A Non-Blocking Atomic-Multicast Service for Scalable In-memory Transaction Systems

Ryan Emerson and Paul Ezhilchelvan
TECHNICAL REPORT SERIES

No. CS-TR-1435 October, 2014

A Non-Blocking Atomic-Multicast Service for Scalable In-memory Transaction Systems

R. Emerson and P. Ezhilchelvan

Abstract

Atomic Multicasting is central to managing replication. Our focus is on its deployment for enhancing scalability and performance of in-memory transactional systems where data replication is essential. When it is deployed as an external service, enhancement is demonstrably accomplished when no node within the service crashes. Since known atomic-multicast protocols block whenever a node is crashed or appears so, using them for service implementation risks the transaction system making no progress when blocking prevails. To eliminate this risk, our service simultaneously runs two protocols with distinct properties. The first one is the fastest when crashes are absent, but blocks until a crashed node is isolated. The second, newly-designed one never blocks but can fail to deliver messages with a probability that can be made vanishingly small in return for slower performance. The service uses slow protocol when the fast one is blocked and the switch-over is instantaneous. Extensive performance study confirms best crash-free performance, crash-uninterrupted service and no delivery failure in practical settings.
Abstract

Atomic Multicasting is central to managing replication. Our focus is on its deployment for enhancing scalability and performance of in-memory transactional systems where data replication is essential. When it is deployed as an external service, enhancement is demonstrably accomplished when no node within the service crashes. Since known atomic-multicast protocols block whenever a node is crashed or appears so, using them for service implementation risks the transaction system making no progress when blocking prevails. To eliminate this risk, our service simultaneously runs two protocols with distinct properties. The first one is the fastest when crashes are absent, but blocks until a crashed node is isolated. The second, newly-designed one never blocks but can fail to deliver messages with a probability that can be made vanishingly small in return for slower performance. The service uses slow protocol when the fast one is blocked and the switch-over is instantaneous. Extensive performance study confirms best crash-free performance, crash-uninterrupted service and no delivery failure in practical settings.

About the authors

Ryan Emerson is a final year PhD student who has studied at Newcastle University since 2007. Ryan graduated with a 2:1 BSc (Hons) in Computer Science in 2010 and obtained his Masters (ITEC) with Distinction in 2011. His research interests include atomic broadcast, middleware, scalability, fault-tolerance and cloud computing.

Paul Devadoss Ezhilchelvan received Ph.D. degree in computer science in 1989 from the University of Newcastle upon Tyne, United Kingdom. He received the Bachelor of Engineering degree in 1981 from the University of Madras, India, and the Master of Engineering degree in 1983 from the Indian Institute of Science, Bangalore. He joined the School of Computing Science of the University of Newcastle upon Tyne in 1983 where he is currently a Reader in Distributed Computing.

Suggested keywords

TRANSACTIONS
EVENT ORDERING
REPLICATION
CRASH-TOLERANCE
A Non-Blocking Atomic-Multicast Service for Scalable In-memory Transaction Systems

Ryan Emerson  
School of Computing Science 
Newcastle University, UK  
Email: ryan.emerson@ncl.ac.uk

Paul Ezhilchelvan  
School of Computing Science  
Newcastle University, UK  
Email: paul.ezhilchelvan@ncl.ac.uk

Abstract—Atomic Multicasting is central to managing replication. Our focus is on its deployment for enhancing scalability and performance of in-memory transactional systems where data replication is essential. When it is deployed as an external service, enhancement is demonstrably accomplished when no node within the service crashes. Since known atomic-multicast protocols block whenever a node is crashed or appears so, using them for service implementation risks the transaction system making no progress when blocking prevails. To eliminate this risk, our service simultaneously runs two protocols with distinct properties. The first one is the fastest when crashes are absent, but blocks until a crashed node is isolated. The second, newly-designed one never blocks but can fail to deliver messages with a probability that can be made vanishingly small in return for slower performance. The service uses slow protocol when the fast one is blocked and the switch-over is instantaneous. Extensive performance study confirms best crash-free performance, crash-uninterrupted service and no delivery failure in practical settings.

I. INTRODUCTION

Replication is the most common way of enhancing availability and performance. A data-item replicated on multiple host nodes within a cloud computing platform remains accessible even if a node crashes and it is accessed faster when the least-loaded node is used. These benefits, however, require that updates to replicated data be carried out at all host nodes in the same order. Meeting this requirement in a crash-tolerant and non-blocking manner is a challenge that is addressed here. Delivery guarantees of atomic multicasting, amcast for short, are known (e.g., [1]) to considerably simplify the task of replication management. These are:

- G1: If the source of \( m_i \) does not crash until it amcasts \( m_i \), then all operative destinations of \( m_i \) deliver \( m_i \) (validity);
- G2: If the source of \( m_i \) crashes while amcasting \( m_i \) and if any destination delivers \( m_i \), then all operative destinations of \( m_i \) must deliver \( m_i \) (uniform delivery);
- G3: If two amcasts, \( m_i \) and \( m_j \), share common destinations, then all such common destinations that deliver both \( m_i \) and \( m_j \), must deliver them in an identical order (ordered delivery).

Efficient and scalable management of replicated data is at the core of in-memory database systems, such as Red Hat’s Infinispan[2]. Performance study by [3] shows that amcasting is far more effective than using the classical 2-Phase commit. It uses Infinispan and examines all relevant metrics such as abort rate, latency, throughput and average number of nodes involved in a transaction. It employs the amcast protocol of [4] that offers the best performance compared to the popular, Paxos-style alternatives, when all destinations of an amcast are operative. If a destination crashes before or during an amcast, delivery is blocked until another protocol, viz. group membership protocol or GM for short, unblocks it. GM constructs a new group view comprising only operative nodes and ensures a virtually-synchronous closure on messages amcast by the crashed node [4]. (Infinispan uses the GM of JGroups.) Crash detection and recovery can block amcast delivery for a long duration, e.g., in the order of seconds.

