Loose Synchronization for Large-Scale Networked Systems

Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat
University of California, San Diego
{*jalbrecht, ctuttle, snoeren, vahdat*}@*cs.ucsd.edu*

Abstract

Traditionally, synchronization barriers ensure that no cooperating process advances beyond a specified point until all processes have reached that point. In heterogeneous large-scale distributed computing environments, with unreliable network links and machines that may become overloaded and unresponsive, traditional barrier semantics are too strict to be effective for a range of emerging applications. In this paper, we explore several relaxations, and introduce a *partial barrier*, a synchronization primitive designed to enhance liveness in loosely coupled networked systems. Partial barriers are robust to variable network conditions; rather than attempting to hide the asynchrony inherent to wide-area settings, they enable appropriate application-level responses. We evaluate the improved performance of partial barriers by integrating them into three publicly available distributed applications running across PlanetLab. Further, we show how partial barriers simplify a re-implementation of MapReduce that targets wide-area environments.

1 Introduction

Today's resource-hungry applications are increasingly deployed across large-scale networked systems, where significant variations in processor performance, network connectivity, and node reliability are the norm. These installations lie in stark contrast to the tightly-coupled cluster and supercomputer environments traditionally employed by compute-intensive applications. What remains unchanged, however, is the need to synchronize various phases of computation across the participating nodes. The realities of this new environment place additional demands on synchronization mechanisms; while existing techniques provide correct operation, performance is often severely degraded. We show that new, relaxed synchronization semantics can provide significant performance improvements in wide-area distributed systems.

Synchronizing parallel, distributed computation has long been the focus of significant research effort. At a high level, the goal is to ensure that concurrent computation tasks—across independent threads, processors, nodes in a cluster, or spread across the Internet—are able to make independent forward progress while still maintaining some higher-level correctness semantic. Perhaps the simplest synchronization primitive is the barrier [19], which establishes a rendezvous point in the computation that all concurrent nodes must reach before any are allowed to continue. Bulk synchronous parallel programs running on MPPs or clusters of workstations employ barriers to perform computation and communication in phases, transitioning from one consistent view of shared underlying data structures to another.

In past work, barriers and other synchronization primitives have defined strict semantics that ensure safety—*i.e.,* that no node falls out of lock-step with the others—at the expense of liveness. In particular, if employed naively, a parallel computation using barrier synchronization moves forward at the pace of the slowest node and the entire computation must be aborted if any node fails. In closely-coupled supercomputer or cluster environments, skillful programmers optimize their applications to reduce these synchronization overheads by leveraging knowledge about the relative speed of individual nodes. Further, dataflow can be carefully crafted based upon an understanding of transfer times and access latencies to prevent competing demand for the I/O bus. Finally, recovering from failure is frequently expensive; thus, failure during computation is expected to be rare.

In large-scale networked systems—where individual node speeds are unknown and variable, communication topologies are unpredictable, and failure is commonplace—applications must be robust to a range of operating conditions. The performance tuning common in cluster and supercomputing environments is impractical, severely limiting the performance of synchronized concurrent applications. Further, individual node failures are almost inevitable, hence applications are generally engineered to adapt to or recover from failure. Our work focuses on distributed settings where online performance optimization is paramount and correctness is ensured through other mechanisms. We introduce adaptive mechanisms to control the degree of concurrency in cases where parallel execution appears to be degrading performance due to self-interference.

At a high level, emerging distributed applications all implement some form of traditional synchronizing barriers. While not necessarily executing a SIMD instruction stream, different instances of the distributed computation reach a shared point in the global computation. Relative to traditional barriers, however, a node arriving at a barrier need not necessarily block waiting for all other instances to arrive—doing so would likely sacrifice efficiency or even liveness as the system waits for either slow or failed nodes. Similarly, releasing a barrier does not necessarily imply that all nodes should pass through the barrier simultaneously—e.g., simultaneously releasing thousands of nodes to download a software package effectively mounts a denial-of-service attack against the target repository. Instead, applications manage the entry and release semantics of their logical barriers in an *ad hoc*, application-specific manner.

This paper defines, implements, and evaluates a new synchronization primitive that meets the requirements of a broad range of large-scale network services. In this context, we make the following contributions:

- We introduce a *partial barrier*, a synchronization primitive designed for heterogeneous failure-prone environments. By relaxing traditional semantics, partial barriers enhance liveness in the face of slow, failed, and self-interfering nodes.
- Based on the observation that the arrival rate at a barrier will often form a heavy-tailed distribution, we design a heuristic to dynamically detect the *knee of the curve*—the point at which arrivals slow considerably—allowing applications to continue despite slow nodes. We also adapt the rate of release from a barrier to prevent performance degradation in the face of self-interfering processes.
- We adapt three publicly available, wide-area services to use partial barriers for synchronization, and reimplement a fourth. In particular, we use partial barriers to reduce the implementation complexity of an Internet-scale version of MapReduce [9] running across PlanetLab. In all four cases, partial barriers result in a significant performance improvement.

2 Distributed barriers

We refer to nodes as *entering* a barrier when they reach a point in the computation that requires synchronization. When a barrier *releases* or *fires* (we use the terms interchangeably), the blocked nodes are allowed to proceed. According to strict barrier semantics, ensuring safety, *i.e.,* global synchronization, requires all nodes to reach a synchronization point before any node proceeds. In the face of wide variability in performance and prominent failures, however, strict enforcement may force the majority of nodes to block while waiting for a handful of slow or failed nodes to catch up. Many wide-area applications already have the ability to reconfigure themselves to tolerate node failure. We can harness this functionality to avoid excessive waits at barriers: once slow nodes are identified, they can be removed and possibly replaced by quicker nodes for the remainder of the execution.

One of the important questions, then, is determining when to release a barrier, even if all nodes have not arrived at the synchronization point. That is, it is important to dynamically determine the point where waiting for additional nodes to enter the barrier will cost the application more than the benefit brought by any additional arriving nodes in the future. This determination often depends on the semantics of individual applications. Even with full knowledge of application behavior, making the decision appropriately requires future knowledge. One contribution of this work is a technique to dynamically identify "knees" in the node arrival process, *i.e.,* points where the arrival process significantly slows. By communicating these points to the application we allow it to make informed decisions regarding when to release.

A primary contribution of this work is the definition of relaxed barrier semantics to support a variety of wide-area distributed computations. To motivate our proposed extensions consider the following application scenarios:

Application initialization. Many distributed applications require a startup phase to initialize their local state and to coordinate with remote participants before beginning their operation. Typically, developers introduce an artificial delay judged to be sufficient to allow the system to stabilize. If each node entered a barrier upon completing the initialization phase, developers would be freed from introducing arbitrarily chosen delays into the interactive development/debugging cycle.

Phased computation and communication. Scientific applications and large-scale computations often operate in phases consisting of one or many sequential, local computations, followed by periods of global communication or parallel computation. These computations naturally map to the barrier abstraction: one phase of the computation must complete before proceeding to the next phase. Other applications operate on a work queue that distributes tasks to available machines based on the rate that they complete individual tasks. Here, a barrier may serve as a natural point for distributing work.

