Passing Messages while Sharing Memory

Marcos K. Aguilera
VMware Research

Naama Ben-David*
CMU

Irina Calciu
VMware Research

Rachid Guerraoui
EPFL

Erez Petrank
Technion

Sam Toueg
University of Toronto

ABSTRACT

We introduce a new distributed computing model called m&m that allows processes to both pass messages and share memory. Motivated by recent hardware trends, we find that this model improves the power of the pure message-passing and shared-memory models. As we demonstrate by example with two fundamental problems—consensus and eventual leader election—the added power leads to new algorithms that are more robust against failures and asynchrony. Our consensus algorithm combines the superior scalability of message passing with the higher fault tolerance of shared memory, while our leader election algorithms reduce the system synchrony needed for correctness. These results point to a wide new space for future exploration of other problems, techniques, and benefits.

ACM Reference Format:
Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. 2018. Passing Messages while Sharing Memory. In *PODC '18: ACM Symposium on Principles of Distributed Computing, July 23–27, 2018, Egham, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3212734.3212741

1 INTRODUCTION

The distributed computing community has a dichotomy between shared-memory and message-passing models. Books, courses, and papers explicitly separate these models to present results and algorithms. These models differ based on how processes communicate. In the shared-memory model, processes can write and read data in a common area of memory. In the message-passing model, processes can send and receive messages to and from each other.

In this paper, we investigate the benefits of a hybrid model, called *message-and-memory model* or simply *m&m model*, where processes can both pass messages and share memory. We are motivated by recent technologies that permit exactly that, such as Remote Direct Memory Access (RDMA) [39, 41, 65], disaggregated memory [55], and Gen-Z [33]. By using both methods of communication, one can devise a new genre of algorithms that could potentially combine the advantages of shared-memory and message-passing algorithms.

It has been proven that the message-passing and shared-memory models are equivalent [11], by demonstrating that one model can simulate the other. If that is so, what could be the benefit of combining two models that are equivalent? Closer inspection of the equivalence result reveals that it holds only in a certain sense and under some assumptions. In particular, the equivalence is computational: it shows how algorithms for a problem in one model can be translated to the other using an emulation. However, the emulation does not preserve efficiency or synchrony (e.g., a timely process in one model can become untimely as it waits for other processes). Moreover, the emulation requires the assumption that the message-passing model has a majority of correct processes, and that processes know each other's identities as well as the number of processes in the system. In many cases these assumptions do not hold.

In fact, we find that each model has its own advantages and they benefit algorithms in different ways. Our contribution is to propose the new m&m model that merges the capabilities of the pure models, to identify some of the advantages of each pure model, and to show that they can be combined in the m&m model.

To illustrate the advantages of combining the models, consider the fundamental problem of mutual exclusion [26]. In this problem, there is a doorway and a critical section. The goal is to ensure that at most one process in the doorway enters the critical section at a time. The traditional algorithms for this problem, such as the bakery algorithm or Dijkstra's algorithm, have been designed in the shared-memory read-write model [51]. In that abstract model, these algorithms have a common drawback: while some process is in the critical section, other processes in the doorway must spin on one or more shared-memory locations to know when the critical section becomes empty again. Much work was devoted to mitigate this problem, by making the spin local to each process (e.g., [23, 46–48]).

By allowing an algorithm to share memory and pass messages in the m&m model, one finds very simple solutions that do not have to spin: upon leaving the critical section, a process could send a message to the processes in the doorway; those processes, rather than spinning, go to sleep and resume execution when a message arrives. This ability to react to data without spinning is a characteristic of message passing. In practice, by avoiding the spin, the CPU can be better utilized to run other processes. Such benefits escape the abstract equivalence of the message-passing and shared-memory models in [11].

Besides benefits, each model has limitations that the m&m model can overcome. Shared-memory systems have worse scalability than message-passing systems due to hardware limitations. For example, a typical shared-memory system today has tens to thousands of processes, while message-passing systems can be much larger (e.g.,

*Work done in part while the author was at VMware Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODC '18, July 23–27, 2018, Egham, United Kingdom
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5795-1/18/07…$15.00
https://doi.org/10.1145/3212734.3212741

map-reduce systems with tens of thousands of hosts [25], peer-to-peer systems with hundreds of thousands of hosts, the SMTP email system and DNS with hundreds of millions of hosts, etc).

On the other hand, message-passing systems have limitations on fault tolerance and synchrony. On fault tolerance, some fundamental problems in distributed computing—such as consensus and atomic storage—require a majority of correct processes to be solvable in the message-passing model, even under strong partial synchrony assumptions, whereas the same problems can be solved in shared memory with an arbitrary number of correct processes using wait-free algorithms under partial synchrony, randomization, or stronger hardware primitives. With respect to synchrony, due to engineering reasons, message-passing systems have larger variances in their timing than shared memory: the slowest and fastest message delays can be eight orders of magnitude apart (microseconds to tens of seconds), while the variance in the execution speeds of processes in shared memory is much smaller. As a result, partial synchrony bounds tend to be much worse in message-passing systems.

In this paper, we make these advantages and drawbacks more precise. We show that the m&m model can enhance the *robustness* of algorithms. We consider two aspects of robustness: the number of process crashes that algorithms can tolerate and the synchrony assumptions required by them. We illustrate how we improve the first aspect through the consensus problem and the second through the eventual leader election problem. In short, we show that message passing and shared memory complement each other when combined, allowing algorithms that have inherent advantages over the pure message-passing and shared-memory models.

For consensus, we give an algorithm called Hybrid Ben-Or or simply HBO that tolerates more than a majority of failures (like shared-memory algorithms can), while scaling to a large system size (like message-passing systems can). To scale to a large system, the algorithm limits the number of processes that share a given shared-memory location to a small constant number. We define an undirected *shared-memory graph*, whose nodes are processes and there is an edge between processes p and q if they share a memory location. Due to hardware limitations, to scale the system we must limit the maximum degree d of this graph [28, 43]. Our algorithm employs expander graphs to tolerate a majority of crash failures—up to $f < (1-\frac{1}{2(1+h)})n$ of them—in a system with n processes, where h is the expansion of the graph as measured by the *vertex expansion ratio*. Roughly, this ratio indicates by how much a set of vertices expands each time we add their neighbors to the set. The higher the expansion, the more failures the algorithm can tolerate. Our algorithm is a simulation of a pure message-passing consensus algorithm that requires a majority of correct processes, without having that majority in reality. To do that, we use a wait-free shared-memory consensus algorithm among each local neighborhood in the shared-memory graph to emulate a virtual process in the larger message-passing algorithm. This virtual process fails only if all processes in its neighborhood fail. By taking a shared-memory graph with high expansion, we ensure that even with a small d, many processes can fail without affecting a majority of virtual processes. Here, the topology of the shared-memory graph determines the fault tolerance of consensus: graphs with higher expansion allow for higher fault tolerance, because correct processes are adjacent to (and thus can simulate) more processes. We show that this relation is inherent by giving an impossibility result relating graph expansion and fault tolerance.

