# AllConcur: Leaderless Concurrent Atomic Broadcast

Marius Poke
HLRS
University of Stuttgart
marius.poke@hlrs.de

Torsten Hoefler
Department of Computer Science
ETH Zurich
htor@inf.ethz.ch

Colin W. Glass
HLRS
University of Stuttgart
glass@hlrs.de

## ABSTRACT

Many distributed systems require coordination between the components involved. With the steady growth of such systems, the probability of failures increases, which necessitates scalable fault-tolerant agreement protocols. The most common practical agreement protocol, for such scenarios, is leader-based atomic broadcast. In this work, we propose ALLCONCUR, a distributed system that provides agreement through a leaderless concurrent atomic broadcast algorithm, thus, not suffering from the bottleneck of a central coordinator. In ALLCONCUR, all components exchange messages concurrently through a logical overlay network that employs early termination to minimize the agreement latency. Our implementation of ALLCONCUR supports standard sockets-based TCP as well as high-performance InfiniBand Verbs communications. ALLCONCUR can handle up to 135 million requests per second and achieves 17× higher throughput than today's standard leader-based protocols, such as Libpaxos. Thus, ALLCONCUR is highly competitive with regard to existing solutions and, due to its decentralized approach, enables hitherto unattainable system designs in a variety of fields.

## KEYWORDS

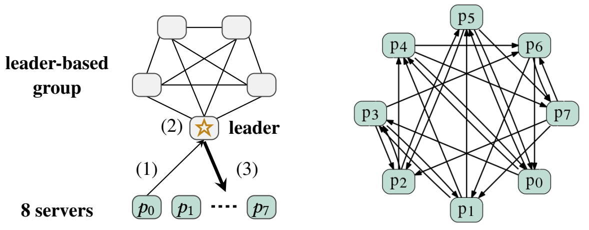
Distributed Agreement; Leaderless Atomic Broadcast; Reliability

**ACM Reference format:**
Marius Poke, Torsten Hoefler, and Colin W. Glass. 2017. AllConcur: Leaderless Concurrent Atomic Broadcast. In *Proceedings of HPDC '17, Washington , DC, USA, June 26-30, 2017,* 14 pages.
https://doi.org/http://dx.doi.org/10.1145/3078597.3078598

## 1 INTRODUCTION

Agreement is essential for many forms of collaboration in distributed systems. Although the nature of these systems may vary, ranging from distributed services provided by datacenters [16, 18, 59] to distributed operating systems, such as Barrelfish [56] and Mesosphere's DC/OS [46], they have in common that all the components involved regularly update a shared state. In many applications, the state updates cannot be reduced, e.g., the actions of players in multiplayer video games. Furthermore, the size of typical distributed systems has increased in recent years, making them more susceptible to single component failures [54].

Atomic broadcast is a communication primitive that provides fault-tolerant agreement while ensuring strong consistency of the overall system. It is often used to implement large-scale coordination services, such as replicated state machines [33] or travel reservation systems [59]. Yet, today's practical atomic broadcast algorithms rely on leader-based approaches, such as Paxos [37, 38], and thus, they may suffer from the bottleneck of a central coordinator, especially at large scale.



(a) Leader-based agreement (b) ALLCONCUR

**Figure 1: Agreement among 8 servers: (a) Using a leader-based group; three operations needed per update—(1) send; (2) replicate; and (3) disseminate. (b) Using a digraph $G$ with degree three and diameter two [58].**

In this paper, we present ALLCONCUR[1]—a distributed agreement system that relies on a leaderless atomic broadcast algorithm. In ALLCONCUR, all participants exchange messages concurrently through an overlay network, described by a digraph $G$ (§ 2.1). The maximum number of failures ALLCONCUR can sustain is given by $G$'s connectivity and can be adapted to system-specific requirements (§ 4.4). Moreover, ALLCONCUR employs a novel early termination mechanism (§ 2.3) that reduces the expected number of communication steps significantly (§ 4.2.2).

**Distributed agreement vs. replication.** Distributed agreement is conceptually different from state machine replication (SMR) [36, 55]: Agreement targets collaboration in distributed systems, while SMR aims to increase data reliability. Moreover, the number of agreeing components is an input parameter, while the number of replicas depends on the required data reliability.

**ALLCONCUR vs. leader-based agreement.** We consider the agreement among $n$ servers (see Figure 1 for $n = 8$). ALLCONCUR has the following properties: (1) subquadratic work, i.e., $O(nd)$, where $d$ is $G$'s degree (§ 4.1); 2) adjustable depth, given by $G$'s diameter and fault diameter (§ 4.2.2); (3) at most $2d$ connections per server; and (4) server-transitivity, i.e., all servers are treated equally, which entails fairness. In contrast, typical leader-based deployments do not have all of the above properties. Figure 1a shows an example of leader-based agreement. Each server sends updates to the group's leader, which, for reliability, replicates them within the group; the replicated updates are then disseminated to all servers. In typical leader-based approaches, such as client-server gaming platforms,

[1] Algorithm for LeaderLess CONCURrent atomic broadcast

servers interact directly with the leader (for both sending and receiving updates). Although such methods have minimal depth and can ensure fairness, they require quadratic work and the leader needs to maintain $n$ connections (§ 4.5).

**Data consistency.** AllConcur provides agreement while guaranteeing strong consistency. In particular, we focus on the strong consistency of state updates; thus, throughout the paper we use both request and update interchangeably. For strongly consistent reads, queries also need to be serialized via atomic broadcast. Serializing queries is costly, especially for read-heavy workloads. Typical coordination services [33] relax the consistency model: Queries are performed locally and, hence, can return stale data. AllConcur ensures that a server's view of the shared state cannot fall behind more than one round, i.e., one instance of concurrent atomic broadcast; thus, locally performed queries cannot be outdated by more than one round.

## 1.1 Applications and summary of results

AllConcur enables decentralized coordination services that require strong consistency at high request rates; thus, it allows for a novel approach to several real-world applications. We evaluate AllConcur using a set of benchmarks, representative of two such applications: (1) travel reservation systems; and (2) multiplayer video games.

**Travel reservation systems** are typical scenarios where updates are preceded by a large number of queries, e.g., clients check many flights before choosing a ticket. To avoid overloading a central server, existing systems either adopt weaker consistency models, such as eventual consistency [18], or partition the state [59], not allowing transactions spanning multiple partitions. AllConcur offers strong consistency by distributing queries over multiple servers that agree on the entire state. Each server's rate of introducing updates in the system is bounded by its rate of answering queries. Assuming 64-byte updates, AllConcur enables the agreement among 8 servers, each generating 100 million updates per second, in $35\mu s$; moreover, the agreement among 64 servers, each generating 32,000 updates per second, takes less than $0.75ms$.

**Multiplayer video games** are an example of applications where the shared state satisfies two conditions—it is too large to be frequently transferred through the network and it is periodically updated. For example, modern video games update the state once every $50ms$ (i.e., 20 frames per second) by only sending changes since the previous state [8, 9]. Thus, such applications are latency sensitive [7]. To decrease latency, existing systems either limit the number of players, e.g., $\approx 8$ players in real time strategy games, or limit the players' view to only a subset of the state, such as the area of interest in first person shooter games [8, 9]. AllConcur allows hundreds of servers to share a global state view at low latency; e.g., it supports the simultaneous interaction of 512 players, using typical update sizes of 40 bytes [8], with an agreement latency of $38ms$, thus, enabling so called epic battles [10], while providing strong consistency.

In addition, AllConcur can handle up to 135 million (8-byte) requests per second and achieves $17\times$ higher throughput than Libpaxos [57], an implementation of Paxos [37, 38], while its average overhead of providing fault-tolerance is 58% (§ 5).

| Notation | Description | Notation | Description |
|---|---|---|---|
| $G$ | the digraph | $d(G)$ | degree |
| $V(G)$ | vertices | $D(G)$ | diameter |
| $E(G)$ | directed edges | $\pi_{u,v}$ | path from $u$ to $v$ |
| $v^+(G)$ | successors of $v$ | $k(G)$ | vertex-connectivity |
| $v^-(G)$ | predecessors of $v$ | $D_f(G,f)$ | fault diameter |

**Table 1: Digraph notations.**

In summary, our work makes four key contributions:

- the design of AllConcur—a distributed system that provides agreement through a leaderless concurrent atomic broadcast algorithm (§ 3);
- a proof of AllConcur's correctness (§ 3.1);
- an analysis of AllConcur's performance (§ 4);
- implementations over standard sockets-based TCP and high-performance InfiniBand Verbs, that allows us to evaluate AllConcur's performance (§ 5).

## 2 THE BROADCAST PROBLEM

We consider $n$ servers connected through an overlay network, described by a digraph $G$. The servers communicate through messages, which cannot be lost (only delayed)—reliable communication. Each server may fail according to a *fail–stop* model: A server either operates correctly or it fails without further influencing other servers in the group. A server that did not fail is called *non-faulty*. We consider algorithms that tolerate up to $f$ failures, i.e., $f$-resilient.

In this paper, we use the notations from Chandra and Toueg [14] to describe both reliable and atomic broadcast: $m$ is a message (that is uniquely identified); *R-broadcast*$(m)$, *R-deliver*$(m)$, *A-broadcast*$(m)$, *A-deliver*$(m)$ are communication primitives for broadcasting and delivering messages reliably (*R-*) or atomically (*A-*); and *sender*$(m)$ is the server that R- or A-broadcasts $m$. Note that any message $m$ can be R- or A-broadcast at most once.

## 2.1 Reliable broadcast

Any (non-uniform) reliable broadcast algorithm must satisfy three properties [14, 29]:

- (Validity) If a non-faulty server R-broadcasts $m$, then it eventually R-delivers $m$.
- (Agreement) If a non-faulty server R-delivers $m$, then all non-faulty servers eventually R-deliver $m$.
- (Integrity) For any message $m$, every non-faulty server R-delivers $m$ at most once, and only if $m$ was previously R-broadcast by *sender*$(m)$.

A simple reliable broadcast algorithm uses a complete digraph for message dissemination [14]. When a server executes *R-broadcast*$(m)$, it sends $m$ to all other servers; when a server receives $m$ for the first time, it executes *R-deliver*$(m)$ only after sending $m$ to all other servers. Clearly, this algorithm solves the reliable broadcast problem. Yet, the all-to-all overlay network is unnecessary: For $f$-resilient reliable broadcast, it is sufficient to use a digraph with vertex-connectivity larger than $f$.

**Fault-tolerant digraphs.** We define a digraph $G$ by a set of $n$ vertices $V(G) = \{v_i : 0 \leq i \leq n-1\}$ and a set of directed edges $E(G) \subseteq \{(u,v) : u,v \in V(G) \text{ and } u \neq v\}$. In the context of reliable broadcast, the following parameters are of interest: the degree $d(G)$; the diameter $D(G)$; the vertex-connectivity $k(G)$; and the fault diameter $D_f(G, f)$. The *fault diameter* is the maximum diameter of $G$ after removing any $f < k(G)$ vertices [35]. Also, we refer to digraphs with $k(G) = d(G)$ as *optimally connected* [20, 47]. Finally, we use the following additional notations: $v^+(G)$ is the set of successors of $v \in V(G)$; $v^-(G)$ is the set of predecessors of $v \in V(G)$; and $\pi_{u,v}$ is a path between two vertices $u, v \in V(G)$. Table 1 summarizes all the digraph notations used throughout the paper.

