Coordination and Synchronization: Designing Practical Detectors for Large-Scale Distributed Systems

Indranil Gupta

Dept. of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801

indy@cs.uiuc.edu

Abstract

Online detectors form an important part of achieving coordination and synchronization in large-scale distributed applications running in peer to peer systems, the Grid, PlanetLab, and large-scale Enterprise like server farms. Detectors can be used to monitor the up/down status of hosts, the malicious behavior among processes, the availability behavior among hosts, and to estimate the number of hosts in a distributed system. We discuss a variety of existing online detectors for these different problems, with emphasis on practical solutions that satisfy two characteristics: they have been implemented and validated in experimental evaluation or practice, and they are based on ideas that are novel and have strong theory behind them. The goal of this article is to enable practitioners to understand these protocols so that they can be easily implemented or adapted for various distributed systems. This article aims to provide the starting researcher a start into, and a good feel for, the area of detectors, in order to enable further learning in this interesting area.

1 Introduction

Large-scale distributed systems such as PlanetLab [36], peer to peer systems (e.g., [48, 42, 39]), Grid networks [16], etc., have exploded in popularity and use in the past few years. It is well-known that such systems are failure-prone, i.e., "nodes" (client machines or computer hosts) can join and leave the system at will (a phenomenon called churn), and that messages can be dropped by the underlying network.

Several distributed applications have begun running atop such clusters, e.g., distributed computations, cooperative file sharing, multimedia and content streaming, resource discovery, application-level DNS, etc. In each of these distributed applications, to enable coordination and synchronization, the application need to keep track of the behavior of each individual node involved in the application. Intuitively, each node has individual "personalities" from the viewpoint of its cooperativeness or willingness to contribute to the overall good of the system. Thus it is important to keep track of the individual characteristics of these nodes in a distributed fashion.

On one side of the problem spectrum, at the most basic level, there are simple node-level that need to be detected. For instance, when a node fails (or joins the system), some other nodes that are currently in the system need to be made aware of this change in the *membership*. Similarly, some nodes may be maliciously modifying messages or deviating from the core protocols specified as a part of the system - it is important to detect (and then perhaps punish) such nodes.

At the other extreme, several applications need to detect system-wide properties. We consider two interesting classes of detectors falling at this end of the spectrum - one requires nodes to be aware of the approximate size of the system, i.e., number of non-faulty nodes currently in the system. In between these two extremes, some applications require to track the individual availability history of nodes, i.e., their up/down characteristics. This availability information can then be used for placing replicas (of files or services) so as to maximize availability of the service being replicated, for ensuring that multicast reliability at recipient nodes varies as a function of the node's availability, etc.

We broadly call the above problems of measuring node-specific or aggregated system-wide properties as the problem of *Detection*. There are a variety of detectors for distributed systems, and it is quite possible that a book-length article could be written covering these various detectors! In order to maintain brevity however, this article chose to focus on and discuss a "sliver" of detectors from across the spectrum of node-level to system-wide detectors.

Specifically, we will focus in this article only on failure-, Byzantine-, and availability-related characteristics of detectors. We will mention, at appropriate places, other detector classes that are not covered here and that the reader may be interested in reading up. The reader should use this article as a starting step to understanding more about the area.

Notice that the latter extreme of detection is related to "Statistics Collection" and aggregation [29, 44], but we are interested only in the actual availability-related or behavior-related characteristics of nodes, and not in collecting statistics that are specific to a particular application. In other words, most solutions to the Detection problems we will discuss can be used by a wide variety of distributed applications. Further, we hope this article will motivate the reader to look up existing literature on other problems like termination, deadlock detection, snapshots, reputation mechanisms, etc.

We focus on practical solutions that have the following two characteristics: (1) they have been implemented and validated in experimental evaluation or practice, and (2) they are based on ideas that are novel and have strong theory behind them. Our goal here is to enable and enhance understanding of such viable and practical solutions for practitioners so that these can be used in real systems.

Thus, this article considers four main classes of Detection problems.

1. *Crash Failure Detectors:* For each node, when the given node in the system crashes, other nodes that knew about this node should be informed about it.

2. *Byzantine Failure Detectors:* For each node, when the given node deviates from the specified application protocol behavior, other nodes that are non-Byzantine need to be informed of this.

3. *Availability Detectors:* For each node, the system (or a small set of other nodes) maintains information about the availability history of this node.