We next turn our attention to the (eventual) leader election problem. This problem is traditionally considered in message-passing systems, and much effort has been devoted to find the weakest synchrony needed to solve it [5, 6, 38]. Prior algorithms required some timeliness on processes *and* communication links. We show that in the m&m model, we can do better by using both shared memory and message passing to obtain different benefits. We use shared memory to reduce the timeliness requirement to processes only: our leader election algorithms require only that a *single* process, the leader, increments its local heartbeat in a timely fashion; other processes can be asynchronous—in particular, they can be arbitrarily slow to read the heartbeat. We use message passing to provide a trade-off between message reliability and amount of work in steady state: we give two algorithms for different types of links. (1) With *reliable* links[1], the only steady-state work is that the leader periodically increments a local heartbeat counter in shared memory, while other processes read the counter; (2) With *fair lossy* links[2], in addition to the above, the leader also periodically *reads* a shared register. In either algorithm, no messages are exchanged in the steady state, and all the communication links can be *asynchronous*. We further prove that the two leader election algorithms are optimal in the following sense. For systems with a timely process and asynchronous links, any algorithm requires the leader to write a shared register periodically (as in both our algorithms). This result holds whether links are reliable or fair lossy. Moreover, with fair lossy links, there are more requirements: either the leader writes and reads shared registers periodically (as our second algorithm), or some process keeps sending messages forever.

To summarize, the contributions of this paper are the following:

- We motivate and introduce the m&m model of distributed computing, which allows processes to both share memory and pass messages.
- We study the consensus problem under the m&m model. We give a new algorithm that improves on the fault tolerance of message-passing algorithms, while limiting memory sharing to allow for scalability. We show that the algorithm's fault tolerance is improved by a shared-memory graph with high expansion, and prove an impossibility result showing that this relationship to expansion is inherent.
- The new consensus algorithm introduces a simulation technique that combines shared memory and message passing. We believe this technique is interesting in its own right, as it could be used to improve other algorithms, to show impossibility results, etc.
- We study the leader election problem under the m&m model and give algorithms that reduce the synchrony required, while maintaining a low communication complexity. We show that these algorithms are tight in their communication.

[1] I.e., links that do not drop messages.

[2] I.e., links that can drop messages, but if a message is repeatedly sent then it is eventually received.

While we focus on two problems and some benefits of the m&m model, we believe the paper opens a large space for future research on other problems and benefits.

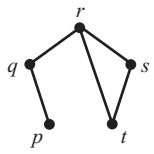
Due to space limitations, we omit proofs and some details. They will be included in the full version of this paper.

2 RELATED WORK

The m&m model is motivated by emerging technologies, such as Remote Direct Memory Access (RDMA) [39, 41, 65], disaggregated memory [31, 55], Gen-Z [33], and OmniPath [40]. These technologies provide remote memory [4] and can be unified under higher-level abstractions [3]. RDMA permits a process to access the memory of a remote host without interrupting the remote processor. It has been widely used in high-performance computing [72] and is now being adopted in modern data centers [34]. Work on RDMA shows how it can improve the performance of important applications, such as key-value storage systems [28, 43, 60], database systems [13, 73], distributed file systems [58], and more [29, 36, 67, 68]. Recent work uses RDMA to improve performance of consensus [64, 70] assuming a majority of processes are correct. Disaggregated memory separates compute and memory, and connects them using a fast network; prior work proposes new architectures for disaggregated memory [9, 22, 62] and studies the network [35] and system [4, 56] requirements for a practical implementation. Gen-Z and OmniPath are commercial technologies under development that offer memory semantics and low-latency access to remote data.

The shared-memory and the message-passing models are well studied in academic research, and have been compared under both theoretical and practical considerations [17, 18, 49, 54]. The two models have been shown to be computationally equivalent [11], though for efficiency, simplicity, or hardware availability, one might prefer one model over the other. For instance, Barrelfish [14] uses message passing to improve performance on a shared-memory multicore machine [24]. Conversely, distributed shared-memory systems [8, 16, 66] offer the abstraction of shared memory on top of a message-passing system. Recent work improves the performance of such systems using RDMA [45, 61]. Integrating message passing and shared memory in hardware has been explored in the MIT Alewife machine [50]. Our work differs in that (1) we propose a new abstract distributed computing model, which encapsulates low-latency remote access technologies, such as RDMA and disaggregated memory, and (2) we show that this model can improve the robustness of algorithms, rather than the performance or simplicity of applications.

Consensus is a fundamental problem in distributed computing. Following the well-known FLP result [32] showing that it cannot be solved in an asynchronous crash-prone message-passing system, much work has focused on getting around the impossibility by using randomization [10, 12, 15], partial synchrony [27, 30], or unreliable failure detectors [19, 20]. In fact, the eventual leader election problem, Ω, is the weakest failure detector that can solve consensus, and is used in several algorithms [53, 63]. This is the problem that we study in the second part of the paper. The eventual leader election problem is known to require some synchrony to implement. We show that the m&m model permits solutions with less synchrony than before, while minimizing the work done.



$$S_p = \{p, q\} \qquad S_s = \{r, s, t\}$$
$$S_q = \{p, q, r\} \qquad S_t = \{r, s, t\}$$
$$S_r = \{q, r, s, t\}$$

$$\mathcal{S} = \{S_p, S_q, S_r, S_s, S_t\}$$
$$= \{\{p, q\}, \{p, q, r\}, \{q, r, s, t\}, \{r, s, t\}\}$$

Figure 1: Example of a shared-memory graph G_{SM} and the resulting shared-memory domain $\mathcal{S}$, where p, q, r, s, t are processes. This shared-memory graph permits registers that can be shared among any of the S_x sets in $\mathcal{S}$. The graph is intended to model the underlying physical connections that implement the shared memory. For example, a register shared among S_r is physically kept in the host containing process r, and processes q, s, t access this register over the connections to r in the graph, while process p cannot access this register.

