An Information-Based Approach to Distributed Systems Design

Andrea C. Arpaci-Dusseau   Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison

Abstract

We propose an information-based methodology for developing distributed systems. At the core of such an approach is the need to understand all possible sources for gathering information. In this paper we develop an information taxonomy and explore the costs and benefits of each source through a case study of the River I/O environment for clusters. The investigation, performed through targeted simulations and implementation measurements, focuses on the two distributed algorithms that are the foundation of dynamic adaptation within River. By following our information-centric approach, we design two efficient and scalable data-transfer mechanisms, which in turn yield robust application performance.

1 Introduction

The key to making intelligent decisions in a parallel or distributed system is timely access to accurate *information*. Each component needs information about the state and behavior of the rest of the system so that it can adapt its behavior to current conditions. For example, when placing jobs across a distributed system, a scheduler would like to know the load on each processor so that it can pick the one with the least load; a TCP client would like to know the congestion along the links of the network so that it can calculate the best sending window-size; when a cache controller marks an entry exclusively owned, it needs to know whether other caches also contain this entry.

Methods used to gather information can affect the performance, reliability, and other properties of a system. Despite this clear importance, many system designers employ an *ad hoc* approach, which can unfortunately lead them to pick one of two non-optimal extremes: to either assume no information is available and thus utilize static algorithms, or to employ explicit queries of remote nodes for information, incurring unnecessary overhead and complexity.

Only by understanding the full range of available methods can system architects select the appropriate algorithms for their system. In this paper, we develop a generalized information-centric methodology for designing distributed systems and introduce an information taxonomy to catalog and characterize the various methods of information dispersal.

In our information taxonomy, there are four ways in which information can be gathered: null, implicit, parasitic, and explicit. The base case is the *null* method, where no information is exchanged through communication between components; components either have no information about the state of the system or rely on algorithmic knowledge of how other components behave. With *implicit* information, components observe the contents and properties of existing communication to infer the traits of remote components. *Parasitic* information inserts a small amount of information into existing communication structures, thus propagating additional information along the natural data flows of the service. Finally, with *explicit* information, components introduce additional communication into the system to query remote entities about their state and behavior.

To investigate the value of our approach and taxonomy, we perform an in-depth exploration of the trade-offs of the different sources of information for a particular distributed system: the River I/O environment for clusters [4, 5]. The main goal of River is to enable data-intensive applications to adapt to *performance faults* of components in a cluster – that is, to perform well in spite of unexpected variations in performance of disks, workstations, or other components within the system.

Central to run-time adaptation in River are two distributed algorithms, available to applications as a layer of reactive middleware. The first of these algorithms is a distributed queue. Applications can utilize the distributed queue to move data to where it is best processed, and thus ameliorate the effects of consumers with performance faults. The second is a parallel, replicated data transfer known as graduated declustering. Graduated declustering allows applications to harness replicated data sources intelligently, equally sharing producer bandwidth among consumers so as to tolerate performance faults on data producers. In tandem, these two constructs can be utilized to fashion robust applications.

We explore the various methods of information dispersal within the distributed queue and graduated declustering algorithms with a combination of simulation and implementation. By modifying the algorithms to use different techniques for information dispersal, we can judge the relative merits of each. In particular, we identify the important decisions made by each algorithm and determine the information needed at those points to foster appropriate actions.

From our case study, we determine that null methods are not appropriate for algorithms that must adapt to performance faults; information is required to avoid components that are not performing as well as expected. We find that implicit and parasitic information is particularly useful for River; these sources allow components to adapt to current conditions in a scalable and efficient manner. Finally, we show that requests for explicit information must be carefully added, as they are likely to increase complexity and perhaps decrease overall robustness of a system.

More generally, we find that our information-based approach facilitates a deeper understanding of the design decisions one must make in a distributed algorithm. By employing the appropriate methods from our taxonomy, we were able to construct two robust scalable data-transfer mechanisms that in turn enable the assembly of robust data-intensive applications.

The rest of the paper is organized as follows. We introduce our information-based methodology and the taxonomy for gathering information in Section 2. Section 3 motivates River as an interesting case study and describes our evaluation environment. In Sections 4 and 5, we study the application of information to the distributed queue and graduated declustering, respectively. We examine application performance in Section 6, and conclude in Section 7.

2 Information-based Methodology

In this section, we discuss how components can obtain information about remote components in a distributed system. To illustrate how obtaining information fits into the behavior of a distributed-system service, we begin by describing a generic framework for designing a service. We then introduce a taxonomy of sources of remote information and illustrate their usage in real systems from the literature.

2.1 Design of Distributed Services

One of the main challenges in designing a distributed service or algorithm is to architect the flows of communication. For our information-centric approach, we suggest breaking this process into three steps, as described below. The key ingredient is the separation of the data transfer from the flow of information.

Design the data transfer: The first step in building a service or application in a distributed system is to architect the flow of data messages across components. We define *data* as the communication that is absolutely essential to implement the service, given the protocol of underlying layers. Thus, the data communicated is specific to each service. For example, the necessary communication for reading a block of data from a remote machine would include the request for the block, followed by the response which includes the requested block. This step produces the *data architecture* of the system.

Identify control points: In this next step, the designer constructs the *control architecture*, by identifying the *control points* of the system. At a control point, a component makes a decision that potentially affects the behavior of other components; for example, a component may decide to which machine to send a particular piece of data.

Design the information flow: Finally, the *information architecture* is assembled. For a component to make an appropriate decision at a control point, it may need information about the state of the rest of the system. There is an inextricable link between the information an algorithm needs and how it should be obtained. For example, low-cost but stale information may be preferred over up-to-date but costly information. Methods for obtaining information are the focus of this paper.

2.2 Information-Gathering Taxonomy

To make an informed decision at a control point, a component needs information about the state and behavior of other components. We classify the fundamental methods for obtaining such information into four primary categories: null, implicit, parasitic, and explicit, as depicted in Table 1. We describe these approaches beginning with those that require the least amount of communication.

Null: The simplest approach to guiding decisions is the *null* approach, which does not leverage communication to gather information in any way. There are two cases in which this approach may be appropri-

Method of Gathering Information	**Definition**	**Advantages**	**Disadvantages**
Null	No communication is leveraged	Simple to build; simple to reason about	Non-adaptive; can lead to poor decisions
Implicit	Observed from existing communication	Available "for free"; elegant	Difficult to reason about; proper info not always available
Parasitic	Embedded in existing communication	Additional information available at low cost	Information flow restricted to that of data flow
Explicit	Adds new communication	Any information with interface can be gathered	Cost of additional communication; interface may not be available

Table 1: **A Taxonomy of Information-Gathering Methods.** *A taxonomy of different information dispersal and gathering techniques is presented. The perceived "cost" of gathering the information increases with each method listed in the table, where cost is measured in terms of additional communication.*

ate: when components need no information at control points, and when information is obtained via algorithmic knowledge. We discuss the two cases in turn.