## 2.2 Atomic broadcast

In addition to the reliable broadcast properties, atomic broadcast must also satisfy the following property [14, 29]:

- (Total order) If two non-faulty servers $p$ and $q$ A-deliver messages $m_1$ and $m_2$, then $p$ A-delivers $m_1$ before $m_2$, if and only if $q$ A-delivers $m_1$ before $m_2$.

There are different mechanisms to ensure total order [19]. A common approach is to use a distinguished server (leader) as a coordinator. Yet, this approach suffers from the bottleneck of a central coordinator (§ 4.5). An alternative entails broadcast algorithms that ensure atomicity through *destinations agreement* [19]: All non-faulty servers agree on a message set that is A-delivered. Destinations agreement reformulates the atomic broadcast problem as *consensus* problem [6, Chapter 5]; note that consensus and atomic broadcast are equivalent [14].

*2.2.1 Lower bound.* Consensus has a known synchronous lower bound: In a synchronous round-based model [6, Chapter 2], any $f$-resilient consensus algorithm requires, in the worst case, at least $f + 1$ rounds. Intuitively, a server may fail after sending a message to only one other server; this scenario may repeat up to $f$ times, resulting in only one server having the message; this server needs at least one additional round to disseminate the message. For more details, see the proof provided by Aguilera and Toueg [1]. Clearly, if $G$ is used for dissemination, consensus requires (in the worst case) $f + D_f(G, f)$ rounds. To avoid assuming always the worst case, we design an *early termination* scheme (§ 2.3).

*2.2.2 Failure detectors.* The synchronous model is unrealistic for real-world distributed systems; more fitting is to consider an asynchronous model. Yet, under the assumption of failures, consensus (or atomic broadcast) cannot be solved in an asynchronous model [25]: We cannot distinguish between failed and slow servers. To overcome this, we use a failure detector (FD). FDs are distributed oracles that provide information about faulty servers [14].

FDs have two main properties: *completeness* and *accuracy*. Completeness requires that all failures are eventually detected; accuracy requires that no server is suspected to have failed before actually failing. If both properties hold, then the FD is *perfect* (denoted by $\mathcal{P}$) [14]. In practice, completeness is easily guaranteed by a heartbeat mechanism: Each server periodically sends heartbeats to its successors; once it fails, its successors detect the lack of heartbeats.

Guaranteeing accuracy in asynchronous systems is impossible—message delays are unbounded. Yet, the message delays in practical distributed systems are bounded. Thus, accuracy can be probabilistically guaranteed (§ 3.2). Also, FDs can guarantee *eventual accuracy*—eventually, no server is suspected to have failed before actually failing. Such FDs are known as *eventually perfect* (denoted by $\diamond\mathcal{P}$) [14]. For now, we consider an FD that can be reliably treated as $\mathcal{P}$. Later, we discuss the implications of using $\diamond\mathcal{P}$, which can falsely suspect servers to have failed (§ 3.3.2).

## 2.3 Early termination

The synchronous lower bound holds also in practice: A message may be retransmitted by $f$ faulty servers, before a non-faulty server can disseminate it completely. Thus, in the worst case, any $f$-resilient consensus algorithm that uses $G$ for dissemination requires $f + D_f(G, f)$ communication steps. Yet, the only necessary and sufficient requirement for safe termination is for *every non-faulty server to A-deliver messages only once it has all the messages any other non-faulty server has*. Thus, early termination requires each server to track all the messages in the system.

Early termination has two parts: (1) deciding whether a message was A-broadcast; and (2) tracking the A-broadcast messages. In general, deciding whether a message was A-broadcast entails waiting for $f + D_f(G, f)$ communication steps (the worst case scenario must be assumed for safety). This essentially eliminates any form of early termination, if at least one server does not send a message. Yet, if every server A-broadcasts a message[2], it is *a priori* clear which messages exist; thus, no server waits for non-existent messages.

Every server tracks the A-broadcast messages through the received failure notifications. As an example, we consider a group of nine servers that communicate via a binomial graph [5]—a generalization of 1-way dissemination [30]. In binomial graphs, two servers $pi$ and $pj$ are connected if $j = i \pm 2^l \pmod n), \forall 0 \leq l \leq \lfloor \log_2 n \rfloor$ (see Figure 2a). We also consider a failure scenario in which $p_0$ fails after sending its message $m_0$ only to $p_1$; $p_1$ receives $m_0$, yet, it fails before it can send it further. How long should another server, e.g., $p_6$, wait for $m_0$?

Server $p_6$ is not directly connected to $p_0$, so it cannot directly detect its failure. Yet, $p_0$'s non-faulty successors eventually detect $p_0$'s failure. Once they suspect $p_0$ to have failed, they stop accepting messages from $p_0$; also, they R-broadcast notifications of $p_0$'s failure. For example, let $p_6$ receive such a notification from $p_2$; then, $p_6$ knows that, if $p_2$ did not already send $m_0$, then $p_2$ did not receive $m_0$ from $p_0$. Clearly, both *A-broadcast*() and *R-broadcast*() use the same paths for dissemination; the only difference between them is the condition to deliver a message. If $p_2$ had received $m_0$ from $p_0$, then it would have sent it to $p_6$ before sending the notification of $p_0$'s failure. Thus, using failure notifications, $p_6$ can track the dissemination of $m_0$. Once $p_6$ receives failure notifications from all of $p_0$'s and $p_1$'s non-faulty successors, it knows that no non-faulty server is in possession of $m_0$.

## 3 THE ALLCONCUR ALGORITHM

ALLCONCUR is a completely decentralized, $f$-resilient, round-based atomic broadcast algorithm that uses a digraph $G$ as an overlay network. In a nutshell, in every round $R$, every non-faulty server performs three tasks: (1) it A-broadcasts a single (possibly empty) message; (2) it tracks the messages A-broadcast in $R$ using the early termination mechanism described in Section 2.3; and (3) once

[2]The message can also be empty—the server A-broadcasts the information that it has nothing to broadcast.

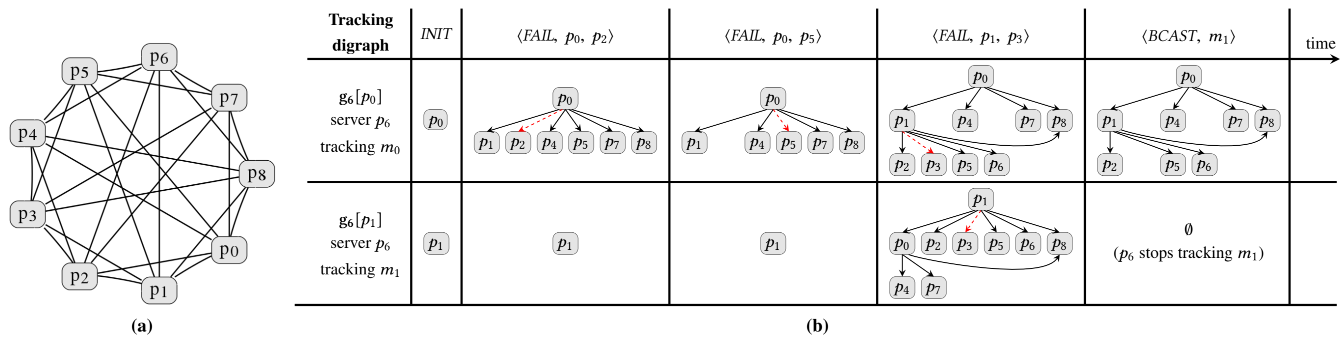


**Figure 2: (a) A binomial graph. (b) Message tracking within a binomial graph. Messages are shown chronologically from left to right. Dashed edges indicate failure notifications; for clarity, we omit the edges to the root of the digraphs.**

done with tracking, it A-delivers—in a deterministic order—all the messages A-broadcast in $R$ that it received. Note that A-delivering messages in a deterministic order entails that A-broadcast messages do not have to be received in the same order. When a server fails, its successors detect the failure and R-broadcast failure notifications to the other servers; these failure notifications enable the early termination mechanism. Algorithm 1 shows the details of ALLCONCUR during a single round. Later, we discuss the requirements of iterating ALLCONCUR.

Initially, we make the following two assumptions: (1) the maximum number of failures is bounded, i.e., $f < k(G)$; and (2) the failures are detected by $\mathcal{P}$. In this context, we prove correctness—we show that the four properties of (non-uniform) atomic broadcast are guaranteed (§ 3.1). Then, we provide a probabilistic analysis of accuracy: If the network delays can be approximated as part of a known distribution, then we can estimate the probability of the accuracy property to hold (§ 3.2). Finally, we discuss the consequences of dropping the two assumptions (§ 3.3).

ALLCONCUR is message-based. Each server $p_i$ receives messages from its predecessors and sends messages to its successors. We distinguish between two message types: (1) $\langle BCAST,\ m_j\rangle$, a message A-broadcast by $p_j$; and (2) $\langle FAIL,\ p_j,\ p_k \in p_j^+(G)\rangle$, a notification, R-broadcast by $p_k$, indicating $p_k$'s suspicion that its predecessor $p_j$ has failed. Note that if $p_i$ receives the notification and $p_k = p_i$, then it originated from $p_i$'s own FD. Algorithm 1 starts when at least one server A-broadcasts a message (line 1). Every server sends a message of its own, at the latest as a reaction upon receiving a message.

**Termination.** ALLCONCUR adopts a novel early termination mechanism (§ 2.3). To track the A-broadcast messages, each server $p_i$ stores an array $\mathbf{g_i}$ of $n$ digraphs, one for each server $p_* \in V(G)$; we refer to these as *tracking digraphs*. The vertices of each tracking digraph $\mathbf{g_i}[p_*]$ consist of the servers which (according to $p_i$) may have $m_*$. An edge $(p_j, p_k) \in E(\mathbf{g_i}[p_*])$ indicates $p_i$'s suspicion that $p_k$ received $m_*$ directly from $p_j$. If $p_i$ has $m_*$, then $\mathbf{g_i}[p_*]$ is no longer needed; hence, $p_i$ removes all its vertices, i.e., $V(\mathbf{g_i}[p_*]) = \emptyset$. Initially, $V(\mathbf{g_i}[p_j]) = \{p_j\}$, $\forall p_j \neq p_i$ and $V(\mathbf{g_i}[p_i]) = \emptyset$. Server $p_i$ A-delivers all known messages (in a deterministic order) once all tracking digraphs are empty (line 8).