Expander graphs are graphs that are sparse, yet well-connected. They are well-studied and have applications in many areas of computer science, including distributed computing [21, 71]. In this paper, we show that the fault tolerance of the m&m model is tightly coupled with the expansion of its shared-memory connections, highlighting another problem in which expander graphs apply.

3 MODEL

We consider a distributed system with n processes $\Pi = \{0, \ldots, n-1\}$ that can communicate with each other using message passing and shared memory. Processes may fail by crashing, in which case they stop taking steps. If a process does not fail, it is said to be *correct* and it keeps taking steps forever. Some algorithms require a bound on the maximum number of processes that can crash; we denote this bound by f. A process has a local state that is private to itself, and it may communicate with others through message passing and shared memory.

Message passing. Processes can send messages over directed links. The link from p to q, denoted $p \rightarrow q$, is an *output link* of p and an *input link* of q. Each link $p \rightarrow q$ satisfies the following property in every run:

- *[Integrity]:* If q receives m from p k times then p previously sent m to q at least k times.

Some links may satisfy additional properties which are described next. We consider two types of links: reliable and fair lossy. Roughly, a reliable link does not drop messages, while a fair lossy link may drop messages but ensures that if a process sends a message repeatedly then it is eventually received.

More precisely, we say that a link $p \rightarrow q$ is *reliable* if it satisfies Integrity and the following property:

- *[No loss]*: If p sends a message m to q and q is correct then eventually q receives m from p.

We say that a link $p \rightarrow q$ is *fair lossy* if it satisfies Integrity and the following property:

- *[Fair loss]*: If p sends a message m infinitely often to q and q is correct, then q receives m infinitely often from p.

In this paper, we consider a fully connected network, that is, for any two processes $p \neq q$, there is a link from p to q.

Shared memory. Processes can also communicate via a set of *shared registers*. Here, we consider only atomic read-write registers.

In practice, a shared register is provided by the hardware, but not all processes may be able to access all registers, as there may be limits on the number of processes that can share the same memory [28, 43, 44, 69]. We define the *shared-memory domain* $\mathcal{S}$ as a set of process subsets; intuitively, $\mathcal{S}$ determines what subsets of processes can share memory. More precisely, for each set $S \in \mathcal{S}$, the model permits having any number of registers shared among processes in S. In general, $\mathcal{S}$ can be arbitrary. However, in practice memory sharing is simpler, as the hardware technology naturally imposes a structure on $\mathcal{S}$: for example, a process might be able to share memory only with processes that connect to it over the underlying hardware. We say that $\mathcal{S}$ is *uniform* if it can be represented by an undirected graph G_{SM} of processes, such that registers can be shared by a process and its neighbors in G_{SM}; intuitively, G_{SM} is the graph of connections of the underlying hardware that implements the shared memory. Formally, G_{SM} is a graph $G_{SM} = (\Pi, E_{SM})$ and the sets in $\mathcal{S}$ are exactly the sets consisting of a process p and its neighbors in G_{SM}. That is, if we let $S_p = \{p\} \cup \{q : (p, q) \in E_{SM}\}$ then $\mathcal{S} = \{S_p : p \in \Pi\}$. For a uniform $\mathcal{S}$, we say that G_{SM} is its *shared-memory graph*. Figure 1 gives an example. In this paper, we are interested in the uniform model, and all our results work with the graph G_{SM}. The broader model based on $\mathcal{S}$ is provided to allow for future theoretical work and potential new hardware platforms. Note that, while the model does not constrain the number or size of registers that can be shared, algorithms may choose to reduce their shared-memory usage for efficiency.

In systems with few processes (e.g., in the tens), G_{SM} could be a fully connected graph, but systems with lots of processes may have to limit the maximum degree of G_{SM} (e.g., limit the connections over the hardware).

We assume that the shared memory does not fail, as in the pure shared-memory model. This assumption can be supported by the hardware: with RDMA, the shared memory can be registered with the kernel so that it remains accessible after processes crash; disaggregated memory can similarly preserve memory accesses after process crashes.

Synchrony. For some results, we make some partial synchrony assumptions about the relative execution speed of processes. We first define what it means for a process p to be timely with respect to another process q:

- *[Pairwise timeliness]:* We say that p is *q-timely* (in a run) if p is correct and there is an integer $i \geq 1$ such that every time interval containing i steps of q has at least one step of p.

The timeliness bound i above is not known to processes: it may depend on each run and each pair of processes p and q. We now define what it means for a process p to be timely:

- *[Timeliness]:* We say that p is *timely* (in a run) if p is q-timely for every process $q \in \Pi$ (in this run).

Intuitively, timeliness means that eventually the process executes within a bounded rate relative to other processes. This is a weak requirement in many ways. First, it is relative to the speed of other processes, so a process can be timely even if it slows down arbitrarily in real time. Second, the bound on execution rate need not be known. Third, the bound is arbitrary and not fixed a priori, so timeliness can be satisfied even if a process is initially arbitrarily slow relative to others.

We consider two types of systems: (1) asynchronous systems, which might not have any timely processes, and (2) systems with little synchrony, that is, systems where at least one process is timely; this process can vary from run to run and is not known to the processes. We make no assumptions about the timeliness of messages.

Consensus problem. In the consensus problem, each process begins with an input value $v \in \{0, 1\}$ which it *proposes*, and it *decides* on an output value in the end. The output values must satisfy three properties:

- *[Uniform Agreement]:* No two processes decide on a different value.
- *[Validity]:* If a process decides on a value v then v was proposed by some process.
- *[Termination]:* Every correct process eventually decides.

We allow randomized solutions, where the above Termination property must hold with probability 1 under a *strong adversary*—one that can schedule processes based on their current state and past history.

A *consensus object* is a shared-memory object with one operation, *propose*(v), which takes a value v and returns the first value that was proposed to the object.

The eventual leader election problem. This problem is formally defined in [20] as the Ω failure detector. Informally, each process p outputs a single process denoted $leader_p$, such that the following property holds:[3]

- There is a correct process ℓ and a time after which, for every correct process p, $leader_p = \ell$.

Note that, at any given time, processes do not know if there is a commonly agreed leader; they only know that eventually there will be a common leader.

4 CONSENSUS

We show that the m&m model can be used improve the fault tolerance of algorithms compared to a pure message-passing system. It is known that consensus cannot be solved deterministically in an asynchronous system subject to failures, even if processes can only fail by crashing and at most one process may fail; this is true in both message-passing and shared-memory models [32, 57]. We thus consider asynchronous systems where processes can toss coins. Then, in a shared-memory system, consensus can be solved (with probability 1) with up to $n-1$ crash failures [1] (n is the number of processes in the system), whereas in a message-passing system, a consensus algorithm can tolerate at most $\lfloor (n-1)/2 \rfloor$ crash failures [15]. We show that the m&m model can strike a balance between shared memory and message passing.