4. *System Size Estimators:* An initiator node (or all nodes in the system) needs to know about the approximate number of non-faulty nodes present as a part of its distributed group (or overlay graph). This could be either a one-shot or a continuous estimation problem.

Two points to note here. First, we are primarily interested in *fully distributed solutions* to these problems. That is, protocols that operate in a peer to peer fashion, without requiring a central server, are of the most interest to us. Second, we will rarely discuss the action taken by the application when such a detection is triggered. In some of the referenced papers, such reactive application behavior may be discussed. However, our discussion in this article attempts to present detectors in a modular fashion, i.e., so they can be used in a plug-and-play manner with a variety of applications.

While the main goal of this article is to make the reader aware of practicalities of Detection problems and solutions, where appropriate, we do highlight relevant theoretical results that form the context or tell us about the difficulties of a problem.

The rest of this article is organized as follows: Section 2 discusses crash failure detectors, Section 3 presents Byzantine failure detectors, Section 4 presents availability detectors, and Section 5 discusses system size estimators. We conclude in Section 6.

2 Crash Failure Detectors

Here, we consider failure detection of nodes under the *fail-stop* failure model. Under this model, either a node is non-faulty (or correct), or it has crashed. Any node can crash at most, and once it has done so, never executes any more instructions (i.e., it never recovers).

Crash failure detection is the core of all peer to peer systems and distributed systems that attempt to operate in non-centralized manner.

Before solving any problem (such as that of Crash Failure Detection), it is important to discuss what *system model* (i.e., assumptions) the problem needs to be solved under. Primarily, there are two types of system models for distributed systems:

1. *Synchronous System Model:* Each non-faulty node has a maximum known time bound on the time taken to execute any instruction. Further, there is a maximum known bound on the delay faced by a message sent by one non-faulty process to another non-faulty process. Examples of systems following this model are multiprocessor systems such as supercomputers. Notice that nodes are still allowed to fail in this model.

2. *Asynchronous System Model:* Unlike the above model, the asynchronous model imposes no limits on either the time taken by a non-faulty node to execute any instruction, or on message delays. In other words, messages can be delayed arbitrarily long, and nodes can be arbitrarily slow (without being faulty). Most practical networks follow the asynchronous system model, e.g., Internet, wireless networks, sensor networks, etc.

Now, Fail-Stop failure detectors are interested in two types of properties:

- Completeness: The percentage of failures that are eventually detected by all concerned non-faulty nodes.

- Accuracy: The percentage of detections that actually correspond to a node that has failed.

Notice that it is easy to trivially to guarantee either 100% Completeness (each node always consider all other nodes as crashed all the time) or 100%

Accuracy (each node never considers any other node as crashed at any point of time).

Chandra and Toueg [6] showed that it is impossible to guarantee both 100% Completeness and 100% Accuracy in an asynchronous system model. In the synchronous system model however, implementing a complete and accurate failure detector is straightforward - any one of the following detectors for asynchronous systems can be used, along with timeouts that are decided based on the message delay and instruction processing bounds.

In view of these impossibilities, most, if not all, distributed applications have come to expect 100% completeness (and thus probabilistic accuracy) from the underlying crash failure detector. This is because each crash of a node in a distributed application needs to be followed by a repair or recovery operation in that application - thus it is important to detect each failure, but it is alright to have mistaken detections. All algorithms we discuss below guarantee 100% completeness.

Chandra and Toueg were the first to present failure detectors with a view to solving consensus. Specifically, they provide a taxonomy of detectors in [6], including the weakest failure detector to solve consensus. There has been substantial work done in the theoretical community since then on failure detectors for a variety of system models, e.g., [7]. However, we preclude such papers (even though classical) since they either do not scale to large distributed systems with thousands of nodes, or they have not really been validated in practice.

For asynchronous systems, practical failure detectors for fail-stop failures tend to be of two types: (1) Heartbeating-based, and (2) Ping-based.

2.1 Heartbeating-based Failure Detectors

Each node n periodically (once every hb seconds) sends an "I am Alive" (heartbeat) message to a subset of other nodes in the system. Successive heartbeat messages are numbered with monotonically increasing sequence numbers so as to be distinguishable. Each other node that is aware of node n maintains the time since the last and latest heartbeat was received from node n. When this time crosses a prefixed timeout threshold (*timeout* seconds), the node n is marked as having failed.