Coordinated measurement. Many distributed applications measure network characteristics. However, uncoordinated measurements can self-interfere, leading to wasted effort and incorrect results. Such systems benefit from a mechanism that limits the number of nodes that simultaneously perform probes. Barriers are capable of

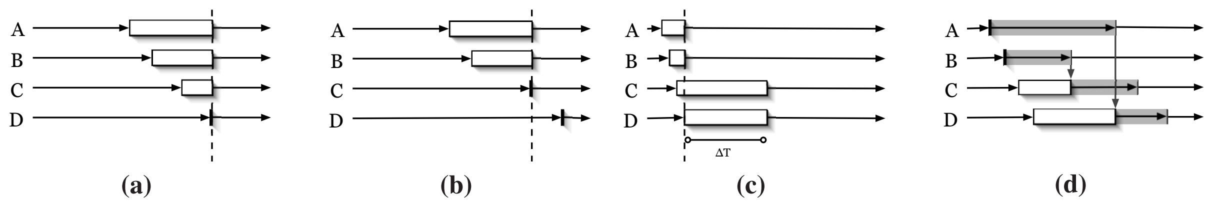

Figure 1: (a) Traditional semantics: All hosts enter the barrier (indicated by the white boxes) and are simultaneously released (indicated by dotted line). (b) Early entry: The barrier fires after 75% of the hosts arrive. (c) Throttled release: Hosts are released in pairs every ΔT seconds. (d) Counting semaphore: No more than 2 processes are simultaneously allowed into a "critical section" (indicated by the grey regions). When one node exits the critical section, another host is allowed to enter.

providing this needed functionality. In this case, the application uses two barriers to delimit a "critical section" of code (e.g., a network measurement). The first barrier releases some maximum number of nodes into the critical section at a time and waits until these nodes reach the second barrier, thereby exiting the critical section, before releasing the next round.

To further clarify the goal of our work, it is perhaps worthwhile to consider what we *are not* trying to accomplish. Partial barriers provide only a loose form of group membership [8, 16, 27]. In particular, partial barriers do not provide any ordering of messages relative to barriers as in virtual synchrony [2, 3], nor do partial barriers require that all participants come to consensus regarding a view change [24]. In effect, we strive to construct an abstraction that exposes the asynchrony and the failures prevalent in wide-area computing environments in a manner that allows the application to make dynamic and adaptive decisions as to how to respond.

It is also important to realize that not all applications will benefit from relaxed synchronization semantics. The correctness of certain classes of applications cannot be guaranteed without complete synchronization. For example, some applications may require a minimum number of hosts to complete a computation. Other applications may approximate a measurement (such as network delay) and continuing without all nodes reduces the accuracy of the result. However, many distributed applications, including the ones described in Section 5, can afford to sacrifice global synchronization without negatively impacting the results. These applications either support dynamically degrading the computation, or are robust to failures and can tolerate mid-computation reconfigurations. Our results indicate that for applications that are willing and able to sacrifice safety, our semantics have the potential to improve performance significantly.

3 Design and implementation

Partial barriers are a set of semantic extensions to the traditional barrier abstraction. Our implementation has a simple interface for customizing barrier functionality. This section describes these extended semantics, details our API, and presents the implementation details.

3.1 Design

We define two new barrier semantics to support emerging wide-area computing paradigms and discuss each below.

Early entry – Traditional barriers require all nodes to enter a barrier before any node may pass through, as in Figure 1(a). A partial barrier with early entry is instantiated with a timeout, a minimum percentage of entering nodes, or both. Once the barrier has met either of the specified preconditions, nodes that have already entered a barrier are allowed to pass through without waiting for the remaining slow nodes (Figure 1(b)). Alternatively, an application may instead choose to receive callbacks as nodes enter and manually release the barrier, enabling the evaluation of arbitrary predicates.

Throttled-release – Typically, a barrier releases all nodes simultaneously when a barrier's precondition is met. A partial barrier with throttled-release specifies a rate of release, such as two nodes every ΔT seconds as shown in Figure 1(c). A special variation of throttled-release barriers allows applications to limit the number of nodes that simultaneously exist in a "critical section" of code, creating an instance of a counting semaphore [10] (shown in Figure 1(d)), which may be used, for example, to throttle the number of nodes that simultaneously perform network measurements or software downloads. A critical distinction between traditional counting semaphores and partial barriers, however, is our support for failure. For instance, if a sufficient number of slow or soon-to-fail nodes pass a counting semaphore, they will limit access to other participants, possibly forever. Thus, as with early-entry barriers, throttled-release barriers eventually time out slow or failed nodes, allowing the system as a whole to make forward progress despite individual failures.

One issue that our proposed semantics introduce that does not arise with strict barrier semantics is handling nodes performing late entry, *i.e.*, arriving at an already released barrier. We support two options to address this

```
class Barrier {
  Barrier(string name, int max, int timeout,
          int percent, int minWait);
  static void setManager(string Hostname);
  void enter(string label, string Hostname);
  void setEnterCallback(bool (*callbackFunc)(string label,
          string Hostname, bool default), int timeout);
  map<string label, string Hostname> getHosts(void);
}
```

Figure 2: Barrier instantiation API.

case: i) pass-through semantics that allow the node to proceed with the next phase of the computation even though it arrived late; ii) catch-up semantics that issue an exception allowing the application to reintegrate the node into the mainline computation in an application-specific manner, which may involve skipping ahead to the next barrier (omitting the intervening section of code) in an effort to catch up to the other nodes.

3.2 Partial barrier API

Figure 2 summarizes the partial barrier API from an application's perspective. Each participating node initializes a barrier with a constructor that takes the following arguments: `name`, `max`, `timeout`, `percent`, and `minWait`. `name` is a globally unique identifier for the barrier. `max` specifies the maximum number of participants in the barrier. (While we do not require *a priori* knowledge of participant identity, it would be straightforward to add.) The `timeout` in milliseconds sets the maximum time that can pass from the point where the first node enters a barrier before the barrier is released. The `percent` field similarly captures a minimum percentage out of the maximum number nodes that must reach the barrier to activate early release. The `minWait` field is associated with the `percent` field and specifies a minimum amount of time to wait (even if the specified percentage of nodes have reached) before releasing the barrier with less than the maximum number of nodes. Without this field, the barrier will deterministically be released upon reaching the threshold percent of entering nodes even when all nodes are entering rapidly. However, the barrier is always released if `max` nodes arrive, regardless of `minWait`. The `timeout` field overrides the `percent` and `minWait` fields; the barrier will fire if the timeout is reached, regardless of the percentage of nodes entering the barrier. The last three parameters to the constructor are optional—if left unspecified the barrier will operate as a traditional synchronization barrier.

Coordination of barrier participants is controlled by a barrier manager whose identity is specified at startup by the `setManager()` method. Participants call the barrier's `enter()` method and pass in their `Hostname` and `label` when they reach the appropriate point in their execution. (The `label` argument supports advanced functionality such as load balancing for MapReduce as described in Section 5.4.) The participant's `enter()` method notifies the manager that the particular node has reached the synchronization point. Our implementation supports blocking calls to `enter()` (as described here) or optionally a callback-based mechanism where the entering node is free to perform other functionality until the appropriate callback is received.