First note that if the shared-memory graph G_{SM} is fully connected then any fault-tolerant shared-memory algorithm also works in the m&m model—the algorithm simply never sends messages. Thus, there are algorithms in the m&m model that can tolerate up to $n-1$ crash failures. However, in a large system, it is impractical to connect all processes over shared memory (§3). When fewer

[3] Henceforth, when we say that there is a time after which some property C holds, we mean that there is a time t such that, for every time $t' \geq t$, property C holds at time t'.

processes can share memory, we show that the m&m model provides a range of choices, where the fault tolerance increases as we improve the shared-memory graph. Specifically, we present an algorithm for the m&m model that tolerates anywhere between $\lfloor (n-1)/2 \rfloor$ and $n-1$ crash failures, depending on the topology of the shared-memory graph (§4.1). We discuss how to best choose this topology (§4.2), building on expander graphs [37]. We then give an impossibility result about the fault tolerance of consensus in the m&m model (§4.3).

4.1 Algorithm

The algorithm for the m&m model is based on Ben-Or's randomized algorithm [15], which can tolerate up to $f < n/2$ process crashes in the message-passing model. This is one of the simplest consensus algorithms, but not the most efficient one. Our goal here is to show feasibility; designing more efficient algorithms for the m&m model is future work.

Ben-Or's algorithm. In Ben-Or's algorithm, each process has an estimate of the decision value, which starts with the process's initial value. The algorithm proceeds in rounds, each with two phases. In the first phase (phase R), each process p sends its current estimate to all processes, waits to receive at least $n-f$ messages, and checks if more than $n/2$ messages have the same value v. If so, p sends this value to all processes in the second phase (phase P). Otherwise, p sends a special value '?' to all processes in the second phase. Process p then waits to receive at least $n-f$ messages. If at least $f+1$ of them have the same non-'?' value, p decides on this value. If at least one of them is a non-'?' value, p changes its estimate to that value. Otherwise, p changes its estimate to a random bit.

Ben-Or's algorithm satisfies the validity and uniform agreement properties of consensus (§3), and it satisfies the termination property with probability 1 if a majority of the processes are correct [7].

Simulating Ben-Or's algorithm in the m&m model. We modify Ben-Or's algorithm, so that correct processes simulate the actions of their neighbors in the shared-memory graph G_{SM}. The idea of the simulation is simple: when sending a message in any phase, process p sends not only its own value, but also the value that its neighbors are supposed to send. That is, p ensures that its neighbors progress at least as much as it does. To do so, for each neighbor q, p reaches agreement with q and q's neighbors on what q's message should be. Then, p sends a message of the form $(phase, round, [\langle q, val\rangle : q \in neighbors(p)])$, where *phase* is a phase (either R or P), *round* is a round number, and the last entry is an array with a tuple for each neighbor q indicating the agreed value of q's message. We say that the message *represents* each of the processes whose ids appear in the tuple. For a set of such messages, we say that the messages represent the union of the processes that are represented by each message separately.

To reach agreement on the message of a neighbor q, there are two arrays of consensus objects, one array for each phase, indexed by q and the round. All of q's neighbors use the same consensus object to determine what q's message might be for a given phase and round. Process q and its neighbors propose their own value to that consensus object. The consensus objects themselves are implemented using known wait-free randomized shared-memory algorithms [10, 12], which work in the m&m model because neighbors in G_{SM} share memory.

```
Shared objects:
  RVals[p, i]: consensus object accessible by {p} ∪ neighbors(p),
               ∀p ∈ Π, ∀i ∈ {1, 2, ...}
  PVals[p, i]: consensus object accessible by {p} ∪ neighbors(p),
               ∀p ∈ Π, ∀i ∈ {1, 2, ...}

Code for process p:
  procedure Consensus(v_p)
    message ← []
    k ← 1
    for q ∈ {p} ∪ neighbors(p) do
      message[q] ← ⟨q, RVals[q, k].propose(v_p)⟩
    while true
      send (R, k, message) to all
      wait for messages of the form (R, k, ∗) representing more than
          n/2 processes
      if received more than n/2 tuples with different ids and
          and the same value v
      then for q ∈ {p} ∪ neighbors(p) do
          message[q] ← ⟨q, PVals[q, k].propose(v)⟩
      else for q ∈ {p} ∪ neighbors(p) do
          message[q] ← ⟨q, PVals[q, k].propose(?)⟩

      send (P, k, message) to all
      wait for messages of the form (P, k, ∗) representing more than
          n/2 processes
      if received more than n/2 tuples with different ids and
          the same value v ≠ ?
      then decide(v)
      k ← k + 1
      if at least one tuple has value v ≠ ?
      then for q ∈ {p} ∪ neighbors(p) do
            message[q] ← ⟨q, RVals[q, k].propose(v)⟩
      else for q ∈ {p} ∪ neighbors(p) do
            v ← 0 or 1 randomly
            message[q] ← ⟨q, RVals[q, k].propose(v)⟩
```

Figure 2: Hybrid Ben-Or (HBO) consensus algorithm.

We call this algorithm the *Hybrid Ben-Or* or *HBO* algorithm. Figure 2 shows the pseudocode. There, processes do not terminate after deciding, but it is easy to modify the algorithm so that they do. This algorithm always satisfies the safety properties of consensus, irrespective of the number of crash failures:

Theorem 4.1. *The HBO algorithm in Figure 2 satisfies the Validity and Uniform Agreement properties of consensus in the m&m model with reliable links.*

The proof of this theorem is given in the full paper. There, we also show that the HBO algorithm terminates as long as a majority of the processes are *represented*.

Theorem 4.2. *The HBO algorithm in Figure 2 satisfies the Termination property of consensus with probability 1 in the m&m model with reliable links where a majority of the processes are represented.*

In the next section, we consider how many failures may occur while still ensuring that a majority of processes are represented.

4.2 Shared-memory expanders

In this section, we consider the fault tolerance of the HBO algorithm: how many crash failures can it tolerate while ensuring that

processes decide. In the algorithm, correct processes represent their neighbors in G_{SM}, so the fault tolerance depends on G_{SM} and how many neighbors correct processes have. We show that, by choosing G_{SM} to be a graph with *high expansion*, we obtain the best trade-off between maintaining low degree and achieving high fault tolerance. Having a low degree is important because the degree indicates the number of connections that a process must have to establish a shared memory, and that number is limited by the hardware (§3).