To limit the scale of performance slow-down due to blocking, the set-up in [3] requires that nodes involved in a transaction also execute the amcast protocol. More precisely, let node \( c \) initiate and coordinate transaction \( Tx \); at some time during the processing of \( Tx \), \( c \) amcasts \( m \) to all nodes that host a replica of any key-value pair \((k, v)\) updated by \( Tx \). Let \( Tx.dst \) denote the set of all nodes \( m \) is destined for. As we explain in Section II, nodes of \( Tx.dst \) delivering \( m \) is a pre-condition for \( c \) to be able to decide on commit or abort of \( Tx \). In [3], nodes of \( Tx.dst \) execute the amcast protocol to deliver \( m \), in addition to participating in processing of \( Tx \).

Crash-free runs of experiments presented in Section IV-A demonstrate that when nodes of \( Tx.dst \) are made to execute the amcast protocol, transaction latency and throughput deteriorate, when the number of nodes in \( Tx.dst \) is 4 or larger, i.e., when \( |Tx.dst| \geq 4 \). Note that replication tends to increase \( |Tx.dst| \); e.g., if \( Tx \) involves 3 key-value pairs and replication is 2-fold then \( |Tx.dst| \) can be \( 3 \times 2 \). Our experiments also demonstrate that transaction performance scales up well when nodes of \( Tx.dst \) are relieved from executing the amcast protocol and amcasting is provided instead as a service by a separate set of dedicated nodes. Note, however, that if this service were to be implemented using [4] for the best crash-free performance, then the entire transaction system would be blocked by a single crash within the service.

The principal aim of our work, also funded by Red Hat, is to build an amcast service that not only retains the best crash-free performance of [4] but also does not block. To this end, a new, non-blocking protocol, called Aramis, is designed to operate in tandem with [4] which is distinguished here as the Base protocol. Base delivers amcasts when all nodes
implementing the service are operative; when a service node crashes and Base is subsequently blocked, Aramis takes over amcast delivery until GM unblocks Base.

Ideally, Aramis should be used only when a service node actually crashes; in practice, it is used whenever responses are found to be absent for ‘too long’ and a crash suspicion is triggered. In cloud environments, modeled commonly as asynchronous [6], message transfer delays can fluctuate, leading to premature timeout expiry and false suspicions. To eliminate false suspicions, GM uses long and pessimistic timeouts, e.g., 10 seconds in JGroups, at the expense of a prolonged blocking period when a crash does occur. Using only reasonable timeout periods, our service admits false suspicions and uses Aramis as often as suspicions arise. Since switching between Base and Aramis is designed to be a lightweight, local operation, it imposes no computational or communication overhead.

That false suspicions are inevitable leads to the well-known FLP impossibility [7]: a deterministic, asynchronous protocol must either admit blocking to meet guarantees G1-G3 or permit a likelihood of guarantees not being met to be non-blocking; i.e., it must sacrifice one - either liveness or safety - to guarantee the other. Known protocols are of two categories: GM dependent (like Base, [8]) or quorum-based [5]. G1-G3 are assured in both; the quorum-based ones block mildly due to false/valid suspicions and hence potentially often, and GM-dependent ones block severely but only when crashes do occur.

Aramis is new and designed differently (and hence named after the third, youngest musketeer.) It is non-blocking and compromises on G1-G3 with a probability that is close to 0; the load imposed on the service has to be pushed to an extreme to observe a probability of $2.7 \times 10^{-4}$ (Table I). Further, Aramis is used only when Base cannot be. So, the probability of the service failing an amcast $m_i$ is the product of two very small probabilities: $m_i$ not being delivered by Base and Aramis failing $m_i$. Thus, in extreme load conditions, the service failure probability observed was $2 \times 10^{-7}$.

The paper makes three contributions: a novel, non-blocking amcast protocol, an amcast service for scalable performance of in-memory transaction systems and an extensive performance study. It is organized as follows. The next section provides the background on the role of amcast in in-memory transaction systems, a description of Base protocol and the motivation for Aramis and its design approach. Section III presents Aramis design features together with the base protocol. Performance evaluation is presented in § IV, which is followed by related work and conclusions in Section V.

II. BACKGROUND AND MOTIVATION

**Amcast Guarantees** G1-G3 ensure that multicast delivery is atomic with regard to both delivering destinations and delivery order: any given multicast $m_i$ is delivered either at all of its operative destinations or nowhere, with the latter constrained to occur only when the source of $m_i$ crashes in such a way that no destination can deliver $m_i$; further, all destinations that deliver any two multicasts, $m_i$ and $m_j$, do so in the same order: either $m_i$ before $m_j$ or vice versa.

Guarantee statements distinguish receiving an amcast $m$ from delivering $m$ (to higher level applications). Delivery is an one-time, irreversible operation: once $m$ is delivered, it cannot be ‘undelivered’ and any violation of delivery guarantees cannot be undone at amcast level. So, meeting G1-G3 requires addressing two challenges, C1 and C2, stated below.

Let $m$ be amcast to a set of destinations, $m.dst$ which, by convention, includes the source of $m$, $m.o$, as well. (So, $m.o$ ‘receives’ $m$ from itself when it amcasts $m$.)

- **C1**: If an operative $d \in m.dst$ receives $m$, then every operative $d' \in m.dst$ must be enabled to receive $m$.
- **C2**: Every $d$ that receives $m$ must deduce a safe moment to deliver $m$ so that G3 is not violated.

**Base and Blocking.** In Base, C1 is addressed by each $d' \in m.dst - \{m.o\}$ acknowledging $m$ by sending $ack_d(m)$ to every $d \in m.dst$; C2 by having $m$ and each $ack_d(m)$ tentatively timestamped with a value that is one more than the timestamp ever seen or used by the respective source [9]. 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. When $d$ delivers every received $m$ as per (finalized) $m.ts$, all guarantees are met. Proofs are in [9], [4], [8] and the intuition is given below.

Since $m.ts$ 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 GM 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 $m'.ts < m.ts$.