When an algorithm requires no information to guide its control points, then no information needs to be gathered. Such algorithms are static and cannot adapt to current conditions. For example, traditional disk striping and RAID algorithms all make static decisions as to where to place disk blocks, regardless of current load in the system [32].

Of course, the fact that information is not gathered does not imply that each component has no knowledge of remote behavior: components may have *algorithmic* knowledge of how remote components act. For example, a component may know that all other components were implemented using the same algorithm as itself or follow some defined specification. Much theoretical work has investigated the problem of making decisions with only general knowledge of how other components behave; at a high level, *game theory* [30] predicts the actions of adversaries in the absence of any communication, while *decision theory* [9] optimizes the utility of decisions based on uncertain information that has been quantified with a probability measure. As a specific example, in the LARD load-balancing web server presented in [31], the front-end leverages its algorithmic knowledge of how back-end servers cache web pages to usually direct requests for the same page to the same server, thus efficiently using aggregate memory.

The primary advantage of a null method is that there is no communication overhead for gathering the information. The disadvantage is that components may not have accurate information about the state of the rest of the system and may make poor decisions.

Implicit: The lowest-overhead method for obtaining accurate information about the current state of a changing system is to observe existing communication. These *implicit* methods fall into two categories: those that monitor the contents of messages, and those that observe information available outside of defined interfaces. Implicit methods are often most powerful when combined with algorithmic knowledge, thus allowing a component to infer not only the current state of other parts of the system, but also perhaps their subsequent actions.

The contents of messages have been used to guide decisions in many systems, including cache coherence algorithms for shared-memory multiprocessors [18] and contention-avoidance in Ethernet networks [27]. For example, caches on a shared bus snoop all traffic to main memory; if a cache controller detects a write to a data block, it infers that another controller now owns this block and responds by invalidating its own copy [18].

Implicit methods can additionally observe information that is available outside of the specified interface; for example, observing not just the contents of a data message, but also the time between messages. This form of implicit information is identical to a covert channel [24] in the field of security. Distributed systems that have leveraged this type of implicit information include the TCP congestion-control algorithm [21], which observes packet loss to infer congestion, and the coordinated scheduling of parallel jobs [3, 35], which infers remote scheduling state from message arrivals and round-trip time.

The central advantage of implicit methods is that they provide information for free – no communication beyond the requisite data messages is required, only the ability to deduce remote behavior from a local observation. However, there are disadvantages to these methods as well. First, the inferences that must be made are often subtle and can lead to a system that is more difficult to reason about and construct. Second, information flow is restricted to the exact path of the data flow of the system – if those paths are not ideal, then neither is the information flow in the system.

Parasitic: The *parasitic* category leverages existing data messages to disseminate additional information throughout the system, much as a parasite may use its host for means other than that of the original intention. This approach is also referred to as "piggy-backing". Many systems use the parasitic method to reduce the number of messages a protocol requires, for example by inserting an acknowledgement within a data message in the reverse direction [8]. In this paper, we focus on its application to information dispersal. For example, in Sun's NFS version 3 [29], to reduce the frequency of `getattr` requests, attribute information is returned with the results of *all* file operations; because a `getattr` request is only generated if attribute information is more than 30 seconds old, the number of explicit `getattr` requests is reduced.

The advantage of parasitic methods is that they are low cost, usually adding only a few bytes to existing messages. However, similar to implicit methods, information flows are restricted to data communication channels, which can be a limitation.

Explicit: Finally, the most direct manner in which to obtain information is via an *explicit* query of one or more remote entities. Explicit approaches are quite common; for example, a randomized load-balancing scheme described in [1] sends explicit probes to two randomly chosen nodes to determine their load, and then places a job on the lesser loaded of the two.

The main benefit of explicit methods is that they are simple to understand and do not require inferences or knowledge of the behavior of other components to be correct. However, there are also a number of disadvantages. First, there may not be an interface for obtaining the desired information. Second, accessing an explicit interface may be too costly – sending an explicit request consumes shared network resources that may be needed for data transfers and induces additional work on the remote node. Finally, when explicitly contacting a remote node, one must be prepared to deal with the many kinds of remote failure that may occur.

2.3 Discussion

When designing a distributed service, we recommend a simple three-step process. First, design the data architecture, which dictates how communication generally flows to achieve the desired service. Then, develop the control architecture, which indicates where important decisions are made. Finally, layer in the information architecture, which specifies how information is gathered to inform the control decisions.

We also advocate a step-wise approach to the design of the information architecture. We believe that designers of distributed systems should obtain information with the method that incurs the lowest overhead while sufficiently guiding decisions at control points. That is, if information is needed beyond that available from knowledge of the algorithm, the designer should use implicit information available from existing data communication to infer remote state. If this implicit information does not lead to decisions that are of acceptable performance, then the designer should consider adding parasitic information. And, finally, only if parasitic information fails to provide sufficient guidance should explicit information be added. This process is naturally iterative, in that design and testing should occur at each stage to assess the efficacy of the deployed information methods.

3 Case Study: River

To evaluate the range of methods for gathering information in a distributed system, we consider its application in a case study of the River programming environment. In this section, we describe why the River system is an interesting case study for information and then describe our experimental environment.

3.1 Motivation

River provides a generic data-flow programming environment for clusters of workstations, similar in its basic design to previous parallel database environments such as Volcano [19]. River is an interesting environment for studying information for three primary reasons. First, River solves an important problem: it enables data-intensive applications to perform robustly even when underlying components are not performing as expected. Second, River fundamentally requires information about the state and behavior of remote components to adapt to performance problems. Third, the River environment contains multiple distributed algorithms whose information needs can be compared with each other. We now discuss these reasons in more detail.

Solves an important problem: River supports the construction of parallel, data-intensive applications – those at the core of database query-processing primitives, out-of-core scientific programs, and many Internet services. While this is a worthwhile domain in its own right, River solves the additional problem of ensuring that these applications perform robustly even when underlying hardware and software components perform erratically. Although previous work has addressed how to design large-scale systems that can tolerate *correctness faults* in individual components [7, 17, 32, 34], little work has focused on how to design systems that tolerate *performance faults*, where a performance fault is the unexpected low performance of a component within the system.