Roughly, expander graphs are graphs with the property that every sufficiently small set of vertices has many neighbors. To define these graphs more precisely, we follow the survey by Hoory, Linial and Wigderson [37]; we refer the reader to this survey for a detailed treatment.

Definition 1. *Let $G = (V, E)$ be an undirected graph.*

1. *The* vertex boundary *of a set $S \subseteq V$ is*

$$\delta S = \{\, u \in V : \{u, v\} \in E, v \in S \,\} \setminus S.$$

2. *The* vertex expansion ratio *of G, denoted $h(G)$, is defined as*

$$h(G) = \min_{S \subseteq V : |S| \leq |V|/2} \frac{|\delta S|}{|S|}$$

Intuitively, the higher the vertex expansion ratio, the larger the vertex boundary and the better connected the graph.

To apply this definition to the fault tolerance of HBO, consider a system with shared-memory graph G_{SM} and vertex expansion ratio $h(G_{SM})$, where up to f processes may crash. The set of vertices of interest to us is the set C of correct processes. If C has many neighbors, then it can simulate many extra processes in HBO. The adversary may pick *any* set of at least $n - f$ processes to be correct; regardless of the set it picks, that set has a vertex boundary of at least $(n-f) \cdot h(G_{SM})$.

The fault tolerance of HBO improves with the vertex expansion ratio of the graph. This is made precise by the following:

Theorem 4.3. *Consider the m&m model with shared-memory graph G_{SM} where links are reliable and f processes may crash. The HBO algorithm in Figure 2 satisfies the Termination property of consensus with probability 1 if $f < (1 - \frac{1}{2(1+h(G_{SM}))}) \cdot n$.*

Proof. Recall that Ben-Or's algorithm requires that $f < n/2$ for termination. The HBO algorithm simulates the correct processes and the processes in their vertex boundary. Given G_{SM} has vertex expansion ratio $h(G_{SM})$, the number of processes simulated by the algorithm is at least $(n-f) \cdot (1+h(G_{SM}))$. Rearranging the terms leads to the theorem. □

In the full paper, we give an example of graphs we can use, by discussing a construction of a family of expander graphs and showing their expansion ratio $h(G)$.

4.3 Impossibility result

We now show an impossibility result about the fault tolerance of consensus in the m&m model. The impossibility depends on the topology of the shared-memory graph G_{SM}. Intuitively, the impossibility indicates that the expander construction is the correct approach: we show that the fault tolerance of the system is related to the minimum cut that separates a large subgraph from the rest of the G_{SM} graph. In graphs with high expansion, the size of such a cut it guaranteed to be large, and thus many failures can be tolerated.

To establish the impossibility, we extend the well-known *partitioning argument* [59] to the m&m model. Basically, if two processes cannot communicate during the execution of an algorithm, then they cannot decide the same value. Thus, if the adversary can partition the system into two disjoint subgraphs A and B, each of size $\geq n - f$, where processes in A do not communicate with processes in B, then agreement cannot hold. This argument works in message-passing models, where the adversary can arbitrarily delay messages on the network, but it breaks in a shared-memory model, in which communication between processes cannot be delayed without blocking the processes themselves. Thus, in the m&m model, to create such a partition the adversary must get rid of all shared-memory edges of G_{SM} on the cut between A and B.

We now formalize the intuition to arrive at the impossibility. Given a graph $G = (V, E)$, we say that $C = (B, S, T)$ is an *SM-cut* in G if B, S, and T are disjoint subsets of V such that $B \cup S \cup T = V$, and there is a way to partition B into two disjoint subsets B_1 and B_2 such that $(B_1 \cup S, B_2 \cup T)$ is a cut of the graph G, and for every $b_1 \in B_1$, $b_2 \in B_2$, $s \in S$ and $t \in T$, $\{s, t\} \notin E$, $\{b_1, t\} \notin E$, and $\{b_2, s\} \notin E$. Intuitively, B is the set of vertices on the boundary of the cut, and S and T are the remaining vertices on each side.

Theorem 4.4. *Consider the m&m model with shared-memory graph G_{SM}, where links are reliable and f processes may crash. Consensus cannot be solved if G_{SM} has an SM-cut (B, S, T) with $|S| \geq n-f$ and $|T| \geq n-f$.*

We prove this theorem in the full paper. Note that, in a graph with high expansion, there are no *SM-cuts* (B, S, T) with $|S| \geq n-f$ and $|T| \geq n-f$. Intuitively, this is because if we want to build an *SM-cut* and we start with some set S with $|S| \geq n - f$, we must include δS in B_1, and then include $\delta(S \cup B_1)$ in B_2. As these sets expand quickly, we are then left with fewer than $n - f$ vertices to put in T. In the full paper, we formalize the above intuition to relate the impossibility to the expansion properties of G_{SM}.

5 LEADER ELECTION

We now show that the m&m model allows us to not only improve the fault tolerance of message-passing systems, but also to reduce the synchrony needed to solve certain problems. To demonstrate that, we turn our attention to the (eventual) leader election problem. In this problem, each process has a leader, and the goal is for all correct processes to eventually have the *same* correct leader (this is also known as the Ω failure detector [19]). Leader election is used in several well-known consensus algorithms, such as Paxos [52], Raft [63], and CT [20]. To be solvable, leader election requires the system to have some partial synchrony (because it can be used to solve consensus, and consensus is impossible in asynchronous systems [32]). Finding the weakest models of synchrony for solving this problem was the goal of several papers, but all known leader election algorithms for message-passing systems require some synchrony on at least some of the network communication links. In practice, it can be hard to guarantee small bounds on network delays, thus leading to high recovery time when a leader crashes.

We show that the m&m model permits solutions with almost no synchrony: the only requirement is that some process be timely. We

give two algorithms: one assumes reliable links (§5.1), and the other relaxes that requirement and assumes only fair lossy links (§5.2). In both algorithms, the leader regularly increments a heartbeat counter in shared memory, and other processes verify that the leader is alive by monitoring and timing out on this counter. With fair lossy links, the leader in addition periodically reads a register. We can make it easier for the leader to be timely, by placing the shared registers so that eventually the leader accesses only local registers (§5.3). We show that our algorithms are tight in efficiency (§5.4). In this section, we assume that G_{SM} is the complete graph.