Figure 2b illustrates the message-driven changes to the tracking digraphs based on the binomial graph example in Section 2.3. For clarity, we show only two of the messages being tracked by server $p_6$ (i.e., $m_0$ and $m_1$); both messages are tracked by updating $\mathbf{g_6}[p_0]$ and $\mathbf{g_6}[p_1]$, respectively. First, $p_6$ receives from $p_2$ a notification of $p_0$'s failure, which indicates that $p_2$ has not received $m_0$ directly from $p_0$ (§ 2.3). Yet, $p_0$ may have sent $m_0$ to its other successors; thus, $p_6$ adds them to $\mathbf{g_6}[p_0]$. Next, $p_6$ receives from $p_5$ a notification of $p_0$'s failure—$p_5$ has not received $m_0$ directly from $p_0$ either and, thus, the edge $(p_0, p_5)$ is removed. Then, $p_6$ receives from $p_3$ a notification of $p_1$'s failure. Hence, $p_6$ extends both $\mathbf{g_6}[p_0]$ and $\mathbf{g_6}[p_1]$ with $p_1$'s successors (except $p_3$). In addition, due to the previous notifications of $p_0$'s failure, $p_6$ extends $\mathbf{g_6}[p_1]$ with $p_0$'s successors (except $p_2$ and $p_5$). Finally, $p_6$ receives $m_1$; thus, it removes all the vertices from $\mathbf{g_6}[p_1]$ (i.e., it stops tracking $m_1$).

**Receiving $\langle BCAST,\ m_j\rangle$.** When receiving an A-broadcast message $m_j$ (line 14), server $p_i$ adds it to the set $M_i$ of known messages. Also, it A-broadcasts its own message $m_i$, in case it did not do so before. Then, it continues the dissemination of each known message through the network—$p_i$ sends all unique messages it has not already sent to its successors $p_i^+(G)$. Finally, $p_i$ removes all the vertices from the $\mathbf{g_i}[p_j]$ digraph; then, it checks whether the termination conditions are fulfilled.

**Receiving $\langle FAIL,\ p_j,\ p_k\rangle$.** When receiving a notification, R-broadcast by $p_k$, indicating $p_k$'s suspicion that $p_j$ has failed (line 21), $p_i$ disseminates it further. Then, it adds a tuple $(p_j, p_k)$ to the set $F_i$ of received failure notifications. Finally, it updates the tracking digraphs in $\mathbf{g_i}$ that contain $p_j$ as a vertex.

We distinguish between two cases, depending on whether this is the first notification of $p_j$'s failure received by $p_i$. If it is the first, $p_i$ updates all $\mathbf{g_i}[p_*]$ containing $p_j$ as a vertex by adding $p_j$'s successors (from $G$) together with the corresponding edges. The rationale is, that $p_j$ may have sent $m_*$ to his successors, who are now in possession of it. Thus, we track the possible whereabouts of messages. However, there are some exceptions: Server $p_k$ could not have received $m_*$ directly from $p_j$ (§ 2.3). Also, if a successor $p_f \notin V(\mathbf{g_i}[p_*])$ is added, which is already known to have failed, it may have already received $m_*$ and sent it further. Hence, the successors of $p_f$ could be in possession of $m_*$ and are added to $\mathbf{g_i}[p_*]$ in the same way as described above (line 32).

**Algorithm 1:** The AllConcur algorithm; code executed by server $p_i$; see Table 1 for digraph notations.

```
Input: $n; f; G; m_i; M_i \leftarrow \emptyset; F_i \leftarrow \emptyset; V(g_i[p_i]) \leftarrow \emptyset; V(g_i[p_j]) \leftarrow \{p_j\}, \forall j \neq i$
1  def A-broadcast($m_i$):
2      send $\langle BCAST, m_i \rangle$ to $p_i^+(G)$
3      $M_i \leftarrow M_i \cup \{m_i\}$
4      check_termination()
5  def check_termination():
6      if $V(g_i[p]) = \emptyset, \forall p$ then
7          foreach $m \in$ sort($M_i$) do
8              A-deliver($m$)                                   // A-deliver messages
           /* preparing for next round                                 */
9          foreach server $p_*$ do
10             if $m_* \notin M_i$ then
11                 $V(G) \leftarrow V(G) \setminus \{p_*\}$             // remove servers
12         foreach $(p, p_s) \in F_i$ s.t. $p \in V(G)$ do
13             send $\langle FAIL, p, p_s \rangle$ to $p_i^+(G)$           // resend failures
14 receive $\langle BCAST, m_j \rangle$:
15     if $m_i \notin M_i$ then A-broadcast($m_i$)
16     $M_i \leftarrow M_i \cup \{m_j\}$
17     for $m \in M_i$ not already sent do
18         send $\langle BCAST, m \rangle$ to $p_i^+(G)$                 // disseminate messages
19     $V(g_i[p_j]) \leftarrow \emptyset$
20     check_termination()
21 receive $\langle FAIL, p_j, p_k \in p_j^+(G) \rangle$:
       /* if $k = i$ then notification from local FD                   */
22     send $\langle FAIL, p_j, p_k \rangle$ to $p_i^+(G)$               // disseminate failures
23     $F_i \leftarrow F_i \cup \{(p_j, p_k)\}$
24     foreach server $p_*$ do
25         if $p_j \notin V(g_i[p_*])$ then continue
26         if $p_j^+(g_i[p_*]) = \emptyset$ then
               /* maybe $p_j$ sent $m_*$ to someone in $p_j^+(G)$ before
                  failing                                              */
27             $Q \leftarrow \{(p_j, p) : p \in p_j^+(G) \setminus \{p_k\}\}$          // FIFO queue
28             foreach $(p_p, p) \in Q$ do
29                 $Q \leftarrow Q \setminus \{(p_p, p)\}$
30                 if $p \notin V(g_i[p_*])$ then
31                     $V(g_i[p_*]) \leftarrow V(g_i[p_*]) \cup \{p\}$
32                     if $\exists(p, *) \in F_i$ then
33                         $Q \leftarrow Q \cup \{(p, p_s) : p_s \in p^+(G)\} \setminus F_i$
34                 $E(g_i[p_*]) \leftarrow E(g_i[p_*]) \cup \{(p_p, p)\}$
35         else if $p_k \in p_j^+(g_i[p_*])$ then
               /* $p_k$ has not received $m_*$ from $p_j$                    */
36             $E(g_i[p_*]) \leftarrow E(g_i[p_*]) \setminus \{(p_j, p_k)\}$
37             foreach $p \in V(g_i[p_*])$ s.t. $\nexists \pi_{p_*,p}$ in $g_i[p_*]$ do
38                 $V(g_i[p_*]) \leftarrow V(g_i[p_*]) \setminus \{p\}$          // no input
39         if $\forall p \in V(g_i[p_*]), (p, *) \in F_i$ then
40             $V(g_i[p_*]) \leftarrow \emptyset$                      // no dissemination
41     check_termination()
```

If $p_i$ is already aware of $p_j$'s failure (i.e., the above process already took place), the new failure notification informs $p_i$, that $p_k$ (the origin of the notification) has not received $m_*$ from $p_j$—because $p_k$ would have sent it before sending the failure notification. Thus, the edge $(p_j, p_k)$ can be removed from $g_i[p_*]$ (line 35).

In the end, $p_i$ prunes $g_i[p_*]$ by removing the servers no longer of interest in tracking $m_*$. First, $p_i$ removes every server $p$ for which there is no path (in $g_i[p_*]$) from $p_*$ to $p$, as $p$ could not have received $m_*$ from any of the servers in $g_i[p_*]$ (line 37). Then, if $g_i[p_*]$ contains only servers already known to have failed, $p_i$ prunes it entirely—no non-faulty server has $m_*$ (line 39).

**Iterating AllConcur.** Executing subsequent rounds of AllConcur requires the correct handling of failures. Since different servers may end and begin rounds at different times, AllConcur employs a consistent mechanism of tagging servers as failed: At the end of each round, all servers whose messages were not A-delivered are tagged as failed by all the other servers (line 9). As every non-faulty server agrees on the A-delivered messages, this ensures a consistent view of failed servers. In the next round, every server resends the failure notifications, except those of servers already tagged as failed (line 12). Thus, only the tags and the necessary resends need to be carried over from the previous round. Moreover, each message contains the sequence number $R$ of the round in which it was first sent. Thus, all messages can be uniquely identified, i.e., $\langle BCAST, m_j \rangle$ by $(R, p_j)$ tuples and $\langle FAIL, p_j, p_k \rangle$ by $(R, p_j, p_k)$ tuples, which allows for multiple rounds to coexist.

**Initial bootstrap and dynamic membership.** To bootstrap AllConcur, we require a centralized service, such as ZooKeeper [33]: The system must decide on the initial configuration—the identity of the $n$ servers, the fault tolerance $f$ and the digraph $G$. Once AllConcur starts, any further reconfigurations are agreed upon via atomic broadcast. This includes topology reconfigurations and membership changes, i.e., servers leaving and joining the system. In contrast to leader-based approaches, where such changes may necessitate a leader election, in AllConcur, dynamic membership is handled directly by the algorithm.

## 3.1 Correctness

To prove AllConcur's correctness, we show that the four properties of (non-uniform) atomic broadcast are guaranteed (§ 2.2). Clearly, the integrity property holds: Every server $p_i$ executes *A-deliver*() only once for each message in the set $M_i$, which contains only messages A-broadcast by some servers. To show that the validity property holds, it is sufficient to prove that the algorithm terminates (see Lemma 3.5). To show that both the agreement and the total order properties hold, it is sufficient to prove *set agreement*—when the algorithm terminates, all non-faulty servers have the same set of known messages (see Lemma 3.6). To prove termination and set agreement, we introduce the following lemmas:

Lemma 3.1. *Let $p_i$ be a non-faulty server; let $p_j \neq p_i$ be a server; let $\pi_{p_j,p_i} = (a_1, \ldots, a_d)$ be a path (in digraph G) from $p_j$ to $p_i$. If $p_j$ knows a message m (either its own or received), then, $p_i$ eventually receives either $\langle BCAST, m \rangle$ or $\langle FAIL, a_k, a_{k+1} \rangle$ with $1 \leq k < d$.*

Proof. Server $p_j$ can either fail or send $m$ to $a_2$. Further, for each inner server $a_k \in \pi_{p_j,p_i}, 1 < k < d$, we distinguish three scenarios: (1) $a_k$ fails; (2) $a_k$ detects the failure of its predecessor on the path; or (3) $a_k$ further sends the message received from its predecessor on the path. The message can be either $\langle BCAST, m \rangle$ or $\langle FAIL, a_l, a_{l+1} \rangle$ with $1 \leq l < k$. Thus, $p_i$ eventually receives either $\langle BCAST, m \rangle$ or $\langle FAIL, a_k, a_{k+1} \rangle$ with $1 \leq k < d$. Figure 3 shows, in a tree-like fashion, what messages can be transmitted along a three-server path. □

Lemma 3.2. *Let $p_i$ be a non-faulty server; let $p_j \neq p_i$ be a server. If $p_j$ knows a message m (either its own or received), then $p_i$ eventually receives either the message m or a notification of $p_j$'s failure.*