First, this satisfies 100% completeness - once node n fails, it will stop sending heartbeats and since *timeout* is finite, eventually all previously sent heartbeats by n will be received and the timeout will expire (at any given recipient node). In practice though, the value of *timeout* is typically much larger than messages transmission delays; hence, the actual detection time is *timeout* seconds.

However, this algorithm does not guarantee accuracy, especially in an asynchronous network where heartbeat messages can be delayed arbitrarily long. This can cause another to timeout waiting for heartbeats from a (correct) node n, and consequently mistakenly mark n as crashed. Notice that the larger the value of *timeout* is compared to hb, the more is the accuracy of the protocol, i.e., the smaller is the false positive rate. However, larger *timeouts* also entail longer detection times, hence the value of *timeout* trades off between detection time and accuracy.

Heartbeat transmission can be implemented in one of two ways - *explicit* or *implicit*. Explicit heartbeat creates separate messages for heartbeats, while implicit heartbeat either piggybacks heartbeat messages atop application messages, or in some cases, uses application messages themselves as heartbeat messages[1]. For the rest of our discussion, we will assume explicit heartbeating, however our discussion applies to the implicit variety too.

There are several variants of heartbeat-based failure detectors - the difference between these is based on which is the "subset" of nodes receiving the heartbeats from the given node n. This choice is decided based on the *overlay*, or the *membership graph* (i.e., the graph defined by a node's neighbors - see below). Below, we describe several different types of such overlays, along with the associated heartbeat-based protocol.

Simple Overlays: The IBM SP-2 originally used a *ring*-based overlay, with nodes arranged in a virtual ring (with no necessary correlation to their actual locations). Each node merely sent heartbeats to its clockwise neighbor (in addition, the anticlockwise neighbor was also used to increase the fault-tolerance), and these neighbors would be the only ones to detect failure of this node. In a system with N nodes, the overhead of this scheme is $O(N)$ messages every hb seconds. The drawback of this scheme was that multiple

[1]We will ignore this last option mentioned since our goal in this section is to focus on application-independent protocols.

simultaneous failures could cause an unnecessarily long time for detection of failures, especially if a sequence of nodes failed in quick succession. Since the likelihood of this occurrence increases as the total number of nodes increased, the ring-based algorithm was not very scalable.

A different, simpler alternative to send the heartbeat to *all* other nodes in the system. While this was clearly more fault-tolerant than using the ring, this scheme has a very high overhead ($O(N^2)$ messages per hb seconds) and it could also have lower accuracy. Any slow node could mark a very large set of others nodes as faulty merely because it did not receive several heartbeat messages in a timely manner.

Gossip-style Heartbeating: Van Renesse et al [45] made the above all-to-all heartbeating model more accurate by having each node not directly send its heartbeats to every other node, but instead *gossip* the latest heartbeat counters for several other nodes. At any node n, gossiping basically entails randomly selecting a few other nodes periodically, and sending them the array of the latest heartbeat counters (from other nodes) known at node n.

Van Renesse et al showed that if all heartbeats could be included in each gossip message, and each node gossiped with a constant other randomly selected gossip targets every second (on average), it took $O(log(N))$ seconds for any node's updated heartbeat information to spread to all other nodes with high probability. Here, N is the number of nodes in the system. Thus, the timeouts could be set to be in this range (if one knew an upper bound on the value of N). Thus the failure detection times are small - this is because $log(N)$ is a small number, and grows very slowly, e.g., even for values of N up to 2^{32} (the number of possible IPv4 addresses), the value of $log_2(N) = 32$.

Distributed Hash Table-based Overlays (or Structured Overlays): Distributed hash tables (DHTs), also known as structured overlays[2], are overlays that follow a specific structure. For instance, the Pastry p2p overlay follows a hypercube-type structure, with nodes maintaining overlay "neighbors" based on prefix matches of id's assigned to nodes (the id's are assigned by hashing the node's IP address (e.g., by using SHA-1 or MD-5 etc.), but that is orthogonal to our discussion here.) In turn, each node merely sent heartbeats to its neighbors in this overlay.

Similarly, in other DHTs such as Chord, a heartbeat-style strategy was used for failure detection. Thus information about a node failure would propagate to its immediate neighbors - this in turn might cause these nodes to select other, "better" neighbors that were non-faulty.