Thus, the longer GM takes to detect and isolate a crashed node (such as $d'$ above), the larger is the number of nodes (such as $d$) that are likely to receive an amcast (such as $m'$) whose destination set includes the crashed one and, at each such node, the larger is the number of finalized amcasts (such as $m$) blocked from delivery. Any mechanism that delivers amcasts in the interval between a node crash and subsequent crash isolation will surely improve performance at application level. Further, GM can also employ long, conservative timeouts and eliminate false suspicions. These aspects motivate Aramis to be used as a non-blocking back-up for Base.

Finally, when no $d \in m.dst$ is crashed, Base finalizes $m.ts$ within $2 \times x_{mx}$ time after $m$ is amcast, where $x_{mx}$ is the maximum message transfer delay between nodes of $m.dst$. If delivery of $m$ is not blocked due to crashes elsewhere, latency is $2 \times x_{mx} -$ the smallest achievable. Further, message cost can be just one multicast (by $m.o$) if $ack(m)$ can be piggybacked.

**Base Alternatives** are quorum-based, Paxos-style (e.g., [5]), leader driven protocols (e.g., [10], [11]). Of a minimum of 3 Paxos machines, one is appointed as the leader which alone
proposes a timestamp for each \( m \) and confirms that timestamp once it receives \( ack(m) \) from a quorum of machines. Thus, an amcast involves 3 communication phases and latency can be \( 3 \times x_{\text{mx}} \). When the leader is suspected, falsely or validly, responsibility is transferred to another machine during which time amcast delivery is not possible.

**Transactional Caches and Replication.** In-memory database systems involve the database being partitioned on several nodes in a cluster and, for fast data access, partitions are also cached in the RAM of hosting nodes. For availability in the presence of node crashes, each partition is replicated on distinct nodes. In-memory databases hence offer a superior performance (through fast data access and asynchronous disk writes) and can also scale up dynamically (simply by increasing the number of nodes used for hosting database partitions). These benefits are due in part to the emergence of simpler data models, e.g., key-value pairs vs. relational.

The exact nature of using amcasts in transactional caches differs depending on the consistency criteria being used. For details, we refer the reader to [3], [1]; here, we only provide a brief description to highlight that amcast delivery is a precondition for write/update transactions to complete.

Consider a key-value pair \((k, v)\) replicated on two nodes, \( d \) and \( d' \). Let \( Tx_i \) and \( Tx_j \) be two transactions coordinated by \( c_i \) and \( c_j \); let both involve, say, updating \((k, v)\). Having executed their transactions, \( c_i \) and \( c_j \) amcast \( m_i \) and \( m_j \), respectively, containing their respective transaction operations to all nodes that host a key-value pair involved. Thus, \( \{d, d'\} \subseteq m_i, \text{dst} \) and \( \{d, d'\} \subseteq m_j, \text{dst} \). Thanks to amcasting, if \( d \) and \( d' \) both deliver \( m_i \) and \( m_j \), they do so in the same order.

When \( d \) and \( d' \) deliver, say, \( m_i \), they may evaluate some consistency criteria, e.g., write-skew check [12], to decide whether the operation on \((k, v)\) as indicated in \( m_i \) is valid and inform \( c_i \) of the evaluation outcome. Having known the outcome on each such \((k, v)\), \( c_i \) then decides if \( T_i \) needs to be aborted or committed. We note that these consistency criteria are deterministic in nature and, hence, \( d \) and \( d' \) will evaluate identically. So, \( c_i \) needs only one response on \((k, v)\) and can decide even if, say, \( d' \) is crashed.

### A. Motivation

Known techniques and protocols seek to minimize, not eliminate, blocking. *Infinispan*, for example, uses primary-backup replication structure: one of the replicas of \((k, v)\), say, \( d \) acts as primary and \( d' \) as backup; and only \( d \) takes part in Base (also acting on behalf of, and updating \( d' \)) or in 2-phase commit (which is also a blocking protocol). While a crash of \( d' \) is masked from Base/2PC execution, if \( d \) crashes, fail-over to \( d' \) as the primary still requires accurate crash detection and a prolonged blocking. In genuine amcasting [13], \((k, v)\) is replicated on three nodes, say, \( \{d, d', d''\} \), which execute a quorum-based protocol to agree on a single \( ack(m) \) to be sent on behalf of this entire replica sub-group.

Existing approaches require a crash or a suspicion to be dealt with first, before amcast delivery can be resumed. By designing Aramis as a non-blocking, back-up protocol with easy switch-over, amcast delivery can continue unhindered while a crash/suspicion is being dealt with.

It is common, e.g., in [1] as in [3], to have \( c_i \) and all nodes that host a \((k, v)\) involved in \( T_i \) to execute an amcast protocol amongst themselves to deliver \( m_i \). As our performance evaluation demonstrates, this setup leads to performance degradation as the transaction system scales up - thus undermining a distinct advantage, elastic scalability, in using transactional caches. This motivates us to opt for an amcast service using both Base (as the primary protocol) and Aramis (as back-up).

The service is implemented using a set of dedicated nodes which execute amongst themselves the Aramis/Base protocol on behalf of clients, such as \( c_i \) and \( c_j \), that initiate and coordinate transactions. This approach has two other advantages, if transaction coordinators keep the service busy which is likely as the system scales up: several client requests can be bundled into a single amcast \( m \) in protocol execution; and, \( ack(m) \)s can be piggybacked, reducing message cost considerably.

The FLP impossibility means that non-blocking Aramis cannot, in theory, meet delivery guarantees with certainty. Though delay fluctuations in an open network can be arbitrary, our findings within a cluster environment show that most deviations of \( x_{\text{mx}} \) with respect to past observations can be handled through tailored design features and the probability of the service violating delivery guarantees is reduced, in practice, to zero. These findings and design efforts are presented next.