While our standard API provides simplistic support for the early release of a barrier, an application may maintain its own state to determine when a particular barrier should fire and to manage any side effects associated with barrier entry or release. For instance, a barrier manager may wish to kill processes arriving late to a particular (already released) barrier. To support application-specific functionality, the `setEnterCallback()` method specifies a function to be called when any node enters a barrier. The callback takes the `label` and `Hostname` passed to the `enter()` method and a boolean variable that specifies whether the manager would normally release the barrier upon this entry. The callback function returns a boolean value to specify whether the barrier should actually be released or not, potentially overriding the manager's decision. A second argument to `setEnterCallback()` called `timeout` specifies a maximum amount of time that may pass before successive invocations of the callback. This prevents the situation where the application waits a potentially infinite amount of time for the next node to arrive before deciding to release the barrier. We use this callback API to implement our adaptive release techniques presented in Section 4.

Barrier participants may wish to learn the identity of all hosts that passed through a barrier, similar to (but with relaxed semantics from) view advancement or GBCAST in virtual synchrony [4]. The `getHosts()` method returns a map of `Hostnames` and `labels` through a remote procedure call with the barrier manager. If many hosts are interested in membership information, it can optionally be propagated from the barrier manager to all nodes by default as part of the barrier release operation.

Figure 3 describes a subclass of `Barrier`, called `ThrottleBarrier`, with *throttled-release* semantics. These semantics allow for a predetermined subset of the maximum number of nodes to be released at a specified rate. The methods `setThrottleReleasePercent()` and `setThrottleReleaseCount()` periodically release a percentage and number of nodes, respectively, once the barrier fires. `setThrottleReleaseTimeout()` specifies the periodicity of release.

```
class ThrottleBarrier extends Barrier {
  void setThrottleReleasePercent(int percent);
  void setThrottleReleaseCount(int count);
  void setThrottleReleaseTimeout(int timeout);
}
class SemaphoreBarrier extends Barrier {
  void setSemaphoreCount(int count);
  void setSemaphoreTimeout(int timeout);
  void release(string label, string Hostname);
  void setReleaseCallback(int (*callbackFunc)(string label,
          string Hostname, int default), int timeout);
}
```

Figure 3: ThrottleBarrier and SemaphoreBarrier API.

Figure 3 also details a variant of throttled-release barriers, `SemaphoreBarrier`, which specifies a maximum number of nodes that may simultaneously enter a critical section. A `SemaphoreBarrier` extends the throttled-release semantics further by placing a barrier at the beginning and end of a critical section of activity to ensure that only a specific number of nodes pass into the critical section simultaneously. One key difference for this type of barrier is that it does not require any minimum number of nodes to enter the barrier before beginning to release nodes into the subsequent critical section. It simply mandates a maximum number of nodes that may simultaneously enter the critical section. The `setSemaphoreCount()` method sets this maximum number. Nodes call the barrier's `release()` method upon completing the subsequent critical section, allowing the barrier to release additional nodes. `setSemaphoreTimeout()` allows for timing out nodes that enter the critical section but do not complete within a maximum amount of time. In this case, they are assumed to have failed, enabling the release of additional nodes. The `setReleaseCallback()` enables application-specific release policies and timeout of slow or failed nodes in the critical section. The callback function in `setReleaseCallback()` returns the number of hosts to be released.

3.3 Implementation

Partial barrier participants implement the interface described above while a separate barrier manager coordinates communication across nodes. Our implementation of partial barriers consists of approximately $3,000$ lines of C++ code. At a high level, a node calling `enter()` transmits a BARRIER_REACHED message using TCP to the manager with the calling host's unique identifier, barrier name, and label. The manager updates its local state for the barrier, including the set of nodes that have thus far entered the barrier, and performs appropriate callbacks as necessary. The manager starts a timer to support various release semantics if this is the first node entering the barrier and subsequently records the inter-arrival times between nodes entering the barrier.

If a sufficient number of nodes enter the barrier or a specified amount of time passes, the manager transmits FIRE messages using TCP to all nodes that have entered the barrier. For throttled release barriers, the manager releases the specified number of nodes from the barrier in FIFO order. The manager also sets a timer as specified by `setThrottleReleaseTimeout()` to release additional nodes from the barrier when appropriate.

For semaphore barriers, the manager releases the number of nodes specified by `setSemaphoreCount()` and, if specified by `setSemaphoreTimeout()`, also sets a timer to expire for each node entering the critical section. Each call to `enter()` transmits a SEMAPHORE_REACHED message to the manager. In response, the manager starts the timer associated with the calling node. If the semaphore timer associated with the node expires before receiving the corresponding SEMAPHORE_RELEASED message, the manager assumes that node has either failed or is proceeding sufficiently slowly that an additional node should be released into the critical section. Each SEMAPHORE_RELEASED message releases one additional node into the critical section.

For all barriers, the manager must gracefully handle nodes arriving late, *i.e.*, after the barrier has fired. We employ two techniques to address this case. For pass-through semantics, the manager transmits a LATE_FIRE message to the calling node, releasing it from the barrier. In catch-up semantics, the manager issues an exception and transmits a CATCH_UP message to the node. Catch-up semantics allow applications to respond to the exception and determine how to reintegrate the node into the computation in an application-specific manner.

The type of barrier—pass-through or catch-up—is specified at barrier creation time (intentionally not shown in Figure 2 for clarity). Nodes calling `enter()` may register local callbacks with the arrival of either LATE_FIRE or CATCH_UP messages for application-specific re-synchronization with the mainline computation, or perhaps to exit the local computation altogether if re-synchronization is not possible.

3.4 Fault tolerance

One concern with our centralized barrier manager is tolerating manager faults. We improve overall system robustness with support for replicated managers. Our algorithm is a variant of traditional primary/backup systems: each participant maintains an ordered list of barrier controllers. Any message sent from a client to the logical barrier manager is sent to all controllers on the list. Because application-specific entry callbacks may be non-deterministic, a straightforward replicated state machine approach where each barrier controller simultaneously decides when to fire is insufficient. Instead, the primary controller forwards all BARRIER_REACHED messages to the backup controllers. These messages act as implicit "keep alive" messages from the primary. If a backup controller receives BARRIER_REACHED messages from clients but not the primary for a sufficient period of time, the first backup determines the primary has failed and assumes the role of primary controller. The secondary backup takes over should the primary fail, and so on. Note that our approach admits the case where multiple controllers simultaneously act as the primary controller for a short period of time. Clients ignore duplicate FIRE messages for the same barrier, so progress is assured, and one controller eventually emerges as primary.

Although using the replicated manager scheme described above lowers the probability of losing BARRIER_REACHED messages, it does not provide any increased reliability with respect to messages sent from the manager(s) to the remote hosts. All messages are sent using reliable TCP connections. If a connection fails between the manager and a remote host, however, messages may be lost. For example, suppose the TCP connections between the manager and some subset of the remote hosts break just after the manager sends a FIRE message to all participants. Similarly, if a group of nodes fails after entering the barrier, but before receiving the FIRE message, the failure may go undetected until after the controller transmits the FIRE messages. In these cases, the manager will attempt to send FIRE messages to all participants, and detect the TCP failures after the connections time out. Such ambiguity is unavoidable in asynchronous systems; the manager simply informs the application of the failure(s) via a callback and lets the application decide the appropriate recovery action. As with any other failure, the application may choose to continue execution and ignore the failures, attempt to find new hosts to replace the failed ones, or to even abort the execution entirely.

4 Adaptive release

Unfortunately, our extended barrier semantics introduce additional parameters: the threshold for early release and the concurrency level in throttled release. Experience has shown it is often difficult to select values that are appropriate across heterogeneous and changing network conditions. Hence, we provide adaptive mechanisms to dynamically determine appropriate values.