Random Partial Membership Graphs: While DHTs such as Pastry and Chord follow a specific pattern of "neighbor" selection of nodes, in order to make resource discovery and file insertion operations very efficient, a separate class of overlays have been designed for other applications that do

[2]To be more precise, a structured overlay is the actual underlying overlay, while a DHT is layered atop this overlay and provides get- and put-style functionalities to an application. However, today, the two terms are often used as synonyms by several sections of the distributed computing community, and hence we treat "DHT" and "structured overlays" as synonyms in this article.

not primarily use the resource-discovery functionality. For instance, publish-subscribe and multicast applications often rely on the presence of a connected overlay graph among the nodes. Yet, the protocol attempts to achieve this by having each node only maintain a *small random subset* of other nodes in the system as its neighbors.

Below, we briefly discuss the core design of one such random partial membership graph system. The reader is encouraged to read up other algorithms in this class, such as T-Man [21].

SCAMP: SCAMP [17, 18] attempts to maintain a *uniform random* overlay graph among nodes, with each node maintaining $O(log(N))$ neighbors in this graph. This is achieved by the following mechanisms: (1) Each node n maintains a list of neighbors in the overlay, denoted as $Neighbor\_Set(n)$, as well as the list of other nodes than point to them (the in-neighbor list). (2) [Node Join] When a new node joins the system, it obtains at least c contacts (c is a fixed parameter), and forwards its subscription (joining) information to c of these nodes. A node n receiving a new joining node's information will include it in with probability $1/(1+|Neighbor\_Set(n)|)$; otherwise, it merely forwards this subscription to one of its neighbors, selected at random. (3) [Node Departure] A voluntarily leaving node n merely asks the highest-id c neighbors of itself to delete n from their neighbor lists. Every in-neighbor of n is asked to point to another of the previous neighbors of n (excluding the c selected above) - duplicate selections may be allowed.

While the basic SCAMP assumes voluntary departures only, each node does indeed send heartbeats periodically to all of its neighbors. This avoids a node from being partitioned (isolated) out from the network - when a node has not received *any heartbeats* from any other node, it knows that it is partitioned [3].

The authors show in [17] that this protocol ends up with each node having an expected $(c+1)log(N)$ neighbors, and that the distribution of neighbor selection is random (i.e., the probability distribution of the number of in-neighbors at a node has a small standard deviation).

It is easy to see how SCAMP can be extended to handle fail-stop failures - all neighbors of a given node would timeout waiting for a heartbeat and then execute actions similar to the voluntary unsubscriptions described above. However, it is not clear whether this would continue to maintain the uniform randomness of the overlay. Further, false positives could occur - any node missing a heartbeat would start propagating a failure notification, and a suspected node would then be forced to leave the group. This problem is addressed in the SWIM system discussed in Section 2.2.

2.2 Ping-Based Failure Detectors

Unlike heartbeat-based failure detectors, Ping-based failure detectors do not use any kind of heartbeat messages. Instead, each node n is periodically pinged by a subset of other nodes in the system. If the node is unresponsive,

[3]Note that this does not avoid a large subgraph from being partitioned out of the overlay!

then the pinging nodes could retry the pinging. If several retries do not lead to a response, the node n is marked as crashed. Below, we describe two such ping-based failure detectors - SWIM and CYCLON.

SWIM: The SWIM system [12] by Das et al has each node periodically (once every T seconds) select *one* other node (say n) uniformly at random from across the system and ping this remote node. If the remote node is unresponsive (T is assumed to be larger than the typical round-trip time in the system), then the pinging node may ask up to K (value fixed) other nodes to indirectly ping the node n and return replies (if any). If either the direct or any one of the the indirect pings results in a positive reply from n, then the pinging node takes no further action. However, in the absence of a response, the pinging node detects node n as crashed.

Clearly, this protocol satisfies 100% Completeness - a crashed node will eventually be picked as ping target by some node in the system, and be detected as failed. Further, the authors showed that this protocol has a constant failure detection time *on expectation*, e.g., for $K = 0$, the expected time between failure of node n and the *first* other node detecting this failure is $\frac{T}{1-e^{-1}}$ seconds. It is important to note that this time does not depend on the size of the system at all (which is a desirable and scalable property, especially in a really large distributed system)! Further, the authors show how to tune the value of K so as to obtain a tradeoff between the detection time, the false positive rate (the inaccuracy rate), and the overhead (messages per second per node). The reader is referred to [12] for more details.