Within clusters, performance faults are commonplace [2], especially for modern disks. Some of the reasons static and dynamic performance faults appear both within a single disk and across disks include: the presence of multiple zones [28], SCSI bad-block re-mapping [4], thermal recalibration [12], sporadic performance before absolute failure [36], contention due to workload imbalance, and heterogeneity due to incremental growth [13]. Thus, the goal of River is to enable *performance availability* – application performance that tracks the aggregate performance of all components in the system, degrading gracefully under an increasing number of performance faults.

Fundamentally requires information: Current systems that support data-intensive applications do not interact well with performance faults because they use static techniques to exploit parallelism and distribute data. For example, standard striping algorithms for distributing data requests over a set of disks place the same amount of data on each disk. The problem with static schemes is that they make rigid assumptions about the relative performance of different components – often that all perform identically – without gathering information about their actual performance. As a result, the performance of the overall service is determined by when the slowest component finishes.

To dynamically adjust to performance faults, the distributed system must know the current state of remote components. To implement a high performance service that scales to large numbers of components, this information ideally would be accurate, up-to-date, and gathered without overhead or sensitivity to performance faults.

Consists of multiple distributed algorithms: River provides two distributed software constructs, a distributed queue (DQ) and graduated declustering (GD). Each dynamically adjusts how data is transfered from a set of producers (*e.g.*, processes of a parallel application) to a set of consumers (*e.g.*, a set of disks). Because many parallel data-intensive applications consist of one or more such transfers, a performance-available environment can be provided by building a performance-available data transfer. The two constructs in River each solve half of the problem; in tandem, they can be used to build performance-available applications.

When transferring data from a set of producers to a set of consumers, one or more consumers might suffer from a performance fault. The proper reaction in such a case is to move more data to faster consumers, where the amount of data is proportional to the relative speed of the consumer. This functionality is provided by the distributed queue.

A more difficult problem occurs in a data transfer from producers to consumers when a producer incurs a performance fault. If the data source is replicated across multiple producers, River applications can employ graduated declustering, a data transfer mechanism that carefully divides available producer bandwidth equally among consumers, and thus provides performance availability under producer faults.

Evaluating sources of information for both algorithms side-by-side is especially interesting because although the algorithms have many characteristics in common, they vary in one fundamental aspect: the amount of flexibility they have to adjust to slow components. In particular, the DQ can avoid one slow consumer by sending the data to *any* other consumer; GD can only avoid a slow producer by gathering the data from one of the other producers replicating that data. As we will see, this more restrictive environment forces GD to gather more information than the DQ to achieve robust performance.

3.2 Experimental Environment

To study the costs and benefits of different information sources in the distributed queue and graduated declustering, we employ both simulations and implementation. With an event-based simulator [4], we explore basic algorithmic behavior under a wide range of system parameters, most of which would not be feasible to measure in the implementation. Measurements of the implementation are then gathered to confirm simulation results, as well as to bring forth various "systems issues" not modeled in the simulations. The combination of both techniques improves understanding, better separating intrinsic properties from implementation details.

River runs on a cluster of Ultra1 workstations, each running Solaris 2.6 [16]. Each workstation consists of a 167 MHz UltraSPARC I processor [37], 128 MB of memory, and two Seagate Hawk 2.1 GB 5400-RPM disks attached on a fast-narrow SCSI bus. One disk is commonly used by the OS, whereas the other is used by River for data. These disks can deliver a peak bandwidth of 5.45 MB/s.

The workstations are connected together via a high-speed Myrinet local-area network [11]. Each workstation has a single Myrinet card; these cards are capable of moving data into and out of the workstation at approximately 40 MB/s. The entire system is connected together via a series of 8-port 640 MB/s Myrinet switches in a 3-ary fat tree. All communication is performed with Active Messages (AM) [26], which exposes most of the raw performance of Myrinet while integrating with features such as threads, blocking on communication events, and multiple independent endpoints.

4 The Distributed Queue

We now apply our information-based approach to the distributed algorithms of River, beginning with the distributed queue. In each of the next two sections, we first present the general interface to the mechanism under study, identify the control points, and then examine how the different approaches from our information taxonomy can be applied.

4.1 Interface and Semantics

As described in the previous section, the distributed queue presents applications with a high-speed backplane for data sharing and can be used to tolerate consumer-side performance faults. Within an application, producers place data into the distributed queue by calling `Put(Data, Size)`, and consumers receive the data by calling `Get(&Data, &Size)`. Thus, a consumer can logically receive any data block that has been `Put` into the system.[1] The central challenge in designing a DQ is to move data from producers to consumers such that faster consumers receive a proportionally higher amount of the aggregate producer bandwidth. The implementation must accomplish this goal in a distributed and scalable manner.

[1]Note that no ordering of data is guaranteed, except point-to-point; if a producer places A into a DQ before B, and if the same consumer receives both A and B, it will receive A before B. Strictly speaking, this is more like a bag than a queue.

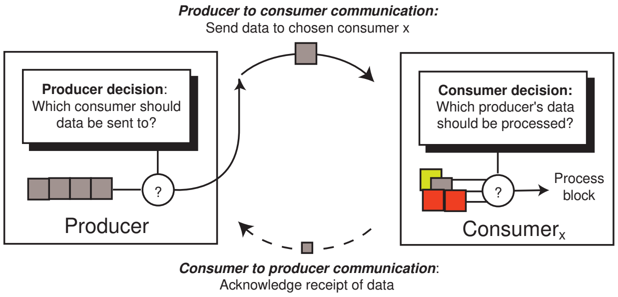

Figure 1: **DQ: Data and Control.** *The diagram depicts the basic data movement and control options of the distributed queue. A producer that wishes to send a data block is faced with a decision: to which consumer should data be sent? Once decided, the data block is sent to the consumer. Each consumer receives blocks from many producers, and thus is faced with its own control option: in which order should blocks be processed from different consumers? For each block received, the consumer sends a reply to the producer, as required by the Active Messages request/response protocol.*

In many ways, the distributed queue is reminiscent of a Linda tuple space [15], but is more restricted in its usage – for example, entities are not allocated storage within the distributed queue and are simply passed directly from producer to consumer. Therefore, the DQ is designed strictly for high-performance data transfers that can tolerate consumer-side performance faults. The DQ also bears similarity to many distributed load balancing constructs [1, 10, 22, 39]; however, the goal of the DQ is performance-available data transfers under consumer performance faults, and as we will see below, this is not a one-to-one mapping to traditional load-balancing schemes.