5.1 Algorithm for reliable links

The basic idea of the first algorithm is that each process p has a "badness" counter that it shares with other processes. Intuitively, this badness counter represents the number of times that other processes suspected p of having crashed. To pick its leader, p keeps a set of processes, called *contenders*, that are contending for leadership; this set always includes p and initially contains no other processes. If there are no other contenders, p picks itself as the leader; otherwise, it picks the process with the smallest badness counter. Note that at this point, different processes could pick different leaders. We show that our algorithm always eventually realizes and corrects such situations.

When p becomes its own leader, it announces its leadership to other processes using a notification mechanism; here, this mechanism simply sends a message to the other processes (in our next algorithm, the mechanism is more complex because messages can be lost). If p thinks it is the leader, it sets an *active* bit in shared memory to indicate that it believes itself to be the leader. Then, it periodically increments a heartbeat counter in shared memory to tell others that it is alive. Process p also periodically checks whether it got any notifications from another process q; if it did, this means that q also wants to be the leader. So, p adds q to the contenders set, and p starts a timer on q; in effect, p now monitors q to see if it remains timely. Lastly, upon adding q to its contenders set, p also notifies q that p is also a contender, in case q does not know.

Whether p is a leader or not, p monitors the processes in its contender set other than itself. Intuitively, p expects each contender $q \neq p$ to periodically increment q's heartbeat counters in shared memory. If q fails to do so within a timeout,[4] p removes q from its contenders set; p then checks whether q has the active bit set in shared memory; if it does, that means q thinks it is the leader, so p sends an accusation to q and increments the timeout value. While p is its own leader, it also checks whether it received any accusation messages. If p receives an accusation, it increments its badness counter. If p stops thinking it is the leader, it clears the active bit in shared memory. The active bit is critical for correctness; it prevents processes from sending an accusation to p after it relinquishes leadership and stops incrementing its heartbeat.

Processes who believe themselves to be the leader fight among themselves for leadership: as described above, they notify each other about their desire to be the leader, adding each other to their contender set, and they pick the contending process with the smallest badness counter as the winning leader. We now explain how all correct processes eventually choose the same leader forever.

First note that every timely process eventually stops being accused (because it increases its heartbeat in a timely fashion when it thinks it is the leader, and it clears its active bit when it thinks it is not the leader). By assumption, the system has at least one timely process, thus there is at least one correct process that stops receiving accusations, and so its badness counter stops growing. Let ℓ be such a correct process whose badness counter is smallest. Therefore, for every process $q \neq \ell$: either q has a badness counter that eventually grows larger than ℓ's badness counter, or q crashes (and so every correct process eventually times out on q and removes q from its contender set). Since the contender set of ℓ contains ℓ forever, it is clear that ℓ eventually selects itself as the leader forever. When ℓ becomes leader, it notifies all the correct processes. Eventually other processes stop thinking they are leader: if a process $q \neq \ell$ thought it were leader infinitely often, it would check notifications infinitely often and eventually get a notification from ℓ, add ℓ to its contender set, see that ℓ has a smaller badness counter, and then pick ℓ rather than itself as its leader. This implies that every correct process eventually selects ℓ as a leader forever.

This algorithm performs little work in steady state, as eventually the following happens: (1) processes stop sending notifications because they do so only when they become a leader or when they are leader and receive a notification; (2) processes stop sending accusations, because eventually ℓ is their only contender (other than themselves), and no process times out on ℓ; (3) ℓ is the only process who writes to a shared register, because only a process who thinks it is the leader does so; (4) ℓ does not read any shared register, because processes stop sending notifications; (5) processes $p \neq \ell$ read only the shared register written by ℓ, because ℓ is their only contender. Also note that each shared register in this algorithm is written only by a single process (single-writer multi-reader shared register).

Figures 3 and 4 show the detailed algorithm. In the text below, when necessary for clarity, we use subscripts on a local variable to denote its process, and possibly superscripts to denote a time (e.g., $state_p$ is *state* variable of p, while $state_p^t$ is the value of this variable at time t).

Figure 4 shows the notification mechanism, which is very simple in this case (where links are reliable): the NOTIFY(q) procedure just sends a notification message to q, while the GET_NOTIFICATIONS() procedure returns the processes from which p got a notification message since the last invocation. Figure 3 has the main pseudocode. Each process p has a $STATE[p]$ shared register, which is a triple with p's heartbeat counter, badness counter, and active bit. Process p has a local variable $state_p[q]$ containing p's local view of $STATE[q]$. Process p executes forever in a loop. In this loop, p first picks its leader (line 9). Then p checks if it has just become the leader and, if so, notifies others (lines 10–11). Next p checks if it has just lost leadership and, if so, clears its active bit (lines 12–14). Next, if p is the leader (line 15), it increments its heartbeat counter and sets the active bit in shared memory (lines 16–18), and checks from whom it received notifications (line 19). For each such process q, it adds q to

[4] In the code, p checks the timer of *all* processes for expiration, but it is easy to see that only processes in p's contender set have an ongoing timer at p.

```
Shared objects:
1  STATE[p] ← (0, 0, false): register accessible by all, ∀p ∈ Π
                                        { fields (hb, counter, active) }
Variables of process p:
2  state[q] ← (0, 0, false), ∀q ∈ Π { local var with fields (hb, counter, active) }
3  hbtimeout[q] ← η + 1, ∀q ∈ Π \ {p};    hbtimer[q] ← off, ∀q ∈ Π \ {p}
4  notifiers ← ∅; competitors ← ∅
5  contenders ← {p}
6  leader ← ⊥
Code for each process p:
7  repeat forever
8    previous_leader ← leader
9    leader ← ℓ s.t.
       (state[ℓ].counter, ℓ) = min{(state[q].counter, q) : q ∈ contenders}
10   if previous_leader ≠ p and leader = p then        { if p becomes leader }
11     for each q ∈ Π \ {p} do Notify(q)                { p tells all others }
12   if previous_leader = p and leader ≠ p then    { if p loses its leadership }
13     state[p].active ← false                        { p indicates it is not active }
14     STATE[p] ← state[p]                { p updates its STATE shared variable }
15   if leader = p then                                          { if p is leader }
16     state[p].hb ← state[p].hb + 1                { p increments its heartbeat }
17     state[p].active ← true                          { p indicates it is active }
18     STATE[p] ← state[p]                { p updates its STATE shared variable }
19     competitors ← Get_Notifications() { p checks if others are leaders }
20     for each q ∈ competitors do    { for each q competing for leadership }
21       contenders ← contenders ∪ {q}                 { p adds q to contenders }
22       hbtimer(q) ← hbtimeout[q]                { p starts a timeout on q }
23       state[q] ← STATE[q]              { p reads q's STATE shared variable }
24       Notify(q)          { p tells q that p is also competing for leadership }
25     if received Accusation do                                  { if p accused }
26       state[p].counter ← state[p].counter + 1       { inc accusation count }
27       STATE[p] ← state[p]              { p updates its STATE shared variable }
28   for each q ∈ Π \ {p} do
29     if hbtimer(q) expired
30       previous_hb[q] ← state[q].hb     { p records q's previous heartbeat }
31       state[q] ← STATE[q]              { p reads q's STATE shared variable }
32       current_hb[q] ← state[q].hb      { p records q's current heartbeat }
33       if current_hb[q] > previous_hb[q] then     { if q's heartbeat grew }
34         hbtimer(q) ← hbtimeout[q]               { p restarts a timeout on q }
35       else
36         contenders ← contenders − {q}    { p removes q from contenders }
37         if state[q].active then                              { if q active }
38           send accusation to q                               { p accuses q }
39           hbtimeout[q] ← hbtimeout[q] + 1  { p increments q's timeout }
```