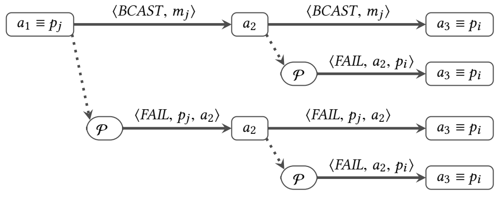

$a_1 \equiv p_j$
$\langle BCAST,\ m_j \rangle$
$a_2$
$\langle BCAST,\ m_j \rangle$
$a_3 \equiv p_i$
$\mathcal{P}$
$\langle FAIL,\ a_2,\ p_i \rangle$
$a_3 \equiv p_i$
$\mathcal{P}$
$\langle FAIL,\ p_j,\ a_2 \rangle$
$a_2$
$\langle FAIL,\ p_j,\ a_2 \rangle$
$a_3 \equiv p_i$
$\mathcal{P}$
$\langle FAIL,\ a_2,\ p_i \rangle$
$a_3 \equiv p_i$

**Figure 3: Possible messages along a three-server path. Dotted arrows indicate failure detection.**

PROOF. If $p_i$ receives $m$, then the proof is done. In the case $p_i$ does not receive $m$, we assume it does not receive a notification of $p_j$'s failure either. Due to $G$'s vertex-connectivity, there are at least $k(G)$ vertex-disjoint paths $\pi_{p_j,p_i}$. For each of these paths, $p_i$ must receive notifications of some inner vertex failures (cf. Lemma 3.1). Since the paths are vertex-disjoint, each notification indicates a different server failure. However, this contradicts the assumption that $f < k(G)$. □

COROLLARY 3.3. *Let $p_i$ be a non-faulty server; let $p_j \neq p_i$ be a server. If $p_j$ receives a message, then $p_i$ eventually receives either the same message or a notification of $p_j$'s failure.*

LEMMA 3.4. *Let $p_i$ be a server; let $\mathrm{g_i}[p_j]$ be a tracking digraph that can no longer be pruned. If $E(\mathrm{g_i}[p_j]) \neq \emptyset$, then $p_i$ eventually removes an edge from $E(\mathrm{g_i}[p_j])$.*

PROOF. We assume that $p_i$ removes no edge from $E(\mathrm{g_i}[p_j])$. Clearly, the following statements are true: (1) $V(\mathrm{g_i}[p_j]) \neq \emptyset$ (since $E(\mathrm{g_i}[p_j]) \neq \emptyset$); (2) $p_j \in V(\mathrm{g_i}[p_j])$ (since $\mathrm{g_i}[p_j]$ can no longer be pruned); and (3) $p_j$ is known to have failed (since $V(\mathrm{g_i}[p_j]) \neq \{p_j\}$). Let $p \in V(\mathrm{g_i}[p_j])$ be a server such that $p_i$ receives no notification of $p$'s failure. The reason $p$ exists is twofold: (1) the maximum number of failures is bounded; and (2) $\mathrm{g_i}[p_j]$ can no longer be pruned (line 39). Then, we can construct a path $\pi_{p_j,p} = (a_1, \dots, a_d)$ in $\mathrm{g_i}[p_j]$ such that every server along the path, except for $p$, is known to have failed (line 37). Eventually, $p$ receives either $\langle BCAST,\ m_j \rangle$ or $\langle FAIL,\ a_k,\ a_{k+1} \rangle$ with $1 \leq k < d$ (cf. Lemma 3.1). Since $p_i$ receives no notification of $p$'s failure, the message received by $p$ eventually arrives at $p_i$ (cf. Corollary 3.3). On the one hand, if $p_i$ receives $\langle BCAST,\ m_j \rangle$, then all edges are removed from $E(\mathrm{g_i}[p_j])$; this leads to a contradiction. On the other hand, if $p_i$ receives $\langle FAIL,\ a_k,\ a_{k+1} \rangle$, then the edge $(a_k, a_{k+1})$ is removed from $E(\mathrm{g_i}[p_j])$ (line 36); this also leads to a contradiction. □

LEMMA 3.5. *(Termination) Let $p_i$ be a non-faulty server. Then, $p_i$ eventually terminates.*

PROOF. If $V(\mathrm{g_i}[p]) = \emptyset$, $\forall p$, then the proof is done (line 6). We assume $\exists p_j$ such that $V(\mathrm{g_i}[p_j]) \neq \emptyset$ and $\mathrm{g_i}[p_j]$ can no longer be pruned. Clearly, $p_j \in V(\mathrm{g_i}[p_j])$. Server $p_i$ receives either $m_j$ or a notification of $p_j$'s failure (cf. Lemma 3.2). If $p_i$ receives $m_j$, then all servers are removed from $V(\mathrm{g_i}[p_j])$, which contradicts $V(\mathrm{g_i}[p_j]) \neq \emptyset$. We assume $p_i$ receives a notification of $p_j$'s failure; then, $p_j^+(\mathrm{g_i}[p_j]) \neq \emptyset$ (since $\mathrm{g_i}[p_j]$ can no longer be pruned); also, $E(\mathrm{g_i}[p_j]) \neq \emptyset$. By repeatedly applying the result of Lemma 3.4, it results that $p_i$ eventually removes all edges from $\mathrm{g_i}[p_j]$. As a result, $\mathrm{g_i}[p_j]$ is eventually completely pruned, which contradicts $V(\mathrm{g_i}[p_j]) \neq \emptyset$. □

LEMMA 3.6. *(Set agreement) Let $p_i$ and $p_j$ be any two non-faulty servers. Then, after* AllConcur*'s termination, $M_i = M_j$.*

PROOF. It is sufficient to show that if $m_* \in M_i$ when $p_i$ terminates, then also $m_* \in M_j$ when $p_j$ terminates. We assume that $p_j$ does not receive $m_*$. Let $\pi_{p_*,p_i} = (a_1, \dots, a_d)$ be one of the paths (in $G$) on which $m_*$ arrives at $p_i$. Let $k$, $1 \leq k \leq d$ the smallest index such that $p_j$ receives no notification of $a_k$'s failure. The existence of $a_k$ is given by the existence of $p_i$, a server that is both non-faulty and on $\pi_{p_*,p_i}$. Clearly, $a_k \in V(\mathrm{g_j}[p_*])$. Since it terminates, $p_j$ eventually removes $a_k$ from $\mathrm{g_j}[p_*]$. In general, $p_j$ can remove $a_k$ when it receives either $m_*$ or a notification of $a_k$'s failure; yet, both alternatives lead to contradictions. In addition, for $k > 1$, $p_j$ can remove $a_k$ when there is no path $\pi_{p_*,a_k}$ in $\mathrm{g_j}[p_*]$. This requires $p_j$ to remove an edge on the $(a_1, \dots, a_k)$ path. Thus, $p_j$ receives a message $\langle FAIL,\ a_l,\ a_{l+1} \rangle$ with $1 \leq l < k$. Yet, since $a_{l+1}$ received $m_*$ from $a_l$, $p_j$ must receive $\langle BCAST,\ m_* \rangle$ first, which leads to a contradiction. □

COROLLARY 3.7. AllConcur *solves the atomic broadcast problem while tolerating up to $f$ failures.*

## 3.2 Probabilistic analysis of accuracy

Algorithm 1 assumes a perfect FD, which requires the accuracy property to hold. Accuracy is difficult to guarantee in practice: Due to network delays, a server may falsely suspect another server to have failed. Yet, when the network delays can be approximated as part of a known distribution, accuracy can be probabilistically guaranteed. Let $T$ be a random variable that describes the network delays. Then, we denote by $Pr[T > t]$ the probability that a message delay exceeds a constant $t$.

We propose an FD based on a heartbeat mechanism. Every non-faulty server sends heartbeats to its successors in $G$; the heartbeats are sent periodically, with a period $\Delta_{hb}$. Every non-faulty server $p_i$ waits for heartbeats from its predecessors in $G$; if, within a period $\Delta_{to}$, $p_i$ receives no heartbeats from a predecessor $p_j$, it suspects $p_j$ to have failed. Since we assume heartbeat messages are delayed according to a known distribution, we can estimate the probability of the FD to be accurate, in particular a lower bound of the probability of the proposed FD to behave indistinguishably from a perfect one.

The interval in which $p_i$ receives two heartbeats from a predecessor $p_j$ is bounded by $\Delta_{hb} + T$. In the interval $\Delta_{to}$, $p_j$ sends $\lfloor \Delta_{to}/\Delta_{hb} \rfloor$ heartbeats to $p_i$. The probability that $p_i$ does not receive the $k$'th heartbeat within the period $\Delta_{to}$ is bounded by $Pr[T > \Delta_{to} - k\Delta_{hb}]$. For $p_i$ to incorrectly suspect $p_j$ to have failed, it has to receive none of the $k$ heartbeats. Moreover, $p_i$ can incorrectly suspect $d(G)$ predecessors; also, there are $n$ servers that can incorrectly suspect their predecessors. Thus, the probability of the accuracy property to hold is at least $(1 - \prod_{k=1}^{\lfloor \Delta_{to}/\Delta_{hb} \rfloor} Pr[T > \Delta_{to} - k\Delta_{hb}])^{n \cdot d(G)}$.

Increasing both the timeout period and the heartbeat frequency increases the likelihood of accurate failure detection. The probability of no incorrect failure detection in the system, together with the probability of less than $k(G)$ failures define the reliability of ALLCONCUR.

## 3.3 Widening the scope

A practical atomic broadcast algorithm must always guarantee safety. Under the two initial assumptions, i.e., $f < k(G)$ and $\mathcal{P}$, AllConcur guarantees both safety and liveness (§ 3.1). In this section, we show that $f < k(G)$ is not required for safety, but only for liveness (§ 3.3.1). Also, we provide a mechanism that enables AllConcur to guarantee safety even when the $\mathcal{P}$ assumption is dropped (§ 3.3.2).

*3.3.1 Disconnected digraph.* In general, Algorithm 1 requires $G$ to be connected. A digraph can be disconnected by either (1) removing a sufficient number of vertices to break the vertex-connectivity, i.e., $f \geq k(G)$, or (2) removing sufficent edges to break the edge-connectivity. Under the assumption of reliable communication (i.e., $G$'s edges cannot be removed), only the fist scenario is possible. If $f \geq k(G)$, termination is not guaranteed (see Lemma 3.2). Yet, some servers may still terminate the round even if $G$ is disconnected. In this case, set agreement still holds, as the proof of Lemma 3.6 does not assume less than $k(G)$ failures. In summary, the $f < k(G)$ assumption is needed only to guarantee liveness; safety is guaranteed regardless of the number of failures (similar to Paxos [37, 38]).

In scenarios where $G$'s edges can be removed, such as network partitioning, a non-faulty server disconnected from one of its non-faulty successors will be falsely suspected to have failed[3]. Thus, the assumption of $\mathcal{P}$ does not hold and we need to relax it to $\diamond\mathcal{P}$.

*3.3.2 Eventual accuracy.* For some distributed systems, it may be necessary to use $\diamond\mathcal{P}$ instead of $\mathcal{P}$. For instance, in cases of network partitioning as discussed above, or for systems in which approximating network delays as part of a known distribution is difficult. Implementing a heartbeat-based $\diamond\mathcal{P}$ is straightforward [14]: When a server falsely suspects another server to have failed, it increments the timeout period $\Delta_{to}$; thus, eventually, non-faulty servers are no longer suspected to have failed. Yet, when using $\diamond\mathcal{P}$, failure notifications no longer necessarily indicate server failures. Thus, to adapt Algorithm 1 to $\diamond\mathcal{P}$, we need to ensure the correctness of early termination, which relies on the information carried by failure notifications.