### III. Amcasting Service by Aramis/Base

Our amcast protocol that combines Aramis and Base is called the ABCast protocol and the service ABService. The latter is implemented using \( n > 1 \) dedicated nodes called service nodes, or simply s-nodes, and denoted as \( N_s, 1 \leq s \leq n \). Another, possibly large, set of nodes in the same cluster implement an in-memory database system. At any moment, a subset, \( Tx_i, \text{dst} \), of these latter nodes execute a transaction \( Tx_i \) with \( c_i \in Tx_i, \text{dst} \) acting as the transaction initiator/coordinator for \( Tx_i \). A node concurrently coordinates several transactions.

When \( Tx_i \) execution completes, \( c_i \) selects some s-node and forwards \( m \) with \( m_i, \text{dst} = Tx_i, \text{dst} \) to the selected s-node, say, \( N_s \), for amcasting \( m \) to \( m_i, \text{dst} \). \( N_s \) actually amcasts \( m_i \) only to other s-nodes using ABCast which leads to all operative s-nodes delivering \( m_i \). When \( N_s \) delivers \( m_i \), it disseminates \( m_i \) to every node \( d \) indicated in \( m_i, \text{dst} \) together with two types of order related information: \( m_i, \text{ts} \) agreed by s-nodes and immediate predecessor. The latter is the identity of \( m_j \) whose delivery must precede immediately before delivery of \( m_i \). More precisely, \( d \in m_i, \text{dst} \) must not deliver \( m_i \) until it delivers \( m_j \), and no amcast other than \( m_i \) must be delivered immediately after \( m_j \) is delivered.

Note that the immediate predecessor of \( m_i \) with respect to all amcasts directed at a given \( d \) - not just those that originate from \( c_i \) nor just those that are handled only by \( N_s \). Thus, it is specific to each \( d \) and ensures that delivery at every \( d \) is as per the finalized \( m_i, \text{ts} \). To illustrate this, let \( c_i \) submit \( m_i \) to \( N_s \), \( c_j \) submit \( m_j \) to \( N_s \) and \( d \in m_i, \text{dst} \cap m_j, \text{dst} \). Say,
s-nodes order $m_j$ before $m_i$. If $d$ receives $m_i$ before $m_j$, it will not deliver $m_i$ until it delivers $m_j$.

**ABService Guarantees.** The terms ‘source’ and ‘destinations’ in guarantee statements G1 to G3 of Section I now refer only to s-nodes. Recall also that a source s-node is a member of $m_{dst}$ when it multicasts $m$ to other s-nodes. For any $m$,

- G3 is preserved; and,
- G2 and G1 with a high probability (1-$Q_S$).

$Q_S$ is the service failure probability and must be negligibly small (zero, if possible). Recall that an s-node $N_s$ delivers $m$ by either Aramis or Base; the latter preserves all G1-G3 and the former is used only when $N_s$ suspects that Base is blocked and concludes that $m$ is ready to be delivered by Aramis. Let $P(m \mid \text{suspect})$ denote the probability that an s-node suspects blocking of Base while handling $m$, and $Q_A$ the failure probability for Aramis. So, $Q_S = P(m \mid \text{suspect}) \times Q_A$. ABCast design seeks to minimize $Q_A$.

**Implications of non-zero $Q_S$.** Suppose that $d$ and $d'$ host replicas of $(k, v)$ and $\{d, d'\} \subseteq m_{dst} \cap m_{j,dst} \cap m_{1,dst}$. Let the immediate predecessor of $m_i$ be $m_j$ and $m_l$ respectively for $d$ and $d'$. Suppose that $c_i$ chooses $N_s$ for $m_{aplacating} m_i$ and $c_j$ chooses $N_s'$ for $m_j$. Let $N_s$ fail to deliver $m_j$. It will incorrectly compute $m_l$ as the immediate predecessor of $m_i$ for $d$ and $d'$. Let $N_s'$ suffer no delivery failure; it will also compute (but correctly) $m_l$ as the immediate predecessor of $m_j$. One of two outcomes is possible.

**Case 1:** Both $d$ and $d'$ receive $m_j$ before $m_i$ (or vice versa). Both cannot deliver the received one, say, $m_j$; If this exception is handled by aborting $Tx_j$, this abort is unnecessary, but no state divergence occurs. **Case 2:** If $d$ and $d'$ receive $m_i$ and $m_j$ in different order, state of $(k, v)$ can diverge and exception handling involves state reconciliation. In this paper, we only focus on efforts to have $Q_S$ close to 0.

### A. ABService Assumptions and Rationale

**Assumption A1.** At most $(n - 1)$ of $n$ s-nodes can crash. However, 2 or more s-nodes cannot crash within an interval of some finite duration $D$ that is smaller than a few seconds.

**Assumption A2.** When an operative s-node multicasts $m$ to all other s-nodes, all operative ones receive $m$.

We use reliable UDP service of JGroups for multicast (one-to-many communication) support which meets A2. When a node crashes while multicasting $m$, reliable UDP alone is not sufficient to avoid some nodes not receiving $m$. So, a reliable multicast support of [14], rmeast for short, is deployed to address C1 of Section (§) II. It uses redundant multicasts and is described in § III-C.

**Assumption A3.** At any moment, clocks of any two operative s-nodes are within $\epsilon$ with probability $P_s \geq (1 - 10^{-5})$.

To meet A3, s-nodes implement the well-known, probabilistic algorithm of [15]. For the parameter values that our implementation used, $\epsilon$ is estimated to be 1 millisecond (ms). A major component of $\epsilon$ is the worst-case allowance for clock drift between successive synchronization. The longer the synchronization interval, the larger needs to be the allowance.

Estimation of $\epsilon = 1$ ms assumes an interval of 45 minutes, but we use a shorter 7-minute interval to increase $P_s$.

**Assumption A4.** Let random variable $x$ denote the message communication delay between any pair of operative s-nodes. The maximum delay, $x_{mx}$, estimated by observing $NT_p$ number of transmissions in recent past, is not exceeded during $NT_F$, $NT_F \leq NT_p$, number of future transmissions to unfold next, with a high probability ($1 - q$).