4.2 Data and Control

Figure 1 presents the logical structure of data flow in a DQ. Each producer has data to distribute amongst the set of consumers, and each consumer has a queue of incoming data blocks to process. Thus, data is transferred in parallel from the set of producers to the set of consumers that are part of the queue.

The flow of a particular block, and thus the behavior of the system as a whole, is determined by two decisions. Each of these control points potentially requires global information, and thus are candidates for exploring our information taxonomy. First, each producer chooses which consumer should receive a given

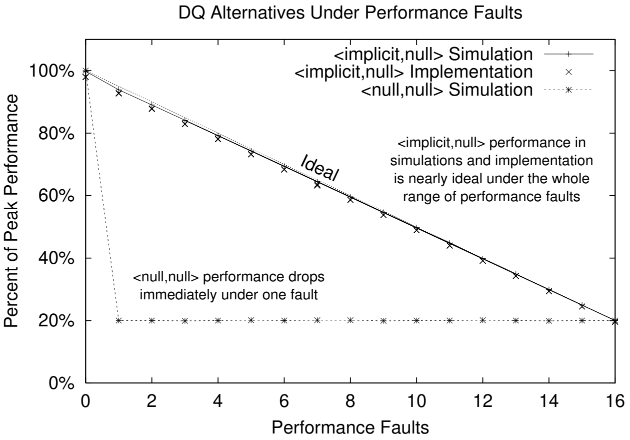

Figure 2: **DQ Performance (Null and Implicit).** *Results from a simulation of both the <null, null> and <implicit, null> DQ, as well as an implementation of the <implicit, null> DQ, are compared to ideal performance under an increasing number of performance faults. In the experiments, producers generate data at 40 MB/s and non-faulty consumers consume data at 5 MB/s. A performance fault reduces a consumer's performance by a factor of 5, down to 1 MB/s.*

data block. We will see that this is the most important decision in a performance-available distributed queue algorithm. Second, each consumer must choose in which order to process the blocks it has received. We now examine how the different levels of the information taxonomy can be applied to make these decisions within the distributed queue.

4.3 Null Information

We begin our study of information alternatives with the simplest alternative: no information. Because the distributed queue consists of two decisions, we refer to the distributed queue with no information for both decisions as the *<null, null>* DQ.

The first such decision is for a producer: which consumer should receive this data block? There are many possible information-free algorithms, such as sending to consumers in a round-robin or random pattern. Because a round-robin pattern may lead to excessive endpoint contention [14], we focus on a random choice of a consumer. The second decision is for a consumer: in what order should blocks be processed? Again, many null alternatives are possible; we study a simple first-come, first-served policy.

Figure 2 illustrates the performance of a simulated *<null, null>* DQ. In these experiments, 16 producers send data to 16 remote consumers (32 total machines). The producers are able to generate data at 40 MB/s, whereas the non-faulty consumers can sink data at 5 MB/s; this set-up emulates a set of processes writing in-memory records in parallel to disk, where the disks (consumers) are the bottlenecks in the transfer. Along the x-axis, we increase the number of consumers undergoing performance faults, where a performance fault reduces the level of a consumer's performance by a factor of five. Thus, the left-most point along the x-axis shows performance in the system under zero performance faults, and the right-most point shows performance when all consumers perform at the perturbed rate.

The y-axis plots performance as a fraction of peak in the non-perturbed case (*i.e.*, no faults in the system), where peak is simply the aggregate bandwidth across the bottleneck in the transfer; in this case, the consumers are the bottleneck. The 'Ideal' performance line, shown in this and subsequent graphs, is computed as the fraction of peak performance that is available under a given number of performance faults.

As shown in the figure, with zero performance faults, the *<null, null>* DQ performs well; throughput is high and scales well to 32 machines. However, with just a single fault, performance drops immediately to that of the slowest consumer. As performance faults are added into the system, null information schemes fundamentally cannot react to the changing characteristics of the consumers in the system and cannot provide performance availability. Thus, to support the desired behavior of the distributed queue, some form of global information must be dispersed throughout the system, allowing producers to adapt to the volatile performance delivered by consumers.

4.4 Implicit Information

Implicit information elegantly solves the most basic problem of the distributed queue: a lack of adaptation to varying rates of consumption. Implicit information provides *feedback* through the request/response paradigm of data messages. When a producer sends data to a consumer in the form of an Active Message request, the protocol stipulates that the consumer replies to that message.[2] In the *<null, null>* DQ, the reply is an empty message. The *<implicit, null>* DQ uses the reply as a "signal"; if the consumer handled the data request, the producer infers that the remote consumer is making progress. Thus, each producer monitors the incoming replies to infer the current per-

[2]This reply message is also required in protocols that utilize acknowledgements to implement reliable message transfers.

formance level of each consumer.

To leverage this implicit information in our $<$*implicit, null*$>$ algorithm, each producer records whether it has an unacknowledged message outstanding to each consumer. When a producer needs to send a block of data, it finds a consumer without an unacknowledged message. If there is more than one such consumer, the producer chooses among them uniformly at random. If there are no such consumers, the producer relinquishes the processor and waits for a consumer to reply.[3] When a consumer receives a data block, it processes blocks in first-come, first-served order, and sends an empty Active Message reply to the producer. When the producer handles the reply, it records that there is no longer an outstanding message to this consumer.

The intuition behind this implicit algorithm is that consumers that respond more quickly to requests will receive more subsequent requests from the producer. Thus, with the implicit information, the producer has a window into consumer performance: at each instant, the producer "trusts" those consumers that have adequately responded to data requests and does not trust those who have not responded.

This algorithm also has the property of unifying the seemingly diverse implementations of "push-based" and "pull-based" approaches often found in message-passing layers [23]. When producers are the bottleneck for the data transfer, all consumers respond quickly to data requests, and the choice of destinations degenerates to a random choice from the entire set of consumers – a "push-based" approach. However, when the consumers are the bottleneck, each producer has all of its messages outstanding, one per consumer; thus, when a reply returns, it implicitly "pulls" the next data request to the replying consumer. Thus, depending on the relative performance rates of producers and consumers, the $<$*implicit, null*$>$ DQ can take on characteristics of either a push-based or pull-based approach.

As shown in Figure 2, the simulated performance of the implicit distributed queue under performance faults is excellent, delivering nearly ideal performance across the range of induced faults. The figure also shows that our implementation results, with system parameters set equal to those of the simulation, closely match the simulation results.