4.1 Early release

There is a fundamental tradeoff in specifying an early-release threshold. If the threshold is too large, the application will wait unnecessarily for a relatively modest number of additional nodes to enter the barrier; if too small, the application will lose the opportunity to have participation from other nodes had it just waited a bit longer. Thus, we use the barrier's callback mechanism to determine release points in response to varying network conditions and node performance.

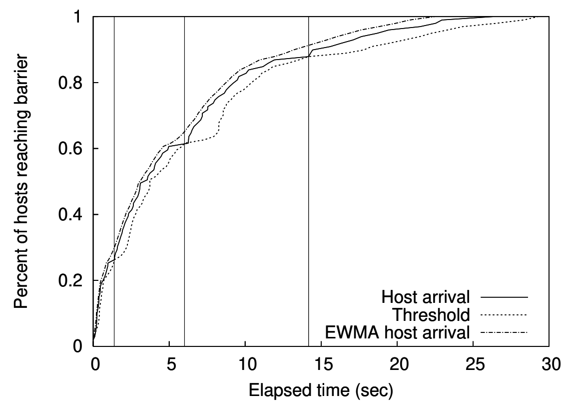

Figure 4: Dynamically determining the knee of arriving processes. Vertical bars indicate a knee detection.

In our experience, the distribution of nodes' arrivals at a barrier is often heavy-tailed: a relatively large portion of nodes arrive at the barrier quickly with a long tail of stragglers entering late. In this case, many of our target applications would wish to dynamically determine the "knee" of a particular arrival process and release the barrier upon reaching it. Unfortunately, while it can be straightforward to manually determine the knee off-line once all of the data for an arrival process is available, it is difficult to determine this point on-line.

Our heuristic, inspired by TCP retransmission timers and MONET [1], maintains an exponentially weighted moving average (EWMA) of the host arrival times (arr), and another EWMA of the deviation from this average for each measurement ($arrvar$). As each host arrives at the barrier, we record the arrival time of the host, as well as the deviation from the average. Then we recompute the EWMA for both arr and $arrvar$, and use the values to compute a maximum wait threshold of $arr + 4 * arrvar$. This threshold indicates the maximum time we are willing to wait for the next host to arrive before firing the barrier. If the next host does not arrive at the barrier before the maximum wait threshold passes, we assume that a knee has been reached. Figure 4 illustrates how these values interact for a simulated group of 100 hosts entering a barrier with randomly generated exponential interarrival times. Notice that a knee occurs each time the host arrival time intersects the threshold line.

With the capability to detect multiple knees, it is important to provide a way for applications to pick the right knee and avoid firing earlier or later than desired. Aggressive applications may choose to fire the barrier when the first knee is detected. Conservative applications may wish to wait until some specified amount of time has passed, or a minimum percentage of hosts have entered the barrier before firing. To support both aggressive and conservative applications, our algorithm allows the application to specify a minimum percentage of hosts, minimum waiting time, or both for each barrier. If an application specifies a minimum waiting time of 5 seconds, knees detected before 5 seconds are ignored. Similarly, if a minimum host percentage of 50% is specified, the knee detector ignores knees detected before 50% of the total hosts have entered the barrier. If both values are specified, the knee detector uses the more conservative threshold so that both requirements (time and host percentage) are met before firing.

One variation in our approach compared to other related approaches is the values for the weights in the moving averages. In the RFC for computing TCP retransmission timers [7], the weight in the EWMA of the *rtt* places a heavier weight (0.875) on previous delay measurements. This value works well for TCP since the average delay is expected to remain relatively constant over time. In our case, however, we expect the average arrival time to increase, and thus we decreased the weight to be 0.70 for previous measurements of arr. This allows our arr value to more closely follow the actual data being recorded. When measuring the average deviation, which is computed by averaging $|sample - arr|$ (where $sample$ represents the latest arrival time recorded), we used a weight of 0.75 for previous measurements, which is the same weight used in TCP for the variation in *rtt*.

4.2 Throttled release

We also employ an adaptive method to dynamically adjust the amount of concurrency in the "critical section" of a semaphore barrier. In many applications, it is impractical to select a single value which performs well under all conditions. Similar in spirit to SEDA's thread-pool controller [35], our adaptive release algorithm selects an appropriate concurrency level based upon recent release times. The algorithm starts with low level of concurrency and increases the degree of concurrency until response times worsen; it then backs off and repeats, oscillating about the optimum value.

Mathematically, the algorithm compares the median of the distributions of recent and overall release times. For example, if there are 15 hosts in the critical section when the 45th host is released, the algorithm computes the median release time of the last 15 releases, and of all 45. If the latest median is more than 50% greater than the overall median, no additional hosts are released, thus reducing the level of concurrency to 14 hosts. If the latest median is more than 10% but less than 50% greater than the overall median, one host is released, maintaining a level of concurrency of 15. In all other cases, two hosts are released, increasing the concurrency level to 16. The thresholds and differences in size are selected to increase the degree of concurrency whenever possible, but keep the magnitude of each change small.

5 Applications

We integrated partial barriers into three wide-area, distributed applications with a range of synchronization requirements. We reimplemented a fourth application whose source code was unavailable. While a detailed discussion of these applications is beyond the scope of this paper, we present a brief overview to facilitate understanding of our performance evaluation in Section 6.

5.1 Plush

Plush [29] is a tool for configuring and managing large-sale distributed applications, such as PlanetLab [28] and the Grid [13]. Users specify a description of: i) a set of nodes to run their application on; ii) the set of software packages, including the application itself and any necessary data files, that should be installed on each node; and iii) a directed acyclic graph of individual processes that should be run, in order, on each node.

Plush may be used to manage long-running services or interactive experimental evaluations of distributed applications. In the latter case, developers typically configure a set of machines (perhaps installing the latest version of the application binary) and then start the processes at approximately the same time on all participating nodes. A barrier may be naturally inserted between the tasks of node configuration and process startup. However, in the heterogeneous PlanetLab environment, the time to configure a set of nodes with the requisite software can vary widely or fail entirely at individual hosts. In this case, it is often beneficial to timeout the "software configuration/install" barrier and either proceed with the available nodes or to recruit additional nodes.

5.2 Bullet

Bullet [21] is an overlay-based large-file distribution infrastructure. In Bullet, a source transmits a file to receivers spread across the Internet. As part of the bootstrap process, all receivers join the overlay by initially contacting the source before settling on their final position in the overlay. The published quantitative evaluation of Bullet presents a number of experiments across PlanetLab. However, to make performance results experimentally meaningful when measuring behavior across a large number of receivers, the authors hard-coded a 30-second delay at the sender from the time that it starts to the time that it begins data transmission. While typically sufficient for the particular targeted configuration, the timeout would often be too long, unnecessarily extending turnaround time for experimentation and interactive debugging. Depending on overall system load and number of participants, the timeout would also sometimes be too short, meaning that some nodes would not complete the join process at the time the sender begins transmitting. While this latter case is not a problem for the correct behavior of the system, it makes interpreting experimental results difficult.

To address this limitation, we integrated partial barriers into the Bullet implementation. This integration was straightforward. Once the join process completes, each node simply enters a barrier. The Bullet sender registers for a callback with the barrier manager to be notified of a barrier release, at which point it begins transmitting data. By calling the `getHosts()` method, the sender can record the identities of nodes that should be considered in interpreting the experimental results. The barrier manager notes the identity of nodes entering the barrier late and instructs them to exit rather than proceed with retrieving the file.