A4 is motivated by a series of experiments conducted in a variety of clusters, including public clouds. Each experiment, repeated at least 10 times in a given cluster, consists of 5 randomly chosen nodes multicasting to each other once every $\tau$, $1 \leq \tau \leq 10$, seconds; of $2 \times NT_p$ transmission delays observed, the cumulative distribution functions (CDFs) of first and second halves were compared using Kolmogorov-Smirnov (K-S) test. The first half thus represents the recent past and the second half the immediate future with $NT_F = NT_p$. Around 90% of K-S tests affirm that CDFs are the same at 95% confidence interval.

Figure 1 (for $\tau = 10$) depicts the typical pattern observed: the K-S test pass rate increases with $NT_p$, irrespective of cluster choice and $\tau$. When CDFs are not the same, more of future delays were either larger or smaller relative to past ones, leading to the possibility of $x_{mx}$ (of recent past) being exceeded or too high (in near future), respectively.

Being conservative, we use $NT_F = 10\%$ of $NT_p$ and $NT_F = 1000$; so, an s-node freshly estimates $x_{mx}$ for every 100 new delays it observes. Each fresh estimation of $x_{mx}$ is followed by estimating $q$ which assumes that past delays uniformly increase by 5%. So, $q$ is the ratio of the number of observed delays that exceed 95% of $x_{mx}$ to $NT_F = 1000$; thus, when more delays are observed closer to $x_{mx}$, $q$ tends to be large. Note that $q$ cannot be accurately estimated, as it relates to future delays. Therefore, Aramis uses $q$ estimates in a manner that accounts for potential inaccuracies and only for estimating one other parameter (see § III-C).

### B. ABService Components

Figure 2 depicts the components of ABService within each s-node. Delays measurement Component (DMC) estimates $x_{mx}$ and $q$ as stated earlier. Flow-control component (FCC) implements a rate-based scheme [16]: if DMC records large delays exceeding the current $x_{mx}$ estimate, FCC enforces a
lower amcasting rate until a new \( x_{mx} \) estimate is made by taking these large delays into account. FCC thus seeks to preserve the validity of prevailing \( x_{mx} \) and \( q \) estimates.

Reliable multicast support (rmcast) allows an s-node to multicast \( m \) to all other s-nodes. It meets two requirements: challenge C1 is addressed even when the multicasting s-node crashes (R1), and there is a high probability with which an operative s-node that receives \( m \) does so within \( D \) time interval after \( m \) was multicast (R2). Given its significance, rmcast protocol is described in detail.

C. rmcast protocol

The rmcast component in \( N_s \) multicasts \( m (\rho + 1) \) times, where \( \rho \) is the redundancy estimate of \( N_s \) for \( m \); redundant multicasts of \( m \) are identical except for \( m.copys = 0, \ldots, \rho \). Successive ones are made \( \eta \) time apart so that their transmission delays remain independent (random) variables. \( N_s \)'s estimates of \( \rho \) and \( \eta \) are included in \( m \) as \( m.\rho \) and \( m.\eta \).

Rmcasting of \( m \) is a success if every operative s-node receives at least one copy. Any destination \( N_s \) that receives \( m.copys = k < \rho \) cooperates to ensure this success: it waits to receive \( m.copys \geq k + 1 \) within a timeout \( \eta + \omega \), where \( \eta = m.\eta \) and \( \omega \) is \( N_s \)'s estimate of ‘jitter’ and is also indicated in \( m \) as \( m.\omega \). If the timeout expires, \( N_s \) assumes that \( N_s \) is crashed while multicasting \( m.copys = k \) and starts multicasting \( m.copys = k, k + 1, \ldots, \rho \) on behalf of \( N_s \). To ensure that no one else is acting on behalf of \( N_s \), \( N_s \) adds a further random wait, \( \zeta \), uniformly distributed on \((0, \eta)\), before acting. This and other seniority mechanisms employed in [14] are effective in preventing multiple nodes multicasting \( m \).

Note that if \( N_s \) does not crash or if it crashes and an operative s-node receives \( m \), then \( m \) is multicast at least \((\rho + 1) \) times in crash-free manner. Though \( \rho = 1 \) is sufficient for crash-tolerance, we estimate it so that at least one copy of \( m \) reaches all operative s-nodes other than \( N_s \) within \( x_{mx} \) delay and with a probability > 99.999%; more precisely, \( \rho \) is estimated as the smallest integer that satisfies \( \rho \geq 1 \) and

\[
(1 - q^{\rho+1})^{n+1} > 0.9999,
\]

which can be re-arranged as:

\[
\rho > \left(\frac{\ln(1-R)}{\ln(q)}\right) - 1 \text{ where } R = (0.9999)^{\frac{1}{n+1}}.
\]

The inequality assumes that \( m \) is multicast exactly \((\rho + 1) \) times in crash-free manner and all \( n+1 \) intended recipients are operative. Both assumptions lead to a conservative estimate of \( \rho \). Moreover, for a given \( R \), an integer \( I = 0, 1, \ldots \) satisfies

\[
I < \left(\frac{\ln(1-R)}{\ln(q)}\right) - 1 < I + 1 \text{ for a wide range of } q \text{ values}; \text{ e.g., for } R \approx 0.9999, \left(\frac{\ln(1-R)}{\ln(q)}\right) - 1 < 1 \text{ for all } q < 0.01 = 1\%.
\]

small inaccuracies in estimating \( q \) may not adversely affect \( \rho \) estimates. In our experiments, \( \rho = 1 \) is typical.

1) Probabilistic Timeliness Guarantees: To indicate how R2 is met, suppose that \( N_s \) invokes rmcast for \( m \) at its clock time \( ts \) which is also timestamped on \( m \) as \( m.ts \). Let time be measured as per \( N_s \)'s clock in the rest of this subsection.

