(1):29–41.

[31] Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317.

[32] Han, J., Haihong, E., Le, G., and Du, J. (2011). Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366.

[33] Hazelcast (2014). Hazelcast.org. http://hazelcast.org/. [Accessed: 29-August-2014].

[34] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA. USENIX Association.

[35] Infinispan (2014). Infinispan: Distributed in-memory key/value data grid and cache. http://infinispan.org/. [Accessed: 29-August-2014].

[36] Jacobson, V. (1988). Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA. ACM.

[37] Jain, R. (1996). Congestion control and traffic management in atm networks: Recent advances and a survey. *Comput. Netw. ISDN Syst.*, 28(13):1723–1738.

[38] JGroups (2014). The jgroups project. http://www.jgroups.org/. [Accessed: 01-September-2014].

[39] JSR-107 (2014). The java community process(sm) program - jsrs: Java specification requests - detail jsr-107. https://jcp.org/en/jsr/detail?id=107. [Accessed: 01-September-2014].

[40] JTA (2014). Java transaction api. http://www.oracle.com/technetwork/java/javaee/jta/index.html. [Accessed: 02-September-2014].

[41] Junqueira, F. P., Reed, B. C., and Serafini, M. (2011). Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA. IEEE Computer Society.

[42] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). 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. ACM.

[43] Kermarrec, A.-M., Massoulié, L., and Ganesh, A. J. (2003). Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):248–258.

[44] Korth, H. F., Levy, E., and Silberschatz, A. (1990). A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 95–106, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[45] Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226.

[46] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[47] Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

[48] Lamport, L. (2001). Paxos made simple. *ACM SIGACT News*, 32(4):18–25.

[49] Lamport, L. (2006). Fast paxos. *Distributed Computing*, 19(2):79–103.

[50] Lamport, L. (2011). Byzantizing paxos by refinement. In Peleg, D., editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer.

[51] Lamport, L. and Massa, M. (2004). Cheap paxos. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 307–314. IEEE Computer Society.

[52] Marandi, P. J., Benz, S., Pedone, F., and Birman, K. (2014). Practical experience report: The performance of paxos in the cloud. *CoRR*, abs/1404.6719.

[53] Marandi, P. J., Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 527–536. IEEE.

[54] Marchioni, F. (2012). *Infinispan Data Grid Platform*. Packt Publishing, Limited.

[55] Mattern, F. (1988). Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland.

[56] Moniruzzaman, A. B. M. and Hossain, S. A. (2013). Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *CoRR*, abs/1307.0191.

[57] Nable, J. (1984). Congestion control in ip/tcp internetworks. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA. RFC-896. [Accessed: 04-February-2015].

[58] Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA. USENIX Association.

[59] Oracle Coherence (2014). Oracle coherence. http://www.oracle.com/technetwork/middleware/coherence/overview/index.html. [Accessed: 10-September-2014].

[60] Pivotal GemFire (2014). Big data | pivotal gemfire. http://www.pivotal.io/big-data/pivotal-gemfire. [Accessed: 10-September-2014].

[61] RadarGun (2014). Radargun. https://github.com/radargun/radargun/wiki. [Accessed: 25-September-2014].

[62] Red Hat (2014). The world's open source leader. http://www.redhat.com/. [Accessed: 01-September-2014].

[63] Reliable UDP (1999). Internet engineering task force - reliable udp. http://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00. [Accessed: 15-September-2014].

[64] Ruivo, P., Couceiro, M., Romano, P., and Rodrigues, L. (2011). Exploiting total order multicast in weakly consistent transactional caches. In *Proceedings of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, PRDC '11, pages 99–108, Washington, DC, USA. IEEE Computer Society.

[65] Santos, N. and Schiper, A. (2012). Tuning paxos for high-throughput with batching and pipelining. In *Proceedings of the 13th International Conference on Distributed Computing and Networking*, ICDCN'12, pages 153–167, Berlin, Heidelberg. Springer-Verlag.

[66] Schiper, N., Sutra, P., and Pedone, F. (2010). P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 214–224, Washington, DC, USA. IEEE Computer Society.

[67] TPC-C (2014). Tpc-c | transaction processing performance council. http://www.tpc.org/tpcc/. [Accessed: 25-September-2014].

[68] UFC-JGroups (2015). Unicast flow control. http://www.jgroups.org/javadoc-3.x/org/jgroups/protocols/UFC.html. [Accessed: 27-January-2015].

[69] XA (2014). Distributed transaction processing: The xa specification. http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf. [Accessed: 02-September-2014].