[3]This scheme is easily extended to allow more than one outstanding block per consumer, though we have found that one is sufficient through a range of experiments reported elsewhere [4].

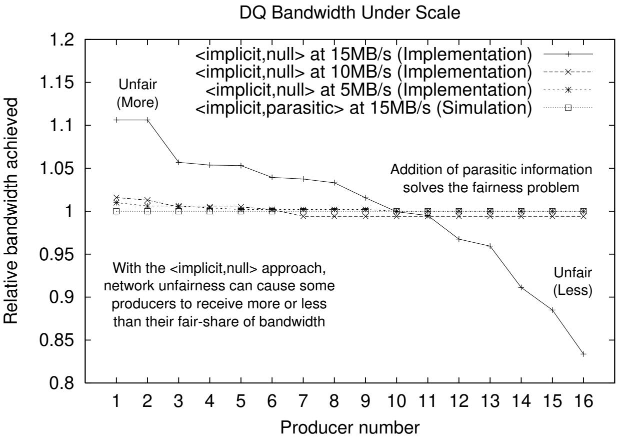

Figure 3: **DQ: Fairness.** *The graph depicts the results of three experiments with an implementation of the* $<$*implicit, null*$>$ *DQ, as well as a simulation of the* $<$*implicit, parasitic*$>$ *DQ. In all experiments, there are 16 producers sending data as fast as they can to 16 remote consumers (a total of 32 machines). The rate of the each consumer is controlled, and varied from 5 to 10 to 15 MB/s. There is a spot along the x-axis for each producer in the system, and the y-axis plots the proportion of total bandwidth that producer x received during the experiment. Thus, a fair distribution of bandwidth leads to all producers with a y-value of '1'.*

4.5 Parasitic Information

Although implicit information is useful in choosing the destination for a particular data block, the $<$*implicit, null*$>$ DQ still uses a non-informed algorithm (first-come, first-served) for processing blocks on the consumer. When performing more detailed measurements of our implementation, we found that under high loads and with many participating nodes, the Myrinet switches do not fairly schedule transfers and thus some producers do not receive their fair share of network bandwidth. Since overall performance is determined by the rate of the slowest entity, unfairness causes performance of the system to drop.

Figure 3 plots the relative bandwidths of producers in three different experiments using our implementation. In these experiments, there are 16 producers sending data to 16 consumers, with no performance faults in the system. The per-consumer bandwidth is set to three different levels: 5, 10, and 15 MB/s. Ideally each producer should obtain a fair-share of the aggregate consumer bandwidth, and indeed, this occurs for per-consumer bandwidths of both 5 MB/s and 10 MB/s. However, at a per-consumer bandwidth of 15 MB/s, some producers receive a substantially lower proportion of aggregate consumer bandwidth.

Parasitic information provides a solution to the fairness problem. Along with the data, each producer also sends its current bandwidth. Consumers then use this information to schedule which data blocks they process: each consumer biases service to those producers that are lagging and thus encourages global fairness and good performance in spite of network-switch unfairness. Note that the <*implicit, parasitic*> DQ is nearly ideal with respect to the amount of communication it performs, logically adding a few bytes of information to every large (8 KB) data transfer.[4]

To verify the performance of the <*implicit, parasitic*> DQ, we modified our simulator to model switch unfairness. The simulation results of this version in Figure 3 indicate that the <*implicit, parasitic*> DQ performs as desired. Although we have not yet implemented this approach in our real system, we are confident that the parasitic information will equally aid its performance. We note that only through a combination of simulation and measurements of our implementation were we able to pinpoint and fix this problem so readily.

4.6 Explicit Information

Whereas we have successfully utilized implicit information to choose a destination for a block of data, others in the load-balancing literature propose explicit approaches. In particular, Adler *et al.*, suggest a simple probe-n model in which a producer sends out explicit probes to a random subset of n consumers before sending a data block [1]. These probes return the load at each consumer, which is used by the producer to pick the consumer with the lowest load.

The intuition behind this algorithm is that the time to send a probe is fast, roughly equal to the round-trip latency of the network. However, when we implemented this approach, we realized that explicit queries make the dangerous performance assumption that the probe returns promptly, exactly the type of assumptions we are seeking to avoid in the distributed queue! Not surprisingly, an implementation of the <*explicit, null*> distributed queue proved unsuccessful; performance dropped to the level of the slow consumers in the system, identical to that of the <*null, null*> algorithm shown in Figure 2.

Thus, even though the probe-n algorithm has been shown to balance load in a cluster, it does not do so in a performance-available manner. The performance assumptions made by explicit information queries result in brittle behavior when the assumptions are invalid, that is, when a consumer does not respond in a timely manner to those queries. The explicit approach has the further disadvantage of adding n probe messages for every data block. Finally, a probe-based approach significantly complicates the distributed queue, adding substantial amounts of code.

4.7 Discussion

We have studied the range of information sources for our performance-available, adaptive distributed queue. Of primary importance to an adaptive algorithm is feedback; thus, null methods cannot be used in systems that tolerate performance faults. This conclusion has implications for other systems as well; for example, traditional disk striping schemes, as well as parallel database techniques such as range- and hash-partitioning, are all null methods, and thus cannot adapt to components of varying performance levels.

The most important piece of information within the DQ is the consumption rate of remote consumers, which is needed when deciding where to send each data block. Implicit information is adequate for this decision, with consumer replies signaling the rate of remote consumer progress. This yields a high-performance algorithm, adequate for most uses.

However, consumer-side scheduling of block processing can be a factor under switch unfairness, and mandates the use of parasitic methods to disperse extra information throughout the system – the rate of progress of each producer. Thus, the combination of parasitic and implicit methods leads to a general and robust distributed queue.

Finally, with explicit methods, though more information about remote entities can be gathered, more performance assumptions are introduced. A producer that queries a number of remote sites for their current load must wait for those queries to return, which does not yield a performance-available DQ. Thus, explicit methods should be avoided unless all performance dependencies are well-understood.

5 Graduated Declustering

Whereas the distributed queue tolerates consumer-side performance faults (by moving more data to faster consumers), it suffers when a producer does not deliver data at the expected rate. If the data is not available from another locale and the producers are the bottleneck in the data transfer, then there is

[4]Our implementation incurs no overhead, because an Active Message has a fixed set of arguments, some of which are unused.