Consider first the case where \( N_s \) does not crash. Let \( D_1 = x_{mx} + \rho \eta + g_{D_1} \), be the probability that an operative destination \( N_s \) does not receive any copy of \( m \) at or before time \( ts + D_1 \). Suppose that no s-node multicasts \( m \) on behalf of \( N_s \). Let \( P(x > \xi) \) be the probability that a copy of \( m \) takes longer than \( \xi \) time to reach a destination \( N_s \). So, the probability that none of the copies of \( m \) multicast at \( ts, ts + \eta, \ldots, ts + \rho \eta \), reaches \( N_s \) by time \( ts + D_1 \), is:

\[
g_{D_1} = P(x > D_1) = P(x > D_1 - \eta) \times \cdots \times P(x > D_1 - \rho \eta).
\]

Since \( P(x > D_1 - \eta) = P(x > x_{mx}) = q \) and \( P(x > \xi) \) decreases as \( \xi \) increases, we have \( g_{D_1} < q^{\rho+1} \). Recall that \( q \) is the probability that a delay in \( NT_P = 100 \) future transmissions exceed \( x_{mx} \) observed in the past \( NT_P = 1000 \) transmissions. Even if \( q \) is as high as 1%, we have \( g_{D_1} < 10^{-4} \) when \( \rho = 1 \). If some destination acts on behalf of \( N_s \), \( \rho \) effectively increases and \( g_{D_1} \) reduces further.

When \( N_s \) does not crash, a crash of \( N_s \) cannot undermine an operative \( N_s \) from receiving \( m \). So, the probability that all operative s-nodes receive \( m \) at or before time \( ts + D_1 \) is

\[
(1 - Q_1) = (1 - g_{D_1})^{n+1} > (1 - q^{\rho+1})^{n+1}.
\]

Suppose now that \( N_s \) crashes before completing redundant transmissions and \( n > 2 \). Consider the worst case that only one s-node, \( N_s \), has \( m \) with \( m.copys = 0 \). (If \( N_s \) has \( m.copys > 0 \), then \( N_s \) crashed only after it completed multicasting earlier copies and some s-node other than \( N_s \) also has \( m \).)

If \( cp \) 0 takes at most \( x_{mx} \) to reach \( N_s \), which occurs with probability \((1 - q) \), \( N_s \) would start acting on behalf of \( N_s \) at or before time \( ts + x_{mx} + \eta + \omega + \zeta \). Setting \( \zeta \) to the largest value it can take, \( \eta \), let us define:

\[
D = x_{mx} + 2\eta + \omega + D_1.
\]

By A1, no other s-node crashes at least until \( ts + D \). Thus, \( N_s \) disseminates \( m \), like \( N_s \) in the crash-free case, to operative destinations which can now be at most \((n-2)\); so, all operative nodes receive \( m \) at or before \( ts + D \) with a probability \((1 - Q) = (1 - g_{D_1})^{n-2} \times (1 - q) > (1 - q^{\rho+1})^{n-2} \times (1 - q) \).

To put the two cases together, let boolean \( \beta = 1 \) if \( n > 2 \) and 0 if \( n = 2 \); further, let \( D_m = \beta \times (x_{mx} + 2\eta + \omega + D_1) \).

When \( N_s \) multicast \( m \) at its clock time \( ts \), if some operative s-node receives \( m \), then every operative s-node receives \( m \) at or before time \( ts + D_m \) (as per \( N_s \)'s clock) with a probability \( \beta \times (1 - Q) + (1 - \beta) \times (1 - Q_1) \). R1 and R2 are thus met.

2) Estimating \( \eta \) and \( \omega \): Whenever DMC takes fresh estimates of \( x_{mx} \) and \( q \), it estimates \( \eta \) conservatively as the largest delay in \( n - 1 \) transmissions (of a given copy \( m \)) with probability \( \alpha = 0.99 \). Assuming exponential distribution, \( \eta = -\bar{x}[\ln(1 - \alpha)] \) where \( \bar{x} \) is the mean of \( NT_P \) observed delays. \( \omega = \eta - \bar{x} \). (If \( n = 3, \eta = 5.3 \times \bar{x} \) and \( \omega = 4.3 \times \bar{x} \).)

D. ABCast Protocol

Core principles are as follows. When any operative s-node \( N_s \) receives \( m \) sent by \( N_s \) with \( m.ts \), all operative ones receive
m (with a high probability) by time \( m.ts + D_m \) as per \( N_s \)’s clock; if each recipient piggybacks \( ack(m) \) onto one of its own \( rmcast \) messages within \( A_d \) time after receiving \( m \), these \( ack(m) \)s are very likely to be received by time \( m.ts + 2 \times D_m + A_d \) as per \( N_s \)’s clock; by A3, \( N_s \)’s clock reads no later than \( m.ts + 2 \times D_m + A_d + \epsilon \) with probability \( P_\epsilon \) when \( N_s \)’s clock reads \( m.ts + 2 \times D_m + A_d \). So, if any \( s \)-node cannot deliver \( m \) by Base until its clock time \( m.ts + 2 \times D_m + A_d + \epsilon \), delivering \( m \) after that local time by Aramis rules meet G1-G3 with a high probability. The protocol has three rules.

**Send Rule.** When \( N_s \) receives \( m \) from some coordinator \( c \), it sets \( m.o = N_s, m.ts = clock m.seq# = local sequence number, and m.xmz, m.p, m.q \) and \( m.\omega \) to its respective estimates; \( m \) is then entrusted to \( rmcast \), possibly after a small wait to adhere to the send rate decided by FCC.

**Acknowledgement Rule.** When \( N_a \) receives \( m \), it prepares \( ack(m) \) as containing only the tuple \( \{m.o, m.ts, m.seq\} \).

(Unlike Base, no tentative ts is proposed for \( ack(m) \).) If \( ack(m) \) cannot be piggybacked on any \( m’ \) that \( N_a \) itself \( amcasts \), within \( A_d \) time after \( m \) is received, it is explicitly multicast using UDP (see Fig 2). \( A_d \) is set as \( A_d = 2\eta + \omega \).