5.3 EMAN

EMAN [12] is a publicly available software package used for reconstructing 3D models of particles using 2D electron micrographs. The program takes a 2D micrograph image as input and then repeatedly runs a "refinement" process on the data to create a 3D model. Each iteration of the refinement consists of both computationally inexpensive sequential computations and computationally expensive parallel computations.

Barriers separate sequential and parallel stages of computation in EMAN. Using partial barrier semantics adds the benefit of being able to detect slow nodes, allowing the application to redistribute tasks to faster machines during the parallel phase of computation. In addition to slow processors, the knee detector also detects machines with low bandwidth capacities and reallocates their work to machines with higher bandwidth. In our test, each refinement phase requires approximately 240 MB of data to be transferred to each node, and machines with low bandwidth links have a significant impact on the overall completion time if their work is not reallocated to different machines. Since the sequential phases run on a single machine, partial barriers are most applicable to the parallel phases of computation. We use the publicly available version of EMAN for our experiments. We wrote a 50-line Perl script to run the parallel phase of computation on 98 PlanetLab machines using Plush.

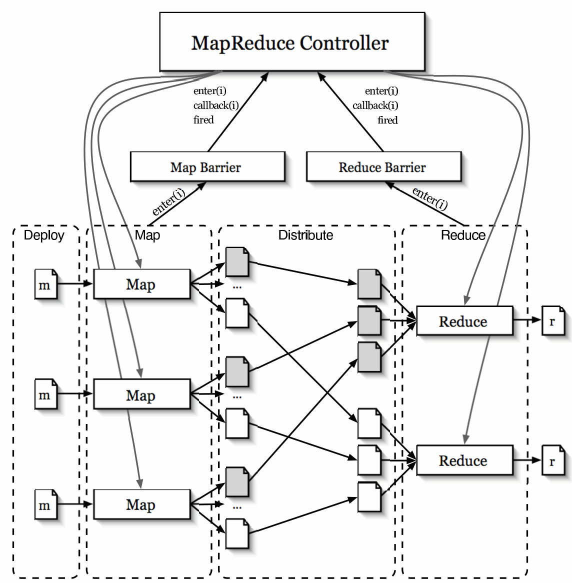

Figure 5: MapReduce execution. As each map task completes, `enter(i)` is called with the unique task identifier. Once all m tasks have entered the Map barrier, it is released. When the MapReduce controller is notified that the Map barrier has fired, the r reduce tasks are distributed and begin execution. When all r reduce tasks have entered the Reduce barrier, MapReduce is complete. In both barriers, `callback(i)` informs the MapReduce controller of task completions.

5.4 MapReduce

MapReduce [9] is a toolkit for application-specific parallel processing of large volumes of data. The model involves partitioning the input data into smaller *splits* of data, and spreading them across a cluster of worker nodes. Each worker node applies a *map* function to the splits of data that they receive, producing intermediate key/value pairs that are periodically written to specific locations on disk. The MapReduce master node tracks these locations, and eventually notifies another set of worker nodes that intermediate data is ready. This second set of workers aggregate the data and pass it to the *reduce* function. This function processes the data to produce a final output file.

Our implementation of MapReduce leverages partial barriers to manage phases of the computation and to orchestrate the flow of data among nodes across the wide area. In our design, we have m map tasks and corresponding input files, n total nodes hosting the computation, and r reduce tasks. A central MapReduce controller distributes the m split input files to a set of available nodes (our current implementation runs on PlanetLab), and spawns the map process on each node. When the map tasks finish, intermediate files are written back to a central repository, and then redistributed to r hosts, who eventually execute the r reduce tasks. There are a number of natural barriers in this application, as shown in Figure 5, corresponding to completion of: i) the initial distribution of m split files to appropriate nodes; ii) executing m map functions; iii) the redistribution of the intermediate files to appropriate nodes, and iv) executing r reduce functions.

As with the original MapReduce work, the load balancing aspects corresponding to barriers (ii) and (iv) (from the previous paragraph) are of particular interest. Recall that although there are m map tasks, the same physical host may execute multiple map tasks. In these cases, the goal is not necessarily to wait for all n hosts to reach the barrier, but for all m or r logical tasks to complete. Thus, we extended the original barrier entry semantics described in Section 3 to support synchronizing barriers at the level of a set of logical, uniquely named tasks or processes, rather than a set of physical hosts. To support this, we simply invoke the `enter()` method of the barrier API (see Figure 2) upon completing a particular map or reduce function. In addition to the physical hostname, we send a label corresponding to a globally unique name for the particular map or reduce task. Thus, rather than waiting for n hosts, the barrier instead waits for m or r unique labels to enter the barrier before firing.

For a sufficiently large and heterogeneous distributed system, performance at individual nodes varies widely. Such variability often results in a heavy-tailed distribution for the completion of individual tasks, meaning that while most tasks will complete quickly, the overall time will be dominated by the performance of the slow nodes. The original MapReduce work noted that one of the common problems experienced during execution is the presence of "straggler" nodes that take an unusually long time to complete a map or reduce task. Although the authors mentioned an application-specific solution to this problem, by using partial barriers in our implementation we were able to provide a more general solution that achieved the same results. We use the arrival rate of map/reduce tasks at their respective barriers to respawn a subset of the tasks that were proceeding slowly.

By using the knee detector described in Section 4, we are able to dynamically determine the transition point between rapid arrivals and the long tail of stragglers. However, rather than releasing the barrier at this point, the MapReduce controller receives a callback from the barrier manager, and instead performs load rebalancing functionality by spawning additional copies of outstanding tasks on nodes disjoint from the ones hosting the slower tasks (potentially first distributing the necessary input/intermediate files). As in the original implementation of MapReduce, the barrier is not concerned with what copies of the computation complete first; the goal is for all m or r tasks to complete as quickly as possible.

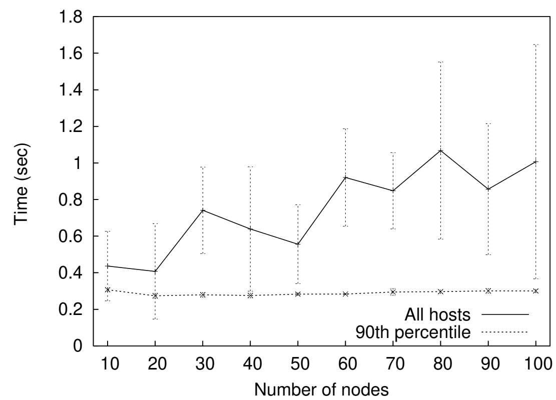

Figure 6: Scalability of centralized barrier implementation. "All hosts" line shows the average time across 5 runs for barrier manager to receive BARRIER_REACHED messages from all hosts. "90th percentile" line shows the average time across 5 runs for barrier manager to receive BARRIER_REACHED messages from 90% of all hosts.

The completion rate of tasks also provides an estimate of the throughput available at individual nodes, influencing future task placement decisions. Nodes with high throughput are assigned more tasks in the future. Our performance evaluation in Section 6 quantifies the benefits of this approach. Cluster-based MapReduce [9] also found this rebalancing to be critical for improving performance in more tightly controlled cluster settings, but did not describe a precise approach for determining when to spawn additional instances of a given computation.