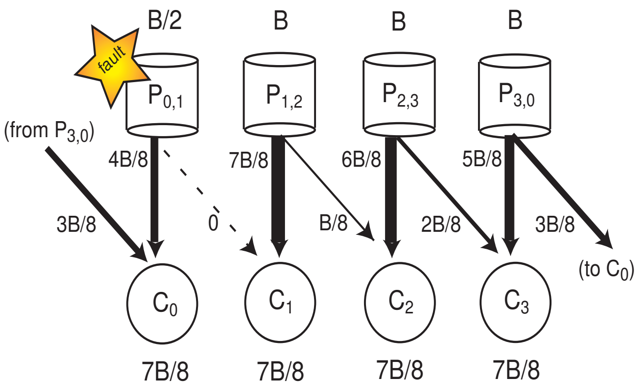

Figure 4: **GD Example.** *This diagram shows how GD alleviates the problem of producer-side performance faults. In the example, the disks are producers, and data is mirrored. Producer $P_{x,y}$ produces data sets x and y, and consumer C_x consumes data set x. Though $P_{0,1}$ performs at half its expected rate (B/2, not B), other producers compensate their bandwidth allocations, and all consumers receive a fair-share of available aggregate bandwidth ($\frac{7}{8}B$).*

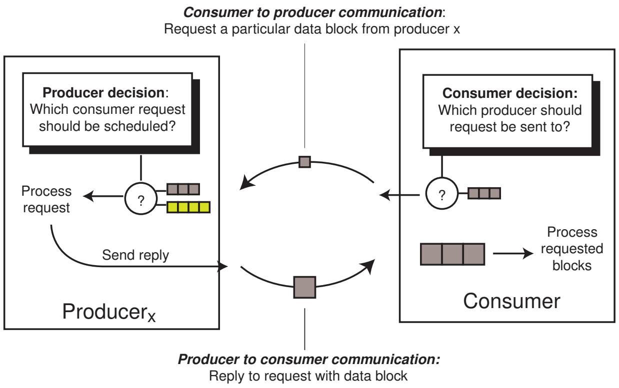

Figure 5: **GD: Data and Control.** *The diagram depicts the basic data movement and control options of graduated declustering. A consumer requesting a block is faced with a decision: to which producer should this request be sent? The request is then sent to the selected producer, who has requests from different consumers. The consumer then faces a decision: which queue should be serviced? Once serviced, data is sent back to the requesting consumer.*

no solution to this problem. However, in many cases, data is replicated for the sake of reliability; for example, in a mirrored disk system, each data block is available from two locations. In River, graduated declustering can be used to coordinate access to a replicated data set by dividing the aggregate producer bandwidth equally among consumers, and can thus avoid producer-side performance faults. An example of how GD should work is shown in Figure 4.

5.1 Interface and Semantics

The internal interface to graduated declustering is slightly different than that of the DQ. The consumer side is simple: consumers call `Get(&Data, &Size)`, which requests and eventually returns the requested data from one of the sources. The producer interface is more complex: producers call `GetRequest(&Request)` to get a request specification, and then `Put(Request,Data,Size)` to send the results of the request back to the requesting consumer.

Graduated declustering is most similar to chained declustering [20], a technique that improves performance under an absolute failure in a mirrored disk system by carefully spreading replicated blocks across many disks. GD extends this concept to tolerate the full range of performance faults. GD is also similar to some of the adaptive techniques for choosing which replica to read within Petal [25]; however, Petal does not coordinate activity among parallel clients in the global manner of GD, and thus is not as successful in avoiding the potentially harsh effects of performance faults on parallel clients.

5.2 Data and Control

Figure 5 presents the logical structure of data flow in graduated declustering. Each consumer sends its request for a particular block to one of the replicated sources for that block; this decision represents the first control point. Each producer must then choose the order to handle consumer's requests; this decision represents the second control point.

5.3 Null Information

The *<null, null>* GD makes both decisions without any information. The consumer chooses a random producer for each data request, and the producer services those requests in a first-come, first-served manner. Performance of a simulated GD is shown in Figure 6, where each of 16 consumers requests data from two possible producers in the set of 16 producers. The non-faulty producers generate data at 5 MB/s, whereas consumers can sink data at 40 MB/s, a scenario which emulates a parallel read from disks (disks are the producers, and application processes are the consumers). Along the x-axis, we increase the number of producers with performance faults, where a performance fault reduces the level of a producers's performance by a factor of five. Not surprisingly, the *<null, null>* approach only works well when there are no performance faults in the system; performance degrades as soon as a single performance fault occurs.

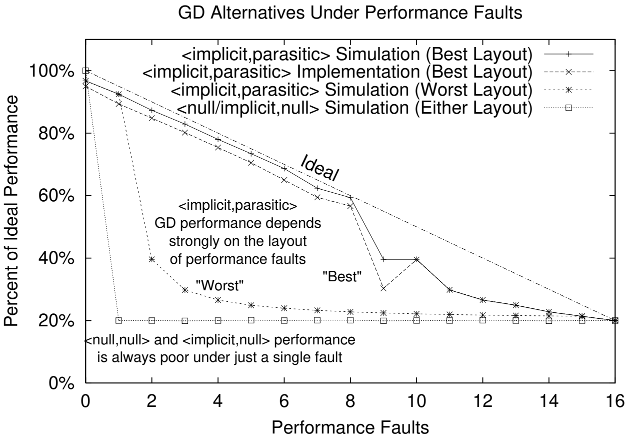

Figure 6: **GD Performance.** *Results from simulations of the <null, null>, <implicit, null>, and <implicit, parasitic> GD, as well as an implementation of <implicit, parasitic> GD, are compared to ideal performance under an increasing number of performance faults. In all experiments, non-faulty producers generate data at 5 MB/s and consumers can consume data at 40 MB/s. A performance fault reduces a producer's performance by a factor of 5, down to 1 MB/s. Because of limited replication, the layout of performance faults matters; 'best' implies that they are spread out, and 'worst' indicates that they occur consecutively on producers.*

5.4 Implicit Information

The *<implicit, null>* GD is modeled after the *<implicit, null>* DQ: each consumer sends its next request to the producer that responded most recently. The intuition behind this approach is to give more requests to more responsive servers, and thereby avoid slow producers. Unfortunately, the performance of the *<implicit, null>* algorithm is identical to that of the null method. Though the consumers adapt to more responsive producers, the producer scheduling does not: it is still first-come, first-served. This policy was successful in the DQ, but only due to the greater flexibility of producers: a producer could send data to *any* consumer. Here, a consumer can send a request to only a subset of producers, due to limited replication. Therefore, all producers must bias their scheduling so as to cooperatively account for slow producers. The implicit method cannot bring about this behavior.