Note that when \( N_a \) receives \( m \), it may not have received an earlier \( amcast \), say, \( m” \) sent by \( N_c \) with \( m”.seq# < m.seq# \); the rule does not require that \( ack(m) \) be held until any such \( m” \) is received. Further, by receiving \( m \), \( N_a \) learns the existence of \( m” \) and this is termed as \( N_a \) knows of \( m” \). Similarly, when \( s \)-node \( N_r \) receives \( ack(m) \) from \( N_a \), it knows of \( m \) if it has not received \( m \) (yet). No node can however acknowledge \( m \) until it receives \( m \).

**A known message.** A message \( m \) is known to \( N_a \) if either \( N_a \) has received \( m \) or \( N_s \) knows of the existence of \( m \).

**Total order \( < \).** A \( \langle m \rangle \) holds when either \( m’ \langle ts < m.ts \) or \( m’.ts = m.ts \wedge m’.o < m.o \). Given that no \( s \)-node initiates two \( amcasts \) with the same timestamp, \( < \) is a total order [9] on all \( amcasts \) launched within the service.

**Delivery Rule.** Let \( \Delta_m = 2 \times D_m + A_d + \epsilon \). \( N_s \) delivers any \( m \) only after \( D1 \) and \( D2 \) below are satisfied. \( D1 \) requires any one of \( D1_A \) (Aramis) or \( D1_B \) (Base) to be met.

\begin{align*}
D1 \quad & \text{clock of } N_s > m.ts + \Delta_m \text{ (D1_A) or } m \text{ is acknowledged by all } s \text{-nodes other than } m.o \text{ (D1_B);} \\
D2 \quad & \text{all known } m' \text{ have been delivered.}
\end{align*}

**Safety Features.** When \( N_s \) knows of \( m’ \), by \( D2 \), delivery of \( m \) is held until \( m’ \) is received. By \( rmcast \), A1 and A2, \( N_s \) is guaranteed to receive \( m’ \) eventually. Thus, \( N_s \) meets G1 and G2 for such \( m’ \) it knows of, even if \( m’ \) takes arbitrarily long time \((> D_m)\) to reach \( N_s \). Further, DMC measures delay \( \Delta \) whenever copy 0 of \( m \) is received from \( m.o \), say, at local clock time \( T_x: x = T_x - m.ts + 2\epsilon \). Only one \( \epsilon \) needs to be added to (or subtracted from) \( T_x - m.ts \) if the sender’s clock is ahead of (or behind respectively) the receiver’s (by at most \( \epsilon \) due to A3); as these cases cannot be discerned, \( 2\epsilon \) is conservatively added in all cases.

**IV. EVALUATING ABService**

**ABService** was implemented using \( n = 2 \) and 3 \( s \)-nodes which are commodity PCs of 2.80GHz Intel Core i7 CPU and 4GB of RAM, running Fedora 19 and communicating over Gigabit Ethernet. The \( s \)-nodes are a part of a large, university cluster and hence communication delays between them can be quite volatile as they are influenced by other network traffic and by jobs launched on \( s \)-nodes by other users.

Ten other nodes, called \( client (c) \) nodes, in the same cluster implement a transaction system. Each \( c \)-node operates 25 concurrent threads to initiate and coordinate transactions and a transaction \( T_z \) involves a set \( T_z, dst \) of 3, 4, …, \( 10 \) \( c \)-nodes (including its coordinator). Each \( T_z \) is write-only and hence requires \( amcast \) for completion. A thread coordinating a transaction starts the next one as soon as it dispatches commit/abort decision for the current one. Thus, at any moment, 250 transactions are in different stages of execution.

A coordinator thread submits its \( amcast \) request for \( T_z \), denoted as \( r(T_z) \), with some \( s \)-node; the latter stores such requests in \( Amcast Request Pool (ARP) \) in the arrival order. The \( Send \) thread that implements the \( send \) rule, bundles some or all of these requests in ARP in the arrival order into an \( m \) with 1kB payload (with padding if necessary); it then \( amcasts \) \( m \) after any wait period prescribed by FCC and cycles back.

**Send thread waits if ARP is empty and resumes bundling once ARP becomes non-empty. Thus, the number of requests bundled in any \( m \) varies depending on request arrival rate relative to the send rate determined (dynamically) by FCC. The larger the permitted send rate relative to request arrival rate, the longer the \( Send \) thread waits for ARP to become non-empty and the fewer are the requests in a given \( m \).

When an \( s \)-node delivers \( m \), it decides \( ts \) for each \( r(T_z) \) bundled in \( m \) as: \( r(T_z).ts = m.ts + \eta \times sequence number of r(T_z) \) within the bundle, where \( \eta \) is the append operator. If the \( s \)-node itself is \( m.o \), it computes the immediate predecessor of \( r(T_z) \) for each \( d \in T_z.dst \); \( r(T_z) \) is then sent to each \( d \) together with \( r(T_z).ts \) and predecessor information.

Prior to accepting requests from \( c \)-nodes, \( s \)-nodes go through a ‘warm-up’ phase lasting about 1-2 seconds during which clocks are synchronized, \( NT_P = 10^3 \) messages of 1kB payload are exchanged and their delays observed, and the initial estimates of \( x.mz, q, p, \eta \) and \( \omega \) are taken.

Evaluation focusses on 4 issues: (i) advantages over the \( peer-to-peer (P2P) \) approach of \( T_z, dst \) nodes executing Base amongst themselves for \( T_z \) (BaseP2P); (ii) performance comparison between \( ABService \) and \( BaseService \) implemented using only Base; (iii) use of Aramis when no crash occurs (i.e., \( P(m | \text{suspect}) \)); and (iv) Aramis failures (\( QA \)).

**A. Performance Evaluation and Comparison**

Here, issues (i) and (ii) are examined by measuring latency and throughput. Former is the time elapsed between a \( c \)-node transmitting to some \( s \)-node a \( r(T_z) \) and all members of \( T_z.dst \) receiving that \( r(T_z) \) with \( r(T_z).ts \) and predecessor information. In BaseService, bundling was done without padding, since no estimation of \( x.mz \) is required and hence \( m \) need not be of fixed size. (For fairness, maximum payload is kept 1kB.) In BaseP2P, latency is the duration for all peers to
complete execution. Throughput is measured as the average number of amcasts delivered per second by a c node.