6 Evaluation

The goal of this section is to quantify the utility of partial barriers for a range of application scenarios. For all experiments, we randomly chose subsets from a pool of 130 responsive PlanetLab nodes.

6.1 Scalability

To estimate baseline barrier scalability, we measure the time it takes to move between two barriers for an increasing number of hosts. In this experiment, the controller waits for all hosts to reach the first barrier. All hosts are released, and then immediately enter the second barrier. We measure the time between when the barrier manager sends the first FIRE message for the first barrier and receives the last BARRIER_REACHED message for the second barrier. No partial barrier semantics are used for these measurements. Figure 6 shows the average completion time for varying numbers of nodes across a total of five runs for each data point with standard deviation.

Notice that even for 100 nodes, the average time for the barrier manager to receive the last BARRIER_REACHED message for the second barrier is approximately 1 second. The large standard deviation values indicate that there is much variability in our results. This is due to the presence of straggler nodes that delay the firing for several seconds or more. The 90th percentile, on the other hand, has little variation and is relatively constant as the number of participants increases. This augurs well for the potential of partial barrier semantics to improve performance in the wide area. Overall, we are satisfied with the performance of our centralized barriers for 100 nodes; we expect to use hierarchy to scale significantly further.

6.2 Admission control

Next, we consider the benefits of a semaphore barrier to perform admission control for parallel software installation in Plush. Plush configures a set of wide-area hosts to execute a particular application. This process often involves installing the same software packages located on a central server. Simultaneously performing this install across hundreds of nodes can lead to thrashing at the server hosting the packages. The overall goal is to ensure sufficient parallelism such that the server is saturated (without thrashing) while balancing the average time to complete the download across all participants.

For our results, we use Plush to install the same 10-MB file on 100 PlanetLab hosts while varying the number of simultaneous downloads using a semaphore barrier. Figure 7 shows the results of this experiment. The data indicates that limiting parallelism can improve overall completion rate. Releasing too few hosts does not fully consume server resources, while releasing too many taxes available resources, increasing the time to first completion. This is evident in the graph since 25 simultaneous downloads finishes more quickly than both 100 and 10 simultaneous transfers.

While statically defining the number of hosts allowed to perform simultaneous downloads works well for our file transfer experiment, varying network conditions means that statically picking any single value is unlikely to perform well under all conditions. Some applications may benefit from a more dynamic throttled release technique that attempts to find the optimal number of hosts that maximizes throughput from the server without causing saturation. The "Adaptive Simultaneous Transfers" line in Figure 7 shows the performance of our adaptive release technique. In this example, the initial concurrency level is 15, and the level varies according to the duration of each transfer. In this experiment the adaptive algorithm line reaches 100% before the lines representing a fixed concurrency level of 10 or 100, but the algorithm was too conservative to match the optimal static level of 25 given the network conditions at the time.

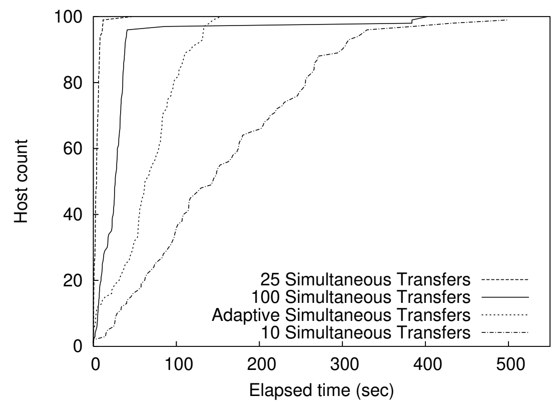

Figure 7: Software transfer from a high-speed server to PlanetLab hosts using a SemaphoreBarrier to limit the number of simultaneous file transfers.

6.3 Detecting knees

In this section we consider our ability to dynamically detect knees in a heavy-tailed arrival processes. We observe that when choosing a substantial number of time-shared PlanetLab hosts to perform the same amount of work, the completion time varies widely, often following an exponential distribution. This variation makes it difficult to coordinate distributed computation. To quantify our ability to more adaptively set timeouts and coordinate the behavior of multiple wide-area nodes, we used a barrier to synchronize senders and receivers in Bullet while running across a varying number of PlanetLab nodes. We set the barrier to dynamically detect the knee in the arrival process as described in Section 4. Upon reaching the knee, nodes already in the barrier are released; one side effect is that the sender begins data transmission. Bullet ignores all late arriving nodes.

Figure 8 plots the cumulative distribution of receivers that enter the startup barrier on the y-axis as a function of time progressing on the x-axis. Each curve corresponds to an experiment with 50, 90, or 130 PlanetLab receivers in the initial target set. The goal is to run with as many receivers as possible from the given initial set without waiting an undue amount of time for a small number of stragglers to complete startup. Interestingly, it is insufficient to filter for any static set of known "slow" nodes as performance tends to vary on fairly small time scales and can be influenced by multiple factors at a particular node (including available CPU, memory, and changing network conditions). Thus, manually choosing an appro-

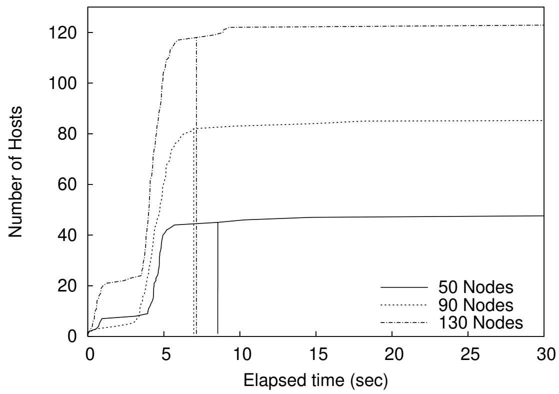

Figure 8: A startup barrier that regulates nodes joining a large-scale overlay. Vertical bars indicate when the barrier detects a knee and fires.

priate static set may be sufficient for one particular batch of runs but not likely the next.

Vertical lines in Figure 8 indicate where the barrier manager detects a knee and releases the barrier. Although we ran the experiment multiple times, for clarity we plot the results from a single run. While differences in time of day or initial node characteristics affect the quantitative behavior, the general shape of the curve is maintained. However, in all of our experiments, we are satisfied with our ability to dynamically determine the knee of the arrival process. The experiments are typically able to proceed with 85-90% of the initial set participating.

6.4 EMAN

For our next evaluation, we added a partial barrier to the parallel computation of EMAN's refinement process. Upon detecting a knee, we reallocate tasks to faster machines. Figure 9 shows the results of running EMAN with and without partial barrier semantics. In this experiment we ran EMAN on the 98 most responsive PlanetLab machines. The workflow consists of several serial tasks (not shown), and a 98-way image classification step, run in parallel. Each participant first downloads a 40-MB archive containing the EMAN executables and a wrapper script. After unpacking the archive, each node downloads a unique 200-MB data file and begins running the classification process. At the end of the computation, each node generates 77 output files stored on the local disk, which EMAN merges into 77 "master" files once all tasks complete.