First, a $\langle FAIL, p_j, p_k\rangle$ message received by $p_i$, indicates that $p_k$ did not receive (and it will not receive until termination) from $p_j$ any message not yet received by $p_i$. Thus, once a server suspects one of its predecessors to have failed, it must ignore any subsequent messages (except failure notifications) received from that predecessor (until the algorithm terminates). As a result, when using $\diamond\mathcal{P}$, it is still possible to decide if a server received a certain message.

Second, $p_i$ receiving notifications of $p_j$'s failure from all $p_j$'s successors indicates both that $p_j$ is faulty and that it did not disseminate further any message not yet received by $p_i$. Yet, when using $\diamond\mathcal{P}$, these notifications no longer indicate that $p_j$ is faulty. Thus, both $p_i$ and $p_j$ can terminate without agreeing on the same set (i.e., $M_i \neq M_j$), which breaks AllConcur's safety. In this case though, $p_i$ and $p_j$ are part of different strongly connected components. For set agreement to hold (§ 3.1), only the servers from one single strongly connected component can A-deliver messages; we refer to this component as the *surviving partition*. The other servers are considered to be faulty (for the properties of reliable broadcast to hold). To ensure the uniqueness of the surviving partition, it must contain at least a majority of the servers.

**Deciding whether to A-deliver.** Each server decides whether it is part of the surviving partition via a mechanism based on Kosaraju's algorithm to find strongly connected components [2, Chapter 6]. In particular, once each server $p_i$ decides on the set $M_i$, it R-broadcasts two messages: (1) a forward message $\langle FWD, p_i\rangle$; and (2) a backward message $\langle BWD, p_i\rangle$. The backward message is R-broadcast using the transpose of $G$. Then, $p_i$ A-delivers the messages from $M_i$ only when it receives both forward and backward messages from at least $\lfloor n/2 \rfloor$ servers. Intuitively, a $\langle FWD, p_j\rangle$ message received by $p_i$ indicates that when $p_j$ decided on its set $M_j$, there was at least one path from $p_j$ to $p_i$; thus, $p_i$ knows of all the messages known by $p_j$ (i.e., $M_j \subseteq M_i$). Similarly, a $\langle BWD, p_j\rangle$ message indicates that $M_i \subseteq M_j$. Thus, when $p_i$ A-delivers it knows that at least a majority of the servers (including itself) A-deliver the same messages.

**Non-terminating servers.** To satisfy the properties of reliable broadcast (§ 2.1), non-terminating servers need to be eventually removed from the system and consequently, be considered as faulty. In practice, these servers could restart after a certain period of inactivity and then try to rejoin the system, by sending a membership request to one of the non-faulty servers.

## 4 PERFORMANCE ANALYSIS

AllConcur is designed as a high-throughput atomic broadcast algorithm. Its performance is given by three metrics: (1) work per server; (2) communication time; and (3) storage requirements. Our analysis focuses on Algorithm 1, i.e., connected digraph and perfect FD, and it uses the LogP model [17]. The LogP model is described by four parameters: the latency $L$; the overhead $o$; the gap between messages $g$; and the number of processes (or servers) $P$, which we denote by $n$. We make the common assumption that $o > g$ [4]; also, the model assumes short messages. AllConcur's performance depends on $G$'s parameters: $d$, $D$, and $D_f$. A discussion on how to choose $G$ is provided in Section 4.4.

### 4.1 Work per server

The amount of work a server performs is given by the number of messages it receives and sends. AllConcur distinguishes between A-broadcast messages and failure notifications. First, without failures, every server receives an A-broadcast message from all of its $d$ predecessors, i.e., $(n-1)\cdot d$ messages received by each server. This is consistent with the $\Omega(n^2 f)$ worst-case message complexity for synchronous $f$-resilient consensus algorithms [21]. Second, every failed server is detected by up to $d$ servers, each sending a failure notification to its $d$ successors. Thus, each server receives up to $d^2$ notifications of each failure. Overall, each server receives at most $n\cdot d + f\cdot d^2$ messages. Since $G$ is regular, each server sends the same number of messages.

In order to terminate, in a non-failure scenario, a server needs to receive at least $(n-1)$ messages and send them further to $d$ successors. We estimate the time of sending or receiving a message by the overhead $o$ of the LogP model [17]. Thus, a lower bound on termination (due to work) is given by $2(n-1)do$.

[3]Note that if $G$ is disconnected by removing vertices, a non-faulty server cannot be disconnected from its non-faulty successors.

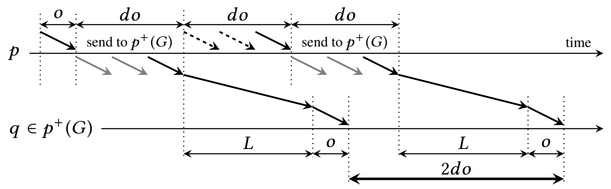


**Figure 4: LogP model of message transmission in AllConcur for $d = 3$. Dashed arrows indicate already sent messages.**

## 4.2 Communication time

In general, the time to transmit a message (between two servers) is estimated by $T(msg) = L + 2o$. We consider only the scenario of a single non-empty message $m$ being A-broadcast and we estimate the time between $sender(m)$ A-broadcasts $m$ and A-delivers $m$.

*4.2.1 Non-faulty scenario.* We split the A-broadcast of $m$ in two: (1) *R-broadcast*$(m)$; and (2) the empty messages $m_\emptyset$ travel back to $sender(m)$. In a non-failure scenario, messages are R-broadcast in $D$ steps, i.e., $T_D(msg) = T(msg)D$. Moreover, to account for contention while sending to $d$ successors, we add to the sending overhead the expected waiting time, i.e., $o_s = o + \frac{d-1}{2}o$. Note that for *R-broadcast*$(m)$, there is no contention while receiving (every server, except $sender(m)$, is idle until it receives $m$). Thus, the time to R-broadcast $m$ is estimated by $T_D(m) = (L + o_s + o)D$.

When the empty messages $m_\emptyset$ are transmitted to $sender(m)$, the servers are no longer idle; $T_D(m_\emptyset)$ needs to account for contention while receiving. On average, servers send further one in every $d$ received messages; thus, a server $p$ sends messages to the same successor $q$ at a period of $2do$ (see Figure 4). In general, once a message arrives at a server, it needs to contend with other received messages. Yet, servers handle incoming connections in a round-robin fashion; processing a round of messages from all $d$ predecessors takes (on average) $2do$, i.e., $2o$ per predecessor (see server $p$ in Figure 4). Thus, on average, the message in-rate on a connection matches the out-rate: There is no contention while receiving empty messages, i.e., $T_D(m_\emptyset) = T_D(m)$.

*4.2.2 Faulty scenario—probabilistic analysis.* Let $\pi_m$ be the longest path a message $m$ has to travel before it is completely disseminated. If $m$ is lost (due to failures), $\pi_m$ is augmented by the longest path the failure notifications have to travel before reaching all non-faulty servers. Let $\mathcal{D}$ be a random variable that denotes the length of the longest path $\pi_m$, for any A-broadcast message $m$, i.e., $\mathcal{D} = \max_m |\pi_m|, \forall m$; we refer to $\mathcal{D}$ as AllConcur's *depth*. Intuitively, the depth is the asynchronous equivalent of the number of rounds from synchronous systems. Thus, $\mathcal{D}$ ranges from $D$, if no servers fail, to $f + D_f$ in the worst case scenario (§ 2.2.1). Yet, $\mathcal{D}$ is not uniformly distributed. A back-of-the-envelope calculation shows that it is very unlikely for AllConcur's depth to exceed $D_f$.

We consider a single AllConcur round, with all $n$ servers initially non-faulty. Also, we estimate the probability $p_f$ of a server to fail, by using an exponential lifetime distribution model, i.e., over a period of time $\Delta$, $p_f = 1 - e^{-\Delta/MTTF}$, where $MTTF$ is the mean time to failure. If $sender(m)$ succeeds in sending $m$ to all of its successors, then $D \leq \pi_m \leq D_f$ (§ 2.2.1). Thus, $Pr[D \leq \mathcal{D} \leq D_f] = e^{-n \cdot d \cdot o/MTTF}$, where $o$ is the sending overhead [17]. Note that this probability increases if the round starts with previously failed servers.

For typical values of $MTTF$ ($\approx$ 2 years [54]) and $o$ ($\approx 1.8\mu s$ for TCP on our InfiniBand cluster § 5), a system of 256 servers connected via a digraph of degree 7 (see Table 3) would finish 1 million AllConcur rounds with $\mathcal{D} \leq D_f$ with a probability larger than 99.99%. This demonstrates why early termination is essential for efficiency, as for most rounds no failures occur and even if they do occur, the probability of $\mathcal{D} > D_f$ is very small. Note that a practical deployment of AllConcur should include regularly replacing failed servers and/or updating $G$ after failures.

*4.2.3 Estimating the fault diameter.* The fault diameter of any digraph $G$ is trivially bounded by $\left\lfloor \frac{n-f-2}{k(G)-f} \right\rfloor + 1$ [15, Theorem 6]. However, this bound is neither tight nor does it relate the fault diameter to the digraph's diameter. In general, the fault diameter is unbounded in terms of the digraph diameter [15]. Yet, if the first $f+1$ shortest vertex-disjoint paths from $u$ to $v$ are of length at most $\delta_f$ for $\forall u, v \in V(G)$, then $D_f(G, f) \leq \delta_f$ [35]. To compute $\delta_f$, we need to solve the min-max $(f + 1)$-disjoint paths problem for every pair of vertices: Find $(f + 1)$ vertex-disjoint paths $\pi_0, \dots, \pi_f$ that minimize the length of the longest path; hence, $\delta_f = \max_i |\pi_i|$, $0 \leq i \leq f$.

Unfortunately, the problem is known to be strongly NP-complete [41]. As a heuristic to find $\delta_f$, we minimize the sum of the lengths instead of the maximum length, i.e., the min-sum disjoint paths problem. This problem can be expressed as a minimum-cost flow problem; thus, it can be solved polynomially with well known algorithms, e.g., successive shortest path [3, Chapter 9]. Let $\hat{\pi}_0, \dots, \hat{\pi}_f$ be the paths obtained from solving the min-sum disjoint paths problem; also, let $\hat{\delta_f} = \max_i |\hat{\pi}_i|$, $0 \leq i \leq f$. Then, from the minimality condition of both min-max and min-sum problems, we deduce the following chain of inequalities:

$$\frac{\sum_{i=0}^{f} |\hat{\pi}_i|}{f+1} \leq \frac{\sum_{i=0}^{f} |\pi_i|}{f+1} \leq \delta_f \leq \hat{\delta_f}. \qquad (1)$$

Thus, we approximate the fault diameter bound by $\hat{\delta_f}$. Then, we use Equation (1) to check the accuracy of our approximation: We check the difference between the maximum and the average length of the paths obtained from solving the tractable min-sum problem.