Fig. 3. Comparison of Latency

Fig. 4. Comparison of Throughput

Figures 3 and 4 show latency and throughput, with 2N and 3N denoting n = 2 and n = 3. Each point is an average of 3 crash-free trials; a trial consists of each c node completing 10^4 transactions for a specific value of |T_x.dst|. Thus, ABService and BaseService receive a total of 10^5 amcast requests in each trial. In BaseP2P, each c node initiates 10^4 Base executions with its peers and the steady throughput in Figure 4 as |T_x.dst| → 10 suggests an absence of node saturation.

Referring to Fig 3, as |T_x.dst| ≥ 4, BaseP2P’s amcast latencies increase considerably, indicating that amcast is best provided as a service for scalable performance. For n = 3, BaseService latencies (in green) were marginally smaller than ABService ones (in red) for small |T_x.dst|, with the maximum difference being about 0.25 ms (16.6%) when |T_x.dst| = 6, and the differences virtually disappear as |T_x.dst| > 6. When n = 2, differences are much smaller with ABService (in blue) appearing to be faster or equally fast for all |T_x.dst|. Comparing throughput in Figure 4 leads to similar conclusions.

Summary: Amcasing is best offered as a service to a large scale transaction system. Additional services used by, and redundant multicasting within, ABcast protocol does not incur any performance penalty when n = 2 or 3.

B. ABService Failure Probability Evaluation

In all the experiments involving ABcast, delivery by Aramis did not occur at all (issue (iii)); all m met D1_B (of receiving all ack(m)) before D1_A of Aramis. That is, \( P(m | \text{suspect}) = 0 \) for ABService with n = 2, 3.

We forced \( P(m | \text{suspect}) = 1 \) by removing D1_B from the delivery rule (see § III-D). Thus, D1 can be met only by meeting D1_A of Aramis. (s-nodes still sent ack(m) as before.) The experiment for n = 3 was repeated with this modification and no Aramis failure was observed, i.e., \( Q_A = 0 \) with \( P(m | \text{suspect}) = 1 \). As expected, latencies were large, and they were so large that an experiment (involving 10^5 transactions) took about 30 minutes to complete! Since GM can certainly deal with a crash in a much shorter duration, this D1_A-only experiment is taken to conclude that Aramis would commit no guarantee violation, had a crash occurred.

1) An Infinite Client System: We explored the underlying reasons for \( P(m | \text{suspect}) = 0 \) in normal ABcast runs with D1_B on. We collected a random sample of several estimates of \( (x_{mx}, \eta) \) made during the experiments for n = 3. \( x_{mx} \) varied widely from 13 ms to 714 ms. Further, when \( x_{mx} \geq 200 \)ms, \( \eta \) was very small relative to \( x_{mx} \); for example, it was 14ms and 24ms when \( x_{mx} \) was 13ms and 714ms, respectively.

Recall that \( x_{mx} \) is estimated as the largest delay observed in the NTp sample of 1000 delays and \( \eta = 5.3 \times 10^{-4} \) when \( n = 3 \), where \( \bar{x} \) is the average of all delays in the sample (see § III-C2). Thus, a large \( x_{mx} \) accompanied by a small \( \eta \) is due to a very few delays in the sample being so exceptionally large (relative to \( \bar{x} \)) that we have \( x_{mx} \gg \eta \). Thus, with \( \eta = 24 \) and \( x_{mx} = 714 \)ms, the 714ms delay observed ought to be an unusually large outlier which we call a spike. On the other hand, \( \eta = 14 \)ms for \( x_{mx} = 13 \)ms seems to indicate an NTp sample with no spikes.

Since NTp = 10% of NTp, a single spike influences 10 consecutive estimates of \( x_{mx} \). Meanwhile, if another spike occurs, \( x_{mx} \) continues to be unusually large. Since \( \Delta_m > 4 \times x_{mx} \) when \( n = 2 \), \( \Delta_m \approx 1 \) second when, say, \( x_{mx} = 200 \)ms. So, all \( \text{ack}(m) \)'s are received well before clock = m.ts + \( \Delta_m \) and D1_B is always met before D1_A.

To have \( 0 < P(m | \text{suspect}) < 1 \), we ‘eliminate’ spikes by making ‘normal’ delays considerably longer: an infinite number of c nodes are simulated which always keep ARP non-empty and would decide on abort if r(T_x) cannot be submitted due to ARP being full. Thus, amcasts are sent at the rate permitted by FCC. Note that FCC does not enforce an upper bound on \( x_{mx} \); it slows down the send rate only until the recently-observed, excessive delays are accounted for in the next \( x_{mx} \) estimate and then amortizes the rate back to its default value of 1000 amcasts per second. Further, ABService completed 10^6 amcasts in each of 10 experiments for n = 3.

Table I presents the number of Aramis deliveries and failures (in brackets). Since delivery by D1_A or D1_B is local, different s-nodes deliver a different number of amcasts by Aramis. A total of 7392 messages could not meet D1_B, of which 2 did not meet G1 due to \( \Delta_m \) used being nor large enough for them. So, \( P(m | \text{suspect}) = \frac{7392}{10 \times 10^6} = 0.07\% \) and
Conservative parameter estimations and spikes in message delays help Aramis not to interfere with Base unnecessarily; when a crash occurs, Aramis proves itself as a reliable back-up for continued service delivery. Only under extreme load conditions, which may prevail only briefly in real-world deployment scenarios, failure probability was in the order of $10^{-4}$ and $10^{-7}$ for Aramis and ABService respectively.

Table[10] and Zookeeper[20] also provide atomic multicasting as an external service by using Paxos protocol which, in the absence of crashes, requires 3 communication steps whereas Base requires only 2 steps, as explained in § II. So, in identical environments, ABService should offer a faster crash-free performance.

### REFERENCES