After 801 seconds, the barrier detects a knee and reallocates the remaining tasks to idle machines among those initially allocated to the experiment; these finish shortly afterward. The "knee" in the curve at approximately 300 seconds indicates that around 21 hosts have good connectivity to the data repository, while the rest have longer

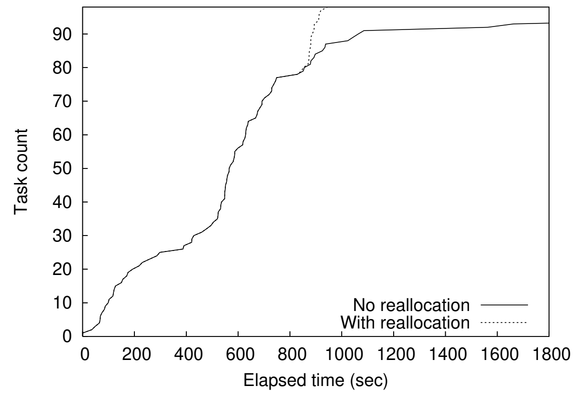

Figure 9: EMAN. Knee detected at 801 seconds. Total runtime is over 2700 seconds.

transfer times. While the knee detection algorithm detects a knee at 300 seconds, a minimum threshold of 60% prevents reconfiguration. The second knee is detected at 801 seconds, which starts a reconfiguration of 20 tasks. These tasks complete by 900 seconds, while the original set continue past 2700 seconds, for an overall speedup of more than 3.

6.5 MapReduce

We now consider an alternative use of partial barriers: to assist not with the synchronization of physical hosts or processors, but with load balancing of logical tasks spread across cooperating wide-area nodes. Further, we wish to determine whether we can dynamically detect knees in the arrival rate of individual tasks, spawning additional copies of tasks that appear to be proceeding slowly on a subset of nodes.

We conduct these experiments with our implementation of MapReduce (see Section 5.4) with $m = 480$ map tasks and $r = 30$ reduce tasks running across $n = 30$ PlanetLab hosts. During each of the map and reduce rounds, the MapReduce controller evenly partitions the tasks over the available hosts and starts the tasks asynchronously. For this experiment, each map task simply reads 2000 random words from a local file and counts the number of instances of certain words. This count is written to an intermediate output file based on the hash of the words. The task is CPU-bound and requires approximately 7 seconds to complete on an unloaded PlanetLab-class machine. The reduce tasks summarize these intermediate files with the same hash values.

Thus, each map and reduce task performs an approximately equal amount of work as in the original MapReduce work [9], though it would be useful to generalize to variable-length computation. When complete, a map or reduce task enters the associated barrier with a unique identifier for the completed work. The barrier manager

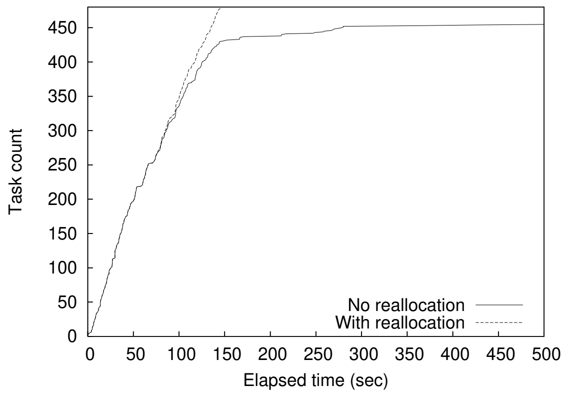

Figure 10: MapReduce: $m = 480, r = 30, n = 30$ with uniform prepartitioning of the data. Knee detection occurs at 68 seconds and callbacks enable rebalancing.

monitors the arrival rate and dynamically determines the knee, which is the where the completion rate begins to slow. We empirically determined that this slowing results from a handful of nodes that proceed substantially slower than the rest of the system. (Note that this phenomenon is not restricted to our wide-area environment; Dean and Ghemawat observed the same behavior for their runs on tightly coupled and more homogeneous clusters [9].) Thus, upon detecting the knee a callback to the MapReduce controller indicates that additional copies of the slow tasks should be respawned, ideally on nodes with the smallest number of outstanding tasks. Typically, by the time the knee is detected there are a number of hosts that have completed their initial allocation of work.

Figure 10 shows the performance of one MapReduce run both with and without task respawn upon detecting the knee. Figure 10 plots the cumulative number of completed tasks on the y-axis as a function of time progressing on the x-axis. We see that the load balancing enabled by barrier synchronization on abstract tasks is critical to overall system performance. With task respawn using knee detection, the barrier manager detects the knee approximately 68 seconds into the experiment after approximately 53% of the tasks have completed. At this point, the MapReduce controller (via a callback from the Barrier class) repartitions the remaining 47% of the tasks across available wide-area nodes. This is where the curves significantly diverge in the graph. Without dynamic rebalancing the completion rate transitions to a long-tail lasting more than 2500 seconds (though the graph only shows the first 500 seconds), while the completion rate largely maintains its initial slope when rebalancing is enabled. Overall, our barrier-based rebalancing results in a factor of sixteen speedup in completion time compared to proceeding with the initial mapping. Multiple additional runs showed similar results.

Note that this load balancing approach differs from the alternate approach of trying to predict the set of nodes likely to deliver the highest level of performance *a priori*. Unfortunately, predicting the performance of tasks with complex resource requirements on a shared computing infrastructure with dynamically varying CPU, network, and I/O characteristics is challenging in the best case and potentially intractable. We advocate a simpler approach that does not attempt to predict performance characteristics *a priori*. Rather, we simply choose nodes likely to perform well and empirically observe the utility of our decisions. Of course, this approach may only be appropriate for a particular class of distributed applications and comes at the cost of performing more work in absolute terms because certain computations are repeated. For the case depicted in Figure 10, approximately 30% of the work is repeated if we assume that the work on both the fast and slow nodes are run to completion (a pessimistic assumption as it is typically easy to kill tasks running on slow nodes once the fast instances complete).

7 Design alternatives

To address potential scalability problems with our centralized approach, a tree of controllers could be built that aggregates BARRIER_REACHED messages from children before sending a single message up the tree [18, 26, 37]. This tree could be built randomly from the expected set of hosts, or it could be crafted to match the underlying communication topology, in effect forming an overlay aggregation tree [34, 36]. In these cases, the master would send FIRE messages to its children, which would in turn propagate the message down the tree. One potentially difficult question with this approach is determining when interior nodes should pass summary BARRIER_REACH messages to their parent. Although a tree-based approach may provide better scalability since messages can be aggregated up the tree, the latency required to pass a message to all participants is likely to increase since the number of hops required to reach all participants is greater than in the centralized approach.

A gossip-based algorithm could also be employed to manage barriers in a fully decentralized manner [2]. In this case, each node acts as a barrier manager and swaps barrier status with a set of remote peers. Given sufficient pair-wise exchanges, some node will eventually observe enough hosts having reached the barrier and it will fire the barrier locally. Subsequent pair-wise exchanges will propagate the fact that the barrier was fired to the remainder of the nodes in the system, until eventually all active participants have been informed. Alternatively, the node that determines that a barrier should be released could also broadcast the FIRE message to all participants. Fully decentralized solutions like this have the benefit of being highly fault tolerant and scalable since the work is shared equally among participants and there is no single point of failure. However, since information is propagated in a somewhat ad hoc fashion, it takes more time to propagate information to all participants, and the total amount of network traffic is greater. There is an increased risk of propagating stale information as well. In our experience, we have not yet observed significant reliability limitations with our centralized barrier implementation to warrant exploring a fully decentralized approach.