Figure 3: Leader election algorithm.

```
1  procedure Notify(q) { send notify to q }
2  procedure Get_Notifications() {
3      notifiers ← {q ∈ Π \ {p} : received new notify message from q }
4      return notifiers }
```

Figure 4: Notification mechanism (reliable links).

```
Shared objects:
1  NOTIFICATIONS[p] ← false          { register written by all, read by p }
2  for each q ∈ Π do NOTIFIES[p][q] ← false
                                     { register written by all, read by p }
Code for each process p:
3  procedure Notify(q)
4    NOTIFIES[q][p] ← true                { tell q that it was notified by p }
5    NOTIFICATIONS[q] ← true                { tell q that it was notified }
6    return
7  procedure Get_Notifications()
8    notifiers ← ∅
9    if NOTIFICATIONS[p] = true then do    { if someone notified p then }
10     NOTIFICATIONS[p] ← false          { p resets its notifications flag }
11     for each q ∈ Π \ {p} do                  { p checks who notified it }
12       if NOTIFIES[p][q] = true then do   { if p was notified by q then }
13         NOTIFIES[p][q] ← false { p resets the notification flag set by q }
14         notifiers ← notifiers ∪ {q}              { add q to its notifiers set }
15   return notifiers
```

Figure 5: Notification mechanism (fair lossy links).

contenders, starts a timer on q,[5] reads $STATE[q]$, and notifies q that it is also a leader contender (lines 20–24). Then p checks whether it received accusations and, if so, increments its badness counter (lines 25–27).

Next, whether p is leader or not, it checks whether the timer on each process $q \neq p$ has expired (lines 28–29). If it has, p checks whether q's heartbeat increased since p started the timer (lines 30–33). If so, p restarts the timer (line 34). Otherwise, p removes q from its contender set; if q is active, p sends an accusation to q and increments its timeout value (lines 35–39).

Theorem 5.1. *The algorithm in Figures 3 and 4 solves the eventual leader election problem in the m&m model where links are reliable and at least one process is timely. Eventually, no messages are sent, and the only accesses to shared memory is that the leader periodically writes a shared register and other processes periodically read this register.*

The proof is in the full paper.

5.2 Algorithm for fair lossy links

We now present an algorithm that works with fair lossy links. This algorithm has an added cost: the leader not only keeps writing a shared register, but also keeps reading a shared register. This cost is necessary, as we see later (§5.4).

The algorithm is similar to the previous one, except that it uses a different notification mechanism. Instead of sending a message, a process p notifies q by setting a bit in a $NOTIFIES[q][p]$ matrix in shared memory; p also sets a bit in a $NOTIFICATIONS[q]$ vector; this is an optimization so that q can monitor just one bit instead of a row of the matrix. More precisely, to know whether p has any notifications, p first examines the $NOTIFICATIONS[p]$ bit; if it is

[5] The timer is a local variable with a counter that is decremented at each step of p (this is not shown in the code for clarity), until it reaches 0 (it "expires").

clear, it need not check anything else; otherwise, it examines the row $NOTIFIES[p][-]$ of the matrix to find out which process sent the notification. Figure 5 has the detailed code. By combining it with in Figure 3, we obtain the leader election algorithm.

THEOREM 5.2. *The algorithm in Figures 3 and 5 solves the eventual leader election problem in the m&m model where links are fair lossy and at least one process is timely. Eventually, no messages are sent, and the only accesses to shared memory is that the leader periodically writes a shared register and periodically reads a shared register, and other processes periodically read a shared register.*

5.3 Locality

Our leader election algorithms are also efficient in terms of the *locality*, where each register is local to some process. This model corresponds well to the reality of an RDMA network, in which each process and each register are on some machine. In this case, if a process p is on the same machine as some register r, we say that r is *local* to p, and that p *owns* r. We say that the owner can read and write the register *locally*, while others read and write *remotely*.

Both leader election algorithms ensure that eventually the leader ℓ accesses registers only locally, by writing $STATE[\ell]$ (first algorithm), or by writing $STATE[\ell]$ and reading $NOTIFICATIONS[\ell]$ (second algorithm). This property further decreases the synchrony needed in practice: recall that we require only one process to be timely (the leader). By making the leader accesses local, the algorithms make it easier for the leader to be timely, since its steps involve only local accesses. On the other hand, the other processes access shared registers remotely, which is slower, but they are not required to be timely.

5.4 Impossibility result

In the full paper, we show that our algorithms are tight in the following sense. For systems with a timely process and asynchronous links, the leader must write shared registers forever. This result holds even if links are reliable and, a fortiori, if links are fair lossy. If, however, links are fair lossy, there is an additional requirement: either the leader writes and reads shared registers forever, or some process keeps sending messages forever.

THEOREM 5.3. *Let A be any eventual leader election algorithm for the m&m model with $n \geq 2$ processes, where links are reliable and at least one process is timely. There is a run of A whose leader writes shared registers infinitely often.*

THEOREM 5.4. *Let A be any eventual leader election algorithm for the m&m model with $n \geq 3$ processes, where links are fair lossy and at least one process is timely. Either (1) there is a run of A whose leader writes and reads shared registers infinitely often, or (2) there is a run of A in which some process sends messages infinitely often.*

6 CONCLUSION

The m&m model provides some inherent advantages over the pure message-passing and shared-memory models. In this paper, we demonstrated advantages in two aspects, fault tolerance and synchrony, and we focused on two problems, consensus and leader election. There are many exciting directions for future work in this space: discovering other benefits of the m&m model, developing better algorithms, studying other problems beyond consensus and leader election, evaluating algorithms in practice, and considering more failure models. On the last point, we addressed only process crashes, but it would be interesting to consider Byzantine failures, where the ability to pass messages and share registers selectively could overcome the difficulties of dealing with Byzantine processes in shared memory. Also interesting is to consider failures of the shared memory, especially if only parts of the memory fails [2, 42].