CYCLON: CYCLON [46] is another membership protocol that attempts to maintain a uniform random membership graph while having each node maintain only a small number of neighbors. Briefly, each node maintains an *age* for each of its neighbors, denoting the time since that neighbor entry was created at node n. Each node periodically does the following - eliminate the neighbor with the maximum age, and exchange neighbor lists with this oldest aged neighbor. CYCLON then describes a specific way of updating the neighbor lists in order to maintain the uniform randomness of the overlay graph. However, notice that this selection of the oldest age neighbor is *implicit failure detection*, but in the heartbeat style. If this oldest neighbor does not respond, then it is deleted. Thus, failed nodes eventually disappear from neighbor lists. If the size of neighbor lists is $O(log(N))$, then the failure detection time is also $O(log(N))$, which is small!

3 Byzantine Failure Detectors

Unlike the fail-stop failure model discussed in Section 2, the *Byzantine* failure model specifies that nodes can behave in any arbitrary and perhaps malicious manner, i.e., a Byzantine-faulty node could deviate from the protocol specified by the application in arbitrary ways. For instance, it could execute instructions that are either unauthorized or do not result from the applying the specified protocol on its received messages, it could send messages with malicious intent or junk content, or claim to have received messages that in fact it never received. In short, the Byzantine model is the most general of all models of failure. Clearly, it encompasses and includes the a fail-stop failure model.

Yet, it is a very realistic model. Hosts whose security has been compromised, e.g., by viruses or worms, or by human hackers, etc., as well as a process based on buggy program code, all follow the Byzantine model. Thus, handling Byzantine failures is an important practical focus. The Byzantine failure model typically does not handle correlated failures, but instead individual node's malicious behaviors. This makes sense for some of the security issues addressed, e.g., virus or worm attaks, but not for others, e.g., buggy software or collaborating botnets (which may have correlated Byzantine nodes).

The traditional approach to handling Byzantine failures has, until very recently, been to *mask*, rather than detect, these types of failures. Most protocols for Byzantine fault-tolerance are replicated state machines, with a focus on solving problems such as atomic commit and consensus [6, 9, 14, 15, 24], i.e., where all the nodes need to agree on the value of a variable. These protocols assume that there are at most f faulty nodes in the system, and there are at least $3f + 1$ total nodes in the system (faulty or not). Several such protocols have been specified in theory [8, 38] and in practice [5, 47]. These protocols are designed to allow the non-faulty nodes to solve the agreement problem in the presence of up to f Byzantine nodes among them. The reader would be interested to know that it has been proved [27] that one cannot implement Byzantine fault-tolerant consensus when more than one-third the nodes are faulty, hence these protocols have "optimal" tolerance, in a sense.

While these protocols (especially Castro and Liskov's [5]) are highly practical and perform well in real systems, they are unable to tolerate more than f failures. Instead, if one used a Byzantine failure detector, the following advantages could be obtained [20]:

- More than f failures could be detected (and tolerated, if the application is equipped with mechanisms to respond to detected failures). In fact, there is no upper bound on the number of Byzantine nodes in the system.

- The common case (where all nodes are non-Byzantine) becomes very efficient w.r.t. performance metrics such as throughput, latency, scalability, etc. Simplicity of design is preserved since detectors are typically designed to fit in very modularly with the rest of the application.

- Many applications do not need to solve the consensus problem, and Byzantine failures are interesting in other problems that are not related to consensus. For these, applications require information about which nodes might be faulty. We remind the reader that one cannot implement Byzantine fault-tolerant consensus when more than one-third the nodes are faulty [27].

The same properties of Completeness and Accuracy discussed in Section 2 apply to Byzantine failure detectors (thus no failure detector can achieve both properties with 100% guarantee). Below, we briefly describe two systems - LOCKSS and PeerReview - that provide some semblance of Byzantine failure detectors. Besides these two systems, there are other systems that come close to providing a detector, but do not provide a fully specified one. Aiyer et al [1] provide a mechanism to monitor quorum systems so that an alarm is raised when failure assumptions are about to be violated. Intrusion detection systems of course work at the level of a single node. Reputation systems (see, e.g., [11]) monitor the behavior of nodes in a p2p system but do not provide a notion of detection of Byzantine failure.

Before we dive in to these systems, we note an important point - a Byzantine failure