5.5 Parasitic Information

We arrive at a performance-available GD algorithm with the addition of parasitic information. Analogous to the DQ, each consumer inserts its rate of progress into each request. A producer, then, has a localized view of the progress of each consumer it serves. The producer biases scheduling so as to service the lagging consumer. It is easily shown that if each producer is able to locally balance the bandwidths of the consumers it services, then all consumers receive an equal share of producer bandwidth. As with the *<implicit, null>* GD, consumers use an implicit method for the first decision, sending requests to the producer who most recently responded.

Figure 6 shows the performance of *<implicit, parasitic>* GD, both via simulation and implementation. For the simulations, two different performance-fault layouts are presented: in the "worst" case, faults occur consecutively, and thus immediately affect both data sources for a given consumer; in the "best" case, they are evenly distributed, never occurring on both of a consumer's producers until more than half of the producers are perturbed. The implementation results are only shown for the best layout. Overall, performance with the best layout of faults is good, though not ideal, due to the limited amount of replication; only if each producer contained every data set would the flexibility necessary for ideal performance be available. More generally, given that R is the level of replication for each data item and P is the number of producers in the system, then the *<implicit, parasitic>* GD algorithm can always tolerate the presence of $R - 1$ performance faults, and, potentially tolerate up to $P \cdot (1 - \frac{1}{R})$ faults.

5.6 Explicit Information

Finally, we examine the utility of explicit information movement within graduated declustering. As we saw in the distributed queue, adding explicit queries into a system built for performance availability can be dangerous; therefore, when adding explicit information to graduated declustering, we take care to avoid new performance assumptions. We are also again in the situation where the *<implicit, parasitic>* algorithm performs quite well; how could explicit information be useful in such a case?

We envision one possible alternative to the parasitic dispersal of progress information. The *<implicit, parasitic>* GD piggy-backs consumer progress on each request to a producer. The benefit of this approach is that it is low cost and achieves the defined goal. However, a potential disadvantage occurs when a consumer does not request data from a particular producer for some time. In that case, the producer does not have up-to-date knowledge of consumer progress and will not make an optimal scheduling decision. The explicit approach solves this problem by continually updating *both* producers: one pro-

Parasitic	Producer			
Consumer #	**1**	**2**	**3**	**4**
1	51.1	48.9	0.0	0.0
2	0.0	51.0	49.0	0.0
3	0.0	0.0	51.1	48.9
4	49.0	0.0	0.0	51.0

Parasitic/Explicit				
Consumer #	**1**	**2**	**3**	**4**
1	99.7	0.3	0.0	0.0
2	0.0	99.7	0.3	0.0
3	0.0	0.0	99.7	0.3
4	0.3	0.0	0.0	99.7

Table 2: **GD: Parasitic vs. Explicit** *The tables show the percent of requests for a particular consumer satisfied by a particular producer, as gathered via simulation. The first table shows the results for the <implicit, parasitic> algorithm, where each consumer receives roughly half of its requests from each possible producer. The second table shows the results for the <implicit, parasitic/explicit> approach, where each consumer receives almost all requests from a particular producer. In both experiments, there are no performance faults.*

ducer is sent the real request, the other a dummy request containing only progress information.

This disadvantage in the <*implicit, parasitic*> approach is seen when there are no faults in the system. In this case, each producer divides its bandwidth into two roughly equal parts for the two consumers it serves. This results in a slight performance cost (5-10%) because additional seeks are incurred when reading two streams from disk. The <*implicit, parasitic/explicit*> GD is able to algorithmically favor a particular consumer and deliver all of its bandwidth to that one consumer, while still being informed that the other consumer is making sufficient progress. Thus, the explicit approach avoids extraneous seeks. Table 2 presents simulation results of both approaches.

However, there are disadvantages and costs associated with the extra explicit information. First, twice as many requests are generated, which in certain environments could noticeably affect system performance. Second, the information-bearing requests may make performance assumptions. If they are implemented on a reliable protocol, each request consumes resources (*e.g.*, buffers for retransmission). Thus, if a particular producer is slow, repeated requests to it consume resources, eventually halting progress. Thus, information-dispersing requests should be built on an unreliable medium, such as raw U-net [38], and not Active Messages.

5.7 Discussion

The design of a successful graduated declustering algorithm requires more information than the baseline distributed queue; consumer progress must be passed from consumers to producers to facilitate producer-side scheduling, and producer rates must be known by consumers to direct requests to where they are best serviced. The former can be efficiently realized via parasitic information, whereas the latter with implicit methods. Without both kinds of information, one cannot develop a robust GD algorithm.

We have also seen that information about consumer progress can be more widely dispersed via explicit techniques, in particular with an explicit information-bearing request to producers. Though it only provides a slight performance improvement, it represents a class of information not available with null, implicit, or parasitic techniques. The <*implicit, parasitic/explicit*> approach also demonstrates that *hybrid* techniques can be leveraged to gather information.

6 Putting It All Together

To demonstrate the utility of the complete distributed queue and graduated declustering mechanisms, we give a brief summary of our application experience within River. Once we had a solid information-centric understanding of each algorithm, the final implementation of both mechanisms was straight-forward. The result is better performance for applications as compared to previous efforts [5]. [5]

We present the performance of six I/O-intensive, database query-processing primitives, each of which has been transformed from a static parallel application into a robust and adaptive version. Programmers insert distributed queues and graduated declustering into their applications to form "points of flexibility": places in the data-flow where performance faults can be tolerated. Note that application writers have no knowledge of information flow – they focus on constructing flexible flows, and the infrastructure handles the rest. By utilizing the two core River mechanisms, each application withstands performance faults on the disks and achieves near-ideal performance.

Figure 7 presents a summary of our results. In the figure, performance of the application with an increasing number of disk performance faults is shown, where a fault to the disk utilizes half of available disk bandwidth. For all applications, 'Ideal' is the amount

[5] There are also more applications in this paper.