As an example, we consider the binomial graph example from [5], i.e., $n = 12$ and $p_i^+ = p_i^- = \left\{p_j : j = i \pm \{1, 2, 4\}\right\}$. The graph has connectivity $k = 6$ and diameter $D = 2$. After solving the min-sum problem, we can estimate the fault diameter bound, i.e., $3 \leq \delta_f \leq 4$. After a closer look, we can see that one of the six vertex-disjoint paths from $p_0$ to $p_3$ has length four, i.e., $p_0 - p_{10} - p_6 - p_5 - p_3$.

## 4.3 Storage requirements

Each server $p_i$ stores five data structures (see Algorithm 1): (1) the digraph $G$; (2) the set of known messages $M_i$; (3) the set of received failure notifications $F_i$; (4) the array of tracking digraphs $\mathbf{g_i}$; and (5) the internal FIFO queue $Q$. Table 2 shows the space complexity of each data structure. In general, for regular digraphs, $p_i$ needs to store $d$ edges per node; yet, some digraphs require less storage, e.g., binomial graphs [5] require only the graph size. Also, each tracking digraph has at most $fd$ vertices; yet, only $f$ of these digraphs may

| Notation | Description | Space complexity per server |
|---|---|---|
| $G$ | digraph | $O(n \cdot d)$ |
| $M_i$ | messages | $O(n)$ |
| $F_i$ | failure notifications | $O(f \cdot d)$ |
| $\mathbf{g}_i$ | tracking digraphs | $O(f^2 \cdot d)$ |
| $Q$ | FIFO queue | $O(f \cdot d)$ |

**Table 2: Space complexity per server for data structures used by Algorithm 1. The space complexity for $G$ holds for regular digraphs, such as $G_S(n, d)$ § 4.4.**

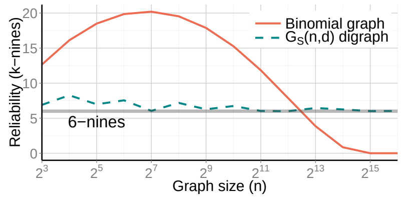


**Figure 5: AllConcur's reliability estimated over a period of 24 hours and a server $MTTF \approx 2$ years.**

have more than one vertex. The space complexity of the other data structures is straightforward (see Table 2).

## 4.4 Choosing the digraph $G$

AllConcur's performance depends on the parameters of $G$—degree, diameter, and fault diameter. Binomial graphs have both diameter and fault diameter lower than other commonly used graphs, such as the binary Hypercube [5]. Also, they are optimally connected, hence, offering optimal work for the provided connectivity. Yet, their connectivity depends on the number of vertices, which reduces their flexibility: Binomial graphs provide either too much or not enough connectivity.

We estimate AllConcur's reliability by $\rho_G = \sum_{i=0}^{k(G)-1} C(n, i) \cdot p_f^i (1 - p_f)^{n-i}$, with $p_f = 1 - e^{-\frac{\Delta}{MTTF}}$ the probability of a server to fail over a period of time $\Delta$ (§ 4.2.2). Figure 5 plots this reliability as a function of $n$. For a reliability target of 6-nines, we can see that the binomial graph offers either too much reliability, resulting in unnecessary work, or not sufficient reliability.

As an alternative, AllConcur uses $G_S(n, d)$ digraphs, for any $d \geq 3$ and $n \geq 2d$ [58]. In a nutshell, the construction of $G_S(n, d)$ entails constructing the line digraph of a generalized de Bruijn digraph [23] with the self-loops replaced by cycles. A more detailed description that follows the steps provided in the original paper [58] is available in an extendend technical report [53]. Similarly to binomial graphs [5], $G_S(n, d)$ digraphs are optimally connected. Contrary to binomial graphs though, they can be adapted to various reliability targets (see Figure 5 for a reliability target of 6-nines). Moreover, $G_S(n, d)$ digraphs have a quasiminimal diameter for $n \leq d^3 + d$: The diameter is at most one larger than the lower bound obtained from the Moore bound, i.e., $DL(n, d) = \lceil \log_d (n(d-1) + d) \rceil - 1$. In addition, $G_S(n, d)$ digraphs have low fault diameter bounds (experimentally verified). Table 3 shows the parameters of $G_S(n, d)$ for different number of vertices and 6-nines reliability; the reliability is estimated over a period of 24 hours according to the data from the TSUBAME2.5 system failure history [28, 54], i.e., server $MTTF \approx 2$ years.

| $G_S(n, d)$ | $D$ | $DL(n, d)$ | $G_S(n, d)$ | $D$ | $DL(n, d)$ |
|---|---|---|---|---|---|
| $G_S(6, 3)$ | 2 | 2 | $G_S(64, 5)$ | 4 | 3 |
| $G_S(8, 3)$ | 2 | 2 | $G_S(90, 5)$ | 3 | 3 |
| $G_S(11, 3)$ | 3 | 2 | $G_S(128, 5)$ | 4 | 3 |
| $G_S(16, 4)$ | 2 | 2 | $G_S(256, 7)$ | 4 | 3 |
| $G_S(22, 4)$ | 3 | 3 | $G_S(512, 8)$ | 3 | 3 |
| $G_S(32, 4)$ | 3 | 3 | $G_S(1024, 11)$ | 4 | 3 |
| $G_S(45, 4)$ | 4 | 3 | | | |

**Table 3: The parameters—vertex count $n$, degree $d$ and diameter $D$—of $G_S(n, d)$ for 6-nines reliability (estimated over a period of 24 hours and a server $MTTF \approx 2$ years). The lower bound for the diameter is $DL(n, d) = \lceil \log_d (n(d-1) + d) \rceil - 1$.**

## 4.5 AllConcur vs. leader-based agreement

For a theoretical comparison to leader-based agreement, we consider the following deployment: a leader-based group, such as Paxos, that enables the agreement among $n$ servers, i.e., clients in Paxos terminology (see Figure 1a). The group size does not depend on $n$, but only on the reliability of the group members. Also, all the servers interact directly with the leader. In principle, the leader can disseminate state updates via a tree [32]; yet, for fault-tolerance, a reliable broadcast algorithm [12] is needed. To the best of our knowledge, there is no implementation of leader-based agreement that uses reliable broadcast for dissemination.

In general, in such a leader-based deployment, not all servers need to send a message. This is an advantage over AllConcur, where the early termination mechanism requires every server to send a message. Yet, for the typical scenarios targeted by AllConcur—the data to be agreed upon is well balanced—we can realistically assume that all servers have a message to send.

**Trade-off between work and total message count.** The work require for reaching agreement in a leader-based deployment is unbalanced. On the one hand, every server sends one message and receives $n - 1$ messages, resulting in $O(n)$ work. On the other hand, the leader requires quadratic work, i.e., $O(n^2)$: it receives one message from every server and it sends every received message to all servers. Note that every message is also replicated, adding a constant amount of work per message.

To avoid overloading a single server (i.e., the leader), AllConcur distributes the work evenly among all servers—every server performs $O(nd)$ work (§ 4.1). This decrease in complexity comes at the cost of introducing more messages to the network. A leader-based deployment introduces $n(n - 1)$ messages to the network (not counting the messages needed for replication). In AllConcur, every message is sent $d$ times; thus, the total number of messages in the network is $n^2 d$.

**Removing and adding servers.** For both AllConcur and leader-based agreement, the cost of intentionally removing and adding servers can be hidden by using a two-phase approach similar to the transitional configuration in Raft [51]. Thus, we focus only on the cost of unintentionally removing a server—a server failure. Also, we consider a worst-case analysis—we compare the impact of a leader failure to that of a AllConcur server. The consequence of a leader failure is threefold: (1) every server receives one failure notification; (2) a leader election is triggered; and (3) the new leader needs to reestablish the connections to the *n* servers. Note that the cost of reestablishing the connection can be hidden if the servers connect from the start to all members of the group. In ALLCONCUR, there is no need for leader election. A server failure causes every server to receive up to $d^2$ failure notifications (§ 4.1). Also, the depth may increase (§ 4.2.2).

**Redundancy.** The amount of redundancy (i.e., *d*) needed by ALLCONCUR is given by the reliability of the agreeing servers. Thus, *d* can be seen as a performance penalty for requiring a certain level of reliability. Using more reliable hardware increases ALLCONCUR's performance. In contrast, in a leader-based deployment, more reliable hardware increases only the performance of message replication (i.e., less replicas are needed), leaving both the quadratic work and the quadratic total message count unchanged.

## 5 EVALUATION

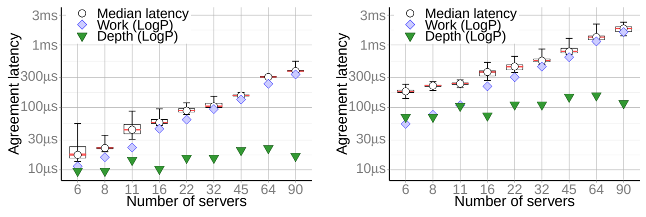
We evaluate ALLCONCUR on two production systems: (1) an InfiniBand cluster with 96 nodes; and (2) the Hazel Hen Cray XC40 system (7712 nodes). We refer to the two systems as IB-hsw and XC40, respectively. On both systems, each node has 128GB of physical memory and two Intel Xeon E5-2680v3 12-core CPUs with a base frequency of 2.5GHz. The IB-hsw system nodes are connected through a Voltair 4036 Fabric (40Gbps); each node uses a single Mellanox ConnectX-3 QDR adapter (40GBps). Moreover, each node is running ScientificLinux version 6.4. The XC40 system nodes are connected through the Cray Aries network.

We implemented ALLCONCUR[4] in C; the implementation relies on *libev*, a high-performance event loop library. Each instance of ALLCONCUR is deployed on a single physical node. The nodes communicate via either standard sockets-based TCP or high-performance InfiniBand Verbs (IBV); we refer to the two variants as ALLCONCUR-TCP and ALLCONCUR-IBV, respectively. On the IB-hsw system, to take advantage of the high-performance network, we use TCP/IP over InfiniBand ("IP over IB") for ALLCONCUR-TCP. The failure detector is implemented over unreliable datagrams. To compile the code, we use GCC version 5.2.0 on the IB-hsw system and Cray Programming Environment 5.2.82 on the XC40 system.

We evaluate ALLCONCUR through a set of benchmarks that emulate representative real-world applications. During the evaluation, we focus on two common performance metrics: (1) the *agreement latency*, i.e., the time needed to reach agreement; and (2) the *agreement throughput*, i.e., the amount of data agreed upon per second. In addition, we introduce the *aggregated throughput*, a performance metric defined as the agreement throughput times the number of servers. Also, all the experiments assume a perfect FD.