REFERENCES

[1] K. Abrahamson. On achieving consensus using a shared memory. In *ACM Symposium on Principles of Distributed Computing*, pages 291–302, Aug. 1988.

[2] Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6):1231–1274, Nov. 1995.

[3] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX Annual Technical Conference*, July 2018.

[4] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *ACM Symposium on Cloud Computing*, pages 121–127, Sept. 2017.

[5] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *ACM Symposium on Principles of Distributed Computing*, pages 328–337, July 2004.

[6] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):239–314, Oct. 2008.

[7] M. K. Aguilera and S. Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, Oct. 2012.

[8] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[9] K. Asanovic and D. Patterson. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Keynote of USENIX Conference on File and Storage Technologies*, Feb. 2014.

[10] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, Sept. 1990.

[11] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, Jan. 1995.

[12] H. Attiya and K. Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):20, Oct. 2008.

[13] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using RDMA. In *International Conference on Management of Data*, pages 1463–1475, May 2015.

[14] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *ACM Symposium on Operating Systems Principles*, pages 29–44, Oct. 2009.

[15] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *ACM Symposium on Principles of Distributed Computing*, pages 27–30, Aug. 1983.

[16] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, Mar. 1990.

[17] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97, Dec. 2013.

[18] S. Chandra, J. R. Larus, and A. Rogers. Where is time spent in message-passing and shared-memory programs? In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–73, Oct. 1994.

[19] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[20] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[21] C. Cooper, T. Radzik, N. Rivera, and T. Shiraga. Fast plurality consensus in regular expanders. In *International Symposium on Distributed Computing*, pages

13:1–13:16, Oct. 2017.

[22] A. Daglis, D. Ustiugov, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot. SABRes: Atomic object reads for in-memory rack-scale computing. In *International Symposium on Microarchitecture*, pages 1–13, Oct. 2016.

[23] R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *International Symposium on Distributed Computing*, pages 71–85, Oct. 2004.

[24] T. David, R. Guerraoui, and M. Yabandeh. Consensus inside. In *International Middleware Conference*, pages 145–156, Dec. 2014.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, Dec. 2004.

[26] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, Sept. 1965.

[27] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987.

[28] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation*, pages 401–414, Apr. 2014.

[29] A. Dragojević, D. Narayanan, E. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles*, pages 54–70, Oct. 2015.

[30] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[31] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems*, pages 17–17, May 2015.

[32] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[33] Gen-Z draft core specification—december 2016. http://genzconsortium.org/draft-core-specification-december-2016.

[34] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity ethernet at scale. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 202–215, Aug. 2016.

[35] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Workshop on Hot Topics in Networks*, pages 10:1–10:7, Nov. 2013.

[36] B. Holt, J. Nelson, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Flat combining synchronized global data structures. In *International Conference on PGAS Programming Models*, pages 76–92, Oct. 2013.

[37] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, Aug. 2006.

[38] M. Hutle, D. Malkhi, U. Schmid, and L. Zhou. Chasing the weakest system model for implementing Ω and consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4):269–281, Oct. 2009.

[39] InfiniBand. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband.

[40] Intel Omni-Path. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[41] iWARP. https://en.wikipedia.org/wiki/IWARP.

[42] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, May 1998.

[43] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 295–306, Aug. 2014.

[44] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Symposium on Operating Systems Design and Implementation*, pages 185–201, Nov. 2016.

[45] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *IEEE International Symposium on High Performance Distributed Computing*, pages 3–14, June 2015.

[46] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):673–685, July 2001.

[47] Y.-J. Kim and J. H. Anderson. Adaptive mutual exclusion with local spinning. In *International Symposium on Distributed Computing*, pages 29–43, Oct. 2000.

[48] Y.-J. Kim and J. H. Anderson. Timing-based mutual exclusion with local spinning. In *International Symposium on Distributed Computing*, pages 30–44, Oct. 2003.

[49] A. C. Klaiber and H. M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *International Symposium on Computer Architecture*, pages 94–105, Apr. 1994.

[50] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory: Early experience. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 54–63, 1993.

[51] L. Lamport. On interprocess communication part I–II. *Distributed Computing*, 1(2):77–101, May 1986.

[52] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[53] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, Dec. 2001.

[54] T. LeBlanc and E. Markatos. Shared memory vs. message passing in shared-memory multiprocessors. In *IEEE Symposium on Parallel and Distributed Processing*, pages 254 – 263, Dec. 1992.

[55] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture*, pages 267–278, June 2009.

[56] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *IEEE Symposium on High Performance Computer Architecture*, pages 189–200, Feb. 2012.

[57] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[58] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference*, pages 773–785, July 2017.

[59] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[60] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, June 2013.

[61] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX Annual Technical Conference*, pages 291–305, July 2015.

[62] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, Mar. 2014.

[63] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–320, June 2014.

[64] M. Poke and T. Hoefler. DARE: High-performance state machine replication on RDMA networks. In *IEEE International Symposium on High Performance Distributed Computing*, pages 107–118, June 2015.

[65] RDMA over converged ethernet. https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet.

[66] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Oct. 1996.

[67] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *Symposium on Operating Systems Design and Implementation*, pages 317–332, Nov. 2016.

[68] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle. DaRPC: Data center RPC. In *ACM Symposium on Cloud Computing*, pages 1–13, Nov. 2014.

[69] S.-Y. Tsai and Y. Zhang. LITE kernel RDMA support for datacenter applications. In *ACM Symposium on Operating Systems Principles*, pages 306–324, Oct. 2017.

[70] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and scalable PAXOS on RDMA. In *ACM Symposium on Cloud Computing*, pages 94–107, Sept. 2017.

[71] U. Wijetunge, S. Perreau, and A. Pollok. Distributed stochastic routing optimization using expander graph theory. In *Australian Communications Theory Workshop*, pages 124–129, Jan. 2011.

[72] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe. High performance RDMA protocols in HPC. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, pages 76–85, Sept. 2006.

[73] E. Zamanian, C. Binnig, T. Kraska, and T. Harris. The end of a myth: Distributed transactions can scale. *Proceedings of the VLDB Endowment*, 10(6):685–696, Feb. 2017.