We expect that all single-controller algorithms will eventually run into scalability limitations based on a single node's ability to manage incoming and outgoing communication with many peers. However, based on our performance evaluation (see Section 6), the performance of centralized barriers is acceptable to at least 100 nodes. In fact, we find that our centralized barrier implementation out-performs an overlay tree with an out-degree of 10 for 100 total participants with regards to the time it takes a single message to propagate to all participants.

8 Related Work

Our work builds upon a number of synchronization concepts in distributed and parallel computing. For traditional parallel programming on tightly coupled multiprocessors, barriers [19] form natural synchronization points. Given the importance of fast primitives for coordinating bulk synchronous SIMD applications, most MPPs have hardware support for barriers [23, 31]. While the synchronization primitives and requirements for large-scale networked systems discussed vary from these traditional barriers, they clearly form one basis for our work. Barriers also form a natural consistency point for software distributed shared memory systems [6, 20], often signifying the point where data will be synchronized with remote hosts. Another popular programming model for loosely synchronized parallel machines is message passing. Popular message passing libraries such as PVM [15] and MPI [25] contain implementations of barriers as a fundamental synchronization service.

Our approach is similar to Gupta's work on Fuzzy Barriers [17] in support of SIMD programming on tightly coupled parallel processors. Relative to our approach, Gupta's approach specified an entry point for a barrier, followed by a subsequent set of instructions that could be executed before the barrier is released. Thus, a processor is free to be anywhere within a given region of the overall instruction stream before being forced to block. In this way, processors completing a phase of computation early could proceed to other computation that does not require strict synchronization before finally blocking. This notion of fuzzy barriers were required in SIMD programs because there could only be a single outstanding barrier at any time. We can capture identical semantics using two barriers that communicate state with one another. The first barrier releases all entering nodes and signals state to a second barrier. The second barrier only begins to release nodes once a sufficient number (perhaps all) of nodes have entered the first barrier.

Our approach to synchronizing large-scale networked systems is related to the virtual synchrony [2, 3] and extended virtual synchrony [27] communication models. These communication systems closely tie together node inter-communication with group membership services. They ensure that a message multicast to a group is either delivered to all participants or to none. Furthermore, they preserve causal message ordering [22] between both individual messages and changes in group membership. This communication model is clearly beneficial to a significant class of distributed systems, including service replication. However, we have a more modest goal: to provide a convenient synchronization point to coordinate the behavior of more loosely coupled systems. It is important to stress that in our target scenarios this synchronization is delivered mostly as a matter of convenience rather than as a prerequisite for correctness. Any inconsistency resulting from our model is typically detected and corrected at the application, similar to soft-state optimizations in network protocol stacks that improve common-case performance but are not required for correctness. Other consistent group membership/view advancement protocols include Harp [24] and Cristian's group membership protocol [8].

Another effort related to ours is Golding's weakly consistent group membership protocol [16]. This protocol employs gossip messages to propagate group membership in a distributed system. Assuming the rate of change in membership is low enough, the system will quickly transition from one stable view to another. One key benefit of this approach is that it is entirely decentralized and hence does not require a central coordinator to manage the protocol. As discussed in Section 7, we plan to explore the use of a distributed barrier that employs gossip to manage barrier entry and release. However, our system evaluation indicates that our current architecture delivers sufficient levels of both performance and reliability for our target settings.

Our loose synchronization model is related in spirit to a variety of efforts into relaxed consistency models for updates in distributed systems, including Epsilon Serializability [30], the CAP principle [14], Bayou [32], TACT [38], and Delta Consistency [33].

Finally, we note that we are not the first to use the arrival rate at a barrier to perform scheduling and load balancing. In Implicit Coscheduling [11], the arrival rate at a barrier (and the associated communication) is one consideration in making local scheduling decisions to approximate globally synchronized behavior in a multiprogrammed parallel computing environment. Further, the way in which we use barriers to reallocate work is similar to methods used by work stealing schedulers like CILK [5]. The fundamental difference here is that idle processors in CILK make local decisions to seek out additional pieces of work, whereas all decisions to reallocate work in our barrier-based scheme are made centrally at the barrier manager.

9 Summary

Partial barriers represent a useful relaxation of the traditional barrier synchronization primitive. The ease with which we were able to integrate them into three different existing distributed applications augurs well for their general utility. Perhaps more significant, however, was the straightforward implementation of wide-area MapReduce as enabled by our expanded barrier semantics. We are hopeful that partial barriers can be used to bring to the wide area other sophisticated parallel algorithms initially developed for tightly coupled environments. In our work thus far, we find in many cases it may be as easy as directly replacing existing synchronization primitives with their relaxed partial barrier equivalents.

Dynamic knee detection in the completion time of tasks in heterogeneous environments is also likely to find wider application. Being able to detect multiple knees has added benefits since many applications exhibit multimodal distributions. Detecting multiple knees gives applications more control over reconfigurations and increases overall robustness, ensuring forward progress even in somewhat volatile execution environments.

References

[1] D. G. Andersen, H. Balakrishnan, and F. Kaashoek. Improving Web Availability for Clients with MONET. In *NSDI*, 2005.

[2] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *SOSP*, 1985.

[3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *SOSP*, 1987.

[4] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *CACM*, 36(12), 1993.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP*, 1995.

[6] M. Z. Brian Bershad and W. Sawdon. The Midway Distributed Shared Memory System. In *CompCon*, 1993.

[7] Computing TCP's Retransmission Timers (RFC). `http://www.faqs.org/rfcs/rfc2988.html`.

[8] F. Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *DC*, 4(4), 1991.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters . In *OSDI*, 2004.

[10] E. Dijkstra. The Structure of the "THE"-Multiprogramming System. *CACM*, 11(5), 1968.

[11] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *SIGMETRICS*, 1996.

[12] EMAN. `http://ncmi.bcm.tmc.edu/EMAN/`.

[13] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. GGF, 2002.

[14] A. Fox and E. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *HotOS*, 1999.

[15] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *C– P&E*, 4(4):293–312, 1992.

[16] R. Golding. A Weak-Consistency Architecture for Distributed Information Services. *CS*, 5(4):379–405, Fall 1992.

[17] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *ASPLOS*, 1989.

[18] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *IJPP*, 18(3):161–180, 1990.

[19] H. F. Jordan. A Special Purpose Architecture for Finite Element Analysis. In *ICPP*, 1978.

[20] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX*, pages 115–131, 1994.

[21] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7), 1978.

[23] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *JPDC*, 33(2):145–158, 1996.

[24] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *SOSP*, 1991.

[25] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[26] S. Moh, C. Yu, B. Lee, H. Y. Youn, D. Han, and D. Lee. Four-ary tree-based barrier synchronization for 2d meshes without nonmember involvement. *TC*, 50(8):811–823, 2001.

[27] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *CACM*, 39(4), 1996.

[28] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets*, 2002.

[29] Plush. `http://plush.ucsd.edu`.

[30] C. Pu and A. Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, 1991.

[31] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS*, 1996.

[32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

[33] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *PODC*, 1999.

[34] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *TCS*, 21(2), 2003.

[35] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[36] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *SIGCOMM*, 2004.

[37] J.-S. Yang and C.-T. King. Designing tree-based barrier synchronization on 2d mesh networks. *TPDS*, 9(6):526–534, 1998.

[38] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *OSDI*, 2000.