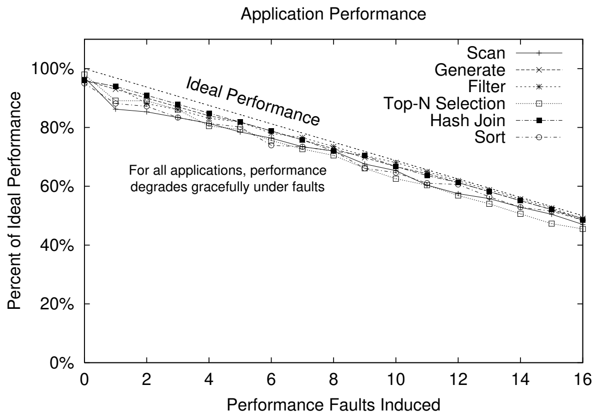

Figure 7: **Application Performance.** *Performance of all database primitives under disk performance faults is presented, as run on the implementation on 16 machines. The applications are a parallel scan of a data set (read only), a parallel generation of a data set (write only), a parallel filter (read data from disk, select certain records based on a user-defined function, write selected records to disk), a top-N selection (find the top N values in the data set, and be able to find the next N efficiently), a parallel hash-join of two data sets, and finally a parallel, external sort. Each application operates on roughly 150 MB of data per disk per node, for a total of 2.4 GB across 16 machines. A performance fault reduces performance by a factor of 2. Each data point represents the average of 10 runs; there was little variance among the numbers.*

of data touched by the application divided by the peak rate available from the disks. For example, the parallel scan reads a total of 2.4 GB from 16 disks, each of which can run at 5.45 MB/s. Thus, ideal time for the scan is $\frac{2.4\ GB}{16 \times 5.45\ MB/s} \approx 28.18$ seconds. In all cases, performance is excellent, with the measured execution time of River applications within 89% of ideal.

In summary, by carefully applying our information-based approach to the DQ and GD, we were able to build solid implementations of the distributed algorithms. "Information-ignorant" applications can take advantage of "information-aware" mechanisms and reap the benefits of excellent behavior under performance faults. In both instances, implicit and parasitic techniques proved to be elegant and low-cost methods for gathering the necessary information that the constructs needed for adaptation.

7 Conclusions and Future Work

In this paper, we have introduced and explored an information-based approach to systems design. By understanding the information needs of two different distributed algorithms and iteratively evaluating the range of approaches in the information taxonomy, we substantially improved the performance of our system. We believe that other system designers can follow the same steps with equally beneficial results.

In our case study of River, we found that implicit and parasitic information were particularly of value. For both the distributed queue and graduated declustering, the information derived by those two methods was sufficient to enable the design and implementation of robust, scalable, and elegant algorithms.

As distributed systems continue to increase in complexity, information-centric techniques will be critical. Each method we have classified – null, implicit, parasitic, and explicit – has its place, depending on the problem domain; evaluating the costs and benefits of each approach is thus central to a successful design.

In the future, we would like to increase the ease of use of information in distributed systems. An important step towards this goal is to develop *Information Programming Interfaces (IPIs)*, similar to traditional *Application Programming Interfaces (APIs)*. IPIs serve to isolate the information-gathering method from the rest of the algorithm, allowing implicit, parasitic, explicit, or hybrid methods to be employed depending on the environment. IPIs also increase portability; for example, when standard TCP congestion control is run in a wireless environment, a dropped message no longer indicates congestion, but can be due simply to the lossy medium [6]. If the congestion signal is encapsulated within an IPI, porting the IPI to a new setting forces designers to re-think how to gather information, avoiding incorrect inferences from implicit observations.

Finally, we plan to formalize not only the gathering of information, but also its *use* in distributed systems; this is especially interesting in environments where feedback across components is involved. We hope that the application of *control theory* [33] will prove fruitful in adding further rigor to the design of distributed algorithms.

References

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing (STOC '95)*, pages 238–247, New York, May 1995. ACM.

[2] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *SPDT '98*, Aug. 1998.

[3] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of 1998 ACM Sigmetrics Interna-*

tional Conference on Measurement and Modeling of Computer Systems, 1998.

[4] R. H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, University of California, Berkeley, 1999.

[5] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *IOPADS '99*, May 1999.

[6] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance Over Wireless Links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, December 1997.

[7] K. P. Birman and R. Cooper. The ISIS project: Real experience with a fault-tolerant programming system. *Operating System Review*, pages 103–107, April 1991.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[9] D. Blackwell and M. A. Girshick. *Theory of Games and Statistical Decisions*. John Wiley & Sons, Inc., New York, 1954.

[10] 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 *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, July 1995.

[11] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet—A Gigabet-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–38, February 1995.

[12] W. J. Bolosky, J. S. Barrera, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. The Tiger Video Fileserver. `http://www.research.microsoft.com/research/os/bolosky/tiger/tiger.html`, 1995.

[13] E. A. Brewer. The Inktomi Web Search Engine. Invited Talk: 1997 SIGMOD, May 1997.

[14] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the 1994 International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[15] N. J. Carriero. *Implementation of Tuple Space*. PhD thesis, Department of Computer Science, Yale University, December 1987.

[16] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. P. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. In *JSPP'97 (9th Joint Symposium on Parallel Processing)*, Kobe, Japan, June 1997.

[17] S. Englert, J. Gray, T. Kocher, and P. Shah. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 245–246, May 1990.

[18] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, Stockholm, Sweden, June 13–17, 1983. *Computer Architecture News,* 11(3), June 1983.

[19] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):102–111, June 1990.

[20] H.-I. Hsiao and D. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[21] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

[22] T. Johnson. Designing a Distributed Queue. In *Proceedings of the Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 304–11, San Antonio, Texas, October 1995.

[23] V. Karamcheti and A. A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *22nd Annual International Symposium on Computer Architecture*, pages 298–307, Santa Margherita Ligure, Italy, June 22–24, 1995.

[24] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[25] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS VII*, pages 84–92, Cambridge, MA, October 1996.

[26] A. Mainwaring and D. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[27] R. Metcalf and D. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–403, July 1976.

[28] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Jan. 1997.

[29] S. Microsysytems. Whitepaper on NFS Version 3.0. `http://www.sun.com/software/white-papers/wp-nfs/`, March 1995.

[30] J. V. Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, New Jersey, second edition, 1947.

[31] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *ASPLOS VIII*, San Jose, California, 1998.

[32] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[33] O. Rubin. *The Design of Automatic Control Systems*. Artech House, Norwood, MA, 1996.

[34] F. B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[35] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the IPPS '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[36] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[37] M. Tremblay, D. Greenley, and K. Normoyle. The Design of the Microarchitecture of UltraSPARC-I. *Proceedings of the IEEE*, 83(12):1653–63, December 1995.

[38] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, USA, 1995.

[39] C.-P. Wen. *Portable Library Support for Irregular Applications*. PhD thesis, University of California, Berkeley, January 1996. Techreport UCB/CSD-96-894.