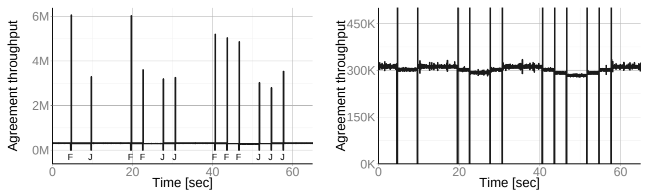
In the following benchmarks, the servers are interconnected via $G_S(n, d)$ digraphs (see Table 3). If not specified otherwise, each server generates requests at a certain rate. The requests are buffered until the current agreement round is completed; then, they are packed into a message that is A-broadcast in the next round. All the figures

[4]Source code: https://github.com/mpoke/allconcur/commit/c09dee8f8f186ee7b2d4fdb23e682016eb3dbde8



**(a) ALLCONCUR-IBV [IB-hsw]** **(b) ALLCONCUR-TCP [IB-hsw]**

**Figure 6: Agreement latency for a single (64-byte) request. The LogP parameters are $L = 1.25\mu s$ and $o = 0.38\mu s$ over IBV and $L = 12\mu s$ and $o = 1.8\mu s$ over TCP.**



**(a) ALLCONCUR-IBV [IB-hsw]** **(b) Zoom-in of (a)**

**Figure 7: Agreement throughput during membership changes—servers failing, indicated by F, and servers joining, indicated by J. Deployment over 32 servers, each generating 10,000 (64-byte) requests per second. The FD has $\Delta_{hb} = 10ms$ and $\Delta_{to} = 100ms$. The spikes in throughput are due to the accumulated requests during unavailability periods.**

report both the median and the 95% nonparametric confidence interval around it [31]. Moreover, for each figure, the system used to obtain the measurements is specified in square brackets.

**Single request agreement.** To evaluate the LogP models described in Section 4, we consider a benchmark where the servers agree on one single request. Clearly, such a scenario is not the intended use case of ALLCONCUR, as all servers, except one, A-broadcast empty messages. Figure 6 plots the agreement latency as a function of system size for both ALLCONCUR-IBV and ALLCONCUR-TCP on the IB-hsw system and it compares it with the LogP models for both work and depth (§ 4). The LogP parameters for the IB-hsw system are $L = 1.25\mu s$ and $o = 0.38\mu s$ over IBV and $L = 12\mu s$ and $o = 1.8\mu s$ over TCP. The models are good indicators of ALLCONCUR's performance; e.g., with increasing the system size, work becomes dominant.

**Membership changes.** To evaluate the effect of membership changes on performance, we deploy ALLCONCUR-IBV on the IB-hsw system. In particular, we consider 32 servers each generating 10,000 (64-byte) requests per second. Servers rely on a heartbeat-based FD with a heartbeat period $\Delta_{hb} = 10ms$ and a timeout period $\Delta_{to} = 100ms$. Figure 7 shows ALLCONCUR's agreement throughput (binned into $10ms$ intervals) during a series of events, i.e., servers failing, indicated by F, and servers joining, indicated by J. Initially, one server fails, causing a period of unavailability ($\approx 190ms$); this is followed by a rise in throughput, due to the accumulated requests (see Figure 7a). Shortly after, the system stabilizes, but at a lower throughput since one server is missing. Next, a server joins the

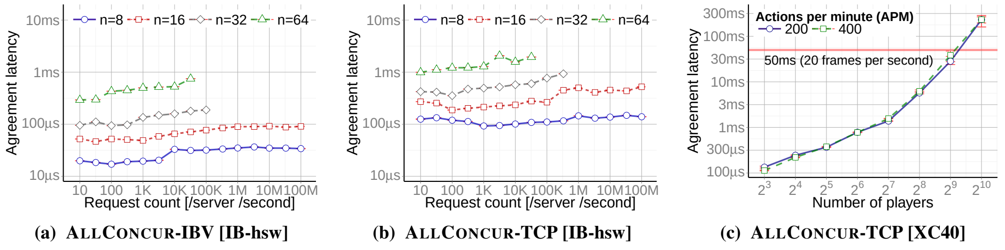


**Figure 8: (a),(b) Constant (64-byte) request rate per server. (c) Agreement latency in multiplayer video games for different APM and 40-byte requests.**

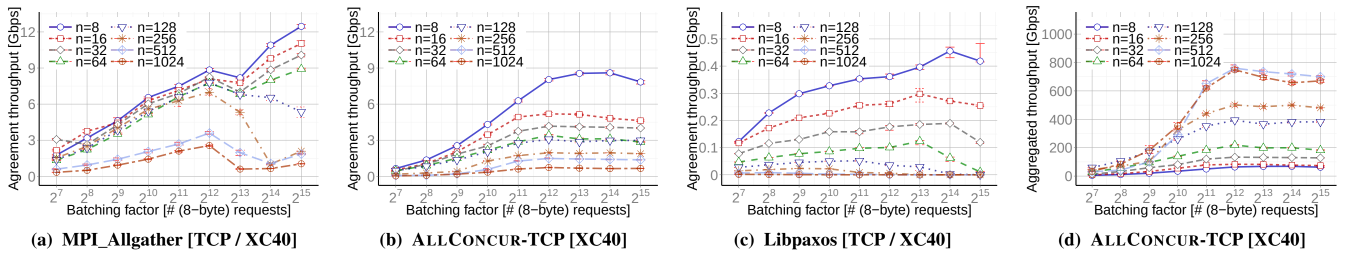


**Figure 9: (a) Unreliable agreement vs. (b) AllConcur vs. (c) leader-based agreement—batching factor effect on the agreement throughput. (d) Batching factor effect on the aggregated throughput.**

system causing another period of unavailability ($\approx 80ms$) followed by another rise in throughput. Similarly, this scenario repeats for two and three subsequent failures[5]. Note that both unavailability periods can be reduced. First, by improving the FD implementation, $\Delta_{to}$ can be significantly decreased [22]. Second, new servers can join the system as non-participating members until they established all necessary connections [51].

**Travel reservation systems.** In this scenario, each server's rate of generating requests is bounded by its rate of answering queries. We consider a benchmark where 64-byte requests are generated with a constant rate per server $r$. Since the *batching factor* (i.e., the amount of requests packed into a message) is not bounded, the system becomes unstable once the rate of generating requests exceeds the agreement throughput; this leads to a cycle of larger messages, leading to longer times, leading to larger messages etc. A practical deployment would bound the message size and reduce the inflow of requests. Figures 8a and 8b plot the agreement latency as a function of $r$; the measurements were obtained on the IB-hsw system. By using AllConcur-IBV, 8 servers, each generating 100 million requests per second, reach agreement in $35\mu s$; while 64 servers, each generating 32,000 requests per second, reach agreement in less than $0.75ms$. AllConcur-TCP has $\approx 3\times$ higher latency.

**Multiplayer video games.** In this scenario, the state is updated periodically, e.g., once every $50ms$ in multiplayer video games [8, 9]; thus, such systems are latency sensitive. Moreover, similarly to travel reservation systems, each server's rate of generating requests is bounded; e.g., in multiplayer video games, each player performs a limited number of actions per minute (APM), i.e., usually 200 APM, although expert players can exceed 400 APM [40]. To emulate such a scenario, we deploy AllConcur on the XC40 system; although not designed for video games, the system enables large-scale deployments. Figure 8c plots the agreement latency as a function of the number of players, for 200 and 400 APM. Each action causes a state update with a typical size of 40 bytes [8]. AllConcur-TCP supports the simultaneous interaction among 512 players with an agreement latency of $28ms$ for 200 APM and $38ms$ for 400 APM. Thus, AllConcur enables so called epic battles [10].

**AllConcur vs. unreliable agreement.** To evaluate the overhead of providing fault-tolerance, we compare AllConcur to an implementation of unreliable agreement. In particular, we use MPI_Allgather [49] to disseminate all messages to every server. We consider a benchmark where every server delivers a fixed-size message per round (fixed number of requests). Figures 9a and 9b plot the agreement throughput as a function of the batching factor. The measurements were obtained on the XC40 system; for a fair comparison, we used Open MPI [26] over TCP to run the benchmark. AllConcur provides a reliability target of 6-nines with an average overhead of 58%. Moreover, for messages of at least 2,048 (8-byte) requests, the overhead does not exceed 75%.

**AllConcur vs. leader-based agreement.** We conclude AllConcur's evaluation by comparing it to Libpaxos [57], an open-source implementation of Paxos [37, 38] over TCP. In particular, we use Libpaxos as the leader-based group in the deployment described in Section 4.5. The size of the Paxos group is five, sufficient for our reliability target of 6-nines. We consider the same benchmark used to compare to unreliable agreement—each server A-delivers a fixed-size message per round. Figures 9b and 9c plot the agreement throughput as a function of the batching factor; the measurements

[5]The $G_S(32, 4)$ has vertex-connectivity four; thus, in general, it cannot safely sustain more than three failures

were obtained on the XC40 system. The throughput peaks at a certain message size, indicating the optimal batching factor to be used. ALLCONCUR-TCP reaches an agreement throughput of $8.6Gbps$, equivalent to $\approx 135$ million (8-byte) requests per second (see Figures 9b). As compared to Libpaxos, ALLCONCUR achieves at least 17× higher throughput (see Figure 9c). The drop in throughput (after reaching the peak), for both ALLCONCUR and Libpaxos, is due to the TCP congestion control mechanism.

ALLCONCUR's agreement throughput decreases with increasing the number of servers. The reason for this performance drop is twofold. First, to maintain the same reliability, more servers entail a higher degree for $G$ (see Table 3), hence, more redundancy. Second, agreement among more servers entails more synchronization. Yet, the number of agreeing servers is an input parameter. Thus, a better metric to measure ALLCONCUR's actual performance is the aggregated throughput. Figure 9d plots the aggregated throughput corresponding to the agreement throughput from Figures 9b. ALLCONCUR-TCP's aggregated throughput increases with the number of servers and it peaks at $\approx 750Gbps$ for 512 and 1,024 servers.

## 6 RELATED WORK

Many existing algorithms and systems can be used to implement atomic broadcast; we discuss here only the most relevant subset. Défago, Schiper, and Urbán provide a general overview of atomic broadcast algorithms [19]. They define a classification based on how total order is established: by the sender, by a sequencer or by the destinations [14]. ALLCONCUR uses destinations agreement to achieve total order, i.e., agreement on a message set. Yet, unlike other destinations agreement algorithms, ALLCONCUR is entirely decentralized and requires no leader.

Lamport's classic Paxos algorithm [37, 38] is often used to implement atomic broadcast. Several practical systems have been proposed [11, 16, 34, 45]. Also, a series of optimizations were proposed, such as distributing the load among all servers or out-of-order processing of not-interfering requests [39, 44, 48]. Yet, the commonly employed simple replication scheme is not designed to scale to hundreds of instances.

State machine replication protocols are similar to Paxos but often claim to be simpler to understand and implement. Practical implementations include ZooKeeper [33], Viewstamped Replication [43], Raft [51], Chubby [13] and DARE [52] among others. These systems commonly employ a leader-based approach which makes them fundamentally unscalable. Increasing scalability comes often at the cost of relaxing the consistency model [18, 42]. Moreover, even when scalable strong consistency is provided [27], these systems aim to increase data reliability, an objective conceptually different than distributed agreement.

Bitcoin [50] offers an alternative solution to the (Byzantine fault-tolerant) atomic broadcast problem: It uses *proof-of-work* to order the transactions on a distributed ledger. In a nutshell, a server must solve a cryptographic puzzle in order to add a block of transactions to the ledger. Yet, Bitcoin does not guarantee *consensus finality* [60]—multiple servers solving the puzzle may lead to a fork (conflict), resulting in branches. Forks are eventually solved by adding new blocks. Eventually one branch outpaces the others, thereby becoming the ledger all servers agree upon. To avoid frequent forks, Bitcoin controls the expected puzzle solution time to 10 minutes and currently limits the block size to 1MB, resulting in limited performance, i.e., around seven transactions per second. To increase performance, Bitcoin-NG [24] uses proof-of-work to elect a leader that can add blocks until a new leader is elected. Yet, conflicts are still possible and consensus finality is not ensured.

## 7 CONCLUSION

In this paper we present ALLCONCUR: a distributed agreement system that relies on a novel leaderless atomic broadcast algorithm. ALLCONCUR uses a digraph $G$ as overlay network; thus, the fault-tolerance $f$ is given by $G$'s vertex-connectivity $k(G)$ and can be adapted freely to the system specific requirements. We show that ALLCONCUR achieves competitive latency and throughput for two real-world scenarios. In comparison to Libpaxos, ALLCONCUR achieves at least 17× higher throughput for the considered scenario. We prove ALLCONCUR's correctness under two assumptions—$f < k(G)$ and a perfect failure detector. Moreover, we show that if $f \geq k(G)$, ALLCONCUR still guarantees safety, and we discuss the changes necessary to maintain safety when relaxing the assumption of a perfect failure detector.

In summary, ALLCONCUR is highly competitive and, due to its decentralized approach, enables hitherto unattainable system designs in a variety of fields.

## ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) as part of the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart. We thank Michael Resch for support; our shepherd Samer Al Kiswany and the anonymous reviewers; Nitin H. Vaidya, José Gracia and Daniel Rubio Bonilla for helpful discussions; and Holger Berger for providing support with the InfiniBand machine.

## REFERENCES

[1] Marcos Kawazoe Aguilera and Sam Toueg. 1999. A simple bivalency proof that t-resilient consensus requires t+1 rounds. *Inform. Process. Lett.* 71, 3 (1999), 155 – 158. https://doi.org/10.1016/S0020-0190(99)00100-3
[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. 1983. *Data Structures and Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
[4] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. 1995. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95).* Santa Barbara, CA, USA. https://doi.org/10.1145/215399.215427
[5] Thara Angskun, George Bosilca, and Jack Dongarra. 2007. Binomial Graph: A Scalable and Fault-tolerant Logical Network Topology. In *Proc. 5th International Conference on Parallel and Distributed Processing and Applications (ISPA'07).* Niagara Falls, Canada. https://doi.org/10.1007/978-3-540-74742-0_43
[6] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons.
[7] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. 2004. The Effects of Loss and Latency on User Performance in Unreal Tournament 2003®. In *Proc. ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '04).* Portland, OR, USA. https://doi.org/10.1145/1016540.1016556
[8] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. 2008. Donnybrook: Enabling Large-scale, High-speed, Peer-to-peer Games. In *Proc. ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM '08).* Seattle, WA, USA. https://doi.org/10.1145/1402958.1403002

[9] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. 2006. Colyseus: A Distributed Architecture for Online Multiplayer Games. In *Proc. 3rd Conference on Networked Systems Design & Implementation (NSDI'06)*. San Jose, CA, USA. http://dl.acm.org/citation.cfm?id=1267680.1267692

[10] Blizzard Entertainment. 2008. WoW PvP battlegrounds. (2008). http://www.worldofwarcraft.com/pvp/battlegrounds.

[11] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. 2003. Reconstructing Paxos. *SIGACT News* 34(2) (2003).

[12] Darius Buntinas. 2012. Scalable Distributed Consensus to Support MPI Fault Tolerance. In *Proc. 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS'12)*. Shanghai, China. https://doi.org/10.1109/IPDPS.2012.113

[13] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. Seattle, WA, USA. http://dl.acm.org/citation.cfm?id=1298455.1298487

[14] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[15] F. R. K. Chung and M. R. Garey. 1984. Diameter bounds for altered graphs. *Journal of Graph Theory* 8, 4 (December 1984), 511–534. https://doi.org/10.1002/jgt.3190080408

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Ka nthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. Hollywood, CA, USA. http://dl.acm.org/citation.cfm?id=2387880.2387905

[17] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a Realistic Model of Parallel Computation. *SIGPLAN Not.* 28, 7 (July 1993), 1–12. https://doi.org/10.1145/173284.155333

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (December 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[19] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.* 36, 4 (December 2004), 372–421. https://doi.org/10.1145/1041680.1041682

[20] Anthony H. Dekker and Bernard D. Colbert. 2004. Network Robustness and Graph Topology. In *Proc. 27th Australasian Conference on Computer Science - Volume 26 (ACSC '04)*. Dunedin, New Zealand. http://dl.acm.org/citation.cfm?id=979922.979965

[21] Danny Dolev and Christoph Lenzen. 2013. Early-deciding Consensus is Expensive. In *Proc. 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. Montréal, Québec, Canada. https://doi.org/10.1145/2484239.2484269

[22] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proc. 25th Symposium on Operating Systems Principles (SOSP '15)*. Monterey, CA, USA. https://doi.org/10.1145/2815400.2815425

[23] D. Z. Du and F. K. Hwang. 1988. Generalized De Bruijn Digraphs. *Netw.* 18, 1 (March 1988), 27–38. https://doi.org/10.1002/net.3230180105

[24] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *Proc. 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. Santa Clara, CA, USA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[25] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[26] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proc. 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary. https://doi.org/10.1007/978-3-540-30218-6_19

[27] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable Consistency in Scatter. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. Cascais, Portugal. https://doi.org/10.1145/2043556.2043559

[28] Global Scientific Information and Computing Center. 2014. Failure History of TSUBAME2.0 and TSUBAME2.5. (2014). http://mon.g.gsic.titech.ac.jp/trouble-list/index.htm.

[29] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report. Ithaca, NY, USA.

[30] Debra Hensgen, Raphael Finkel, and Udi Manber. 1988. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.* 17, 1 (February 1988), 1–17. https://doi.org/10.1007/BF01379320

[31] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Austin, TX, USA. https://doi.org/10.1145/2807591.2807644

[32] Torsten Hoefler and Dmitry Moor. 2014. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Supercomput. Front. Innov.: Int. J.* 1, 2 (July 2014), 58–75. https://doi.org/10.14529/jsfi140204

[33] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. 2010 USENIX Annual Technical Conference (ATC'10)*. Boston, MA, USA. http://dl.acm.org/citation.cfm?id=1855840.1855851

[34] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: An Overview. In *Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*. Yorktown Heights, NY, USA. https://doi.org/10.1145/1529974.1529979

[35] M. S. Krishnamoorthy and B. Krishnamurthy. 1987. Fault Diameter of Interconnection Networks. *Comput. Math. Appl.* 13, 5-6 (April 1987), 577–582. https://doi.org/10.1016/0898-1221(87)90085-X

[36] Leslie Lamport. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)* 2, 2 (May 1978), 95 – 114. https://doi.org/10.1016/0376-5075(78)90045-4

[37] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[38] Leslie Lamport. 2001. Paxos Made Simple. *SIGACT News* 32, 4 (December 2001), 51–58. https://doi.org/10.1145/568425.568433

[39] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

[40] Joshua M Lewis, Patrick Trinh, and David Kirsh. 2011. A Corpus Analysis of Strategy Video Game Play in Starcraft: Brood War. In *Proc. 33rd Annual Conference of the Cognitive Science Society*. Austin, TX, USA.

[41] Chung-Lun Li, Thomas S. McCormick, and David Simich-Levi. 1990. The Complexity of Finding Two Disjoint Paths with Min-max Objective Function. *Discrete Appl. Math.* 26, 1 (January 1990), 105–115. https://doi.org/10.1016/0166-218X(90)90024-7

[42] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. 2013. ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table. In *Proc. 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. Boston, MA, USA. https://doi.org/10.1109/IPDPS.2013.110

[43] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT.

[44] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proc. 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. San Diego, CA, USA. http://dl.acm.org/citation.cfm?id=1855741.1855767

[45] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-Ring Paxos. In *Proc. 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*. Boston, MA, USA. https://doi.org/10.1109/DSN.2012.6263916

[46] Mesosphere. 2017. DC/OS. (2017). https://docs.mesosphere.com/overview/.

[47] F. J. Meyer and D. K. Pradhan. 1988. Flip-Trees: Fault-Tolerant Graphs with Wide Containers. *IEEE Trans. Comput.* 37, 4 (April 1988), 472–478. https://doi.org/10.1109/12.2194

[48] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Farminton, PA, USA. https://doi.org/10.1145/2517349.2517350

[49] MPI Forum. 2015. MPI: A Message-Passing Interface Standard Version 3.1. (June 2015). http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[50] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008). http://bitcoin.org/bitcoin.pdf.

[51] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. 2014 USENIX Annual Technical Conference (ATC'14)*. Philadelphia, PA, USA. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[52] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. Portland, OR, USA. https://doi.org/10.1145/2749246.2749267

[53] Marius Poke, Torsten Hoefler, and Colin W. Glass. 2016. AllConcur: Leaderless Concurrent Atomic Broadcast (Extended Version). *CoRR* abs/1608.05866 (2016). http://arxiv.org/abs/1608.05866

[54] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. 2012. Design and Modeling of a Non-blocking Checkpointing System. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. Salt Lake City, UT, USA. https://doi.org/10.1109/SC.2012.46

[55] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (December 1990), 299–319. https://doi.org/10.1145/98163.98167

[56] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. 2008. Embracing diversity in the Barrelfish manycore operating system. In *Proc. Workshop on Managed Many-Core Systems*. Boston, MA, USA. https://www.microsoft.com/en-us/research/publication/embracing-diversity-in-the-barrelfish-manycore-operating-system/

[57] Daniele Sciascia. 2013. Libpaxos3. (2013). http://libpaxos.sourceforge.net/paxos_projects.php.

[58] Terunao Soneoka, Makoto Imase, and Yoshifumi Manabe. 1996. Design of a d-connected digraph with a minimum number of edges and a quasiminimal diameter II. *Discrete Appl. Math.* 64, 3 (February 1996), 267–279. https://doi.org/10.1016/0166-218X(94)00113-R

[59] Philipp Unterbrunner, Gustavo Alonso, and Donald Kossmann. 2014. High Availability, Elasticity, and Strong Consistency for Massively Parallel Scans over Relational Data. *The VLDB Journal* 23, 4 (August 2014), 627–652. https://doi.org/10.1007/s00778-013-0343-9

[60] Marko Vukolić. 2016. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Proc. IFIP WG 11.4 Workshop on Open Research Problems in Network Security (iNetSec'15)*. Zurich, Switzerland. https://doi.org/10.1007/978-3-319-39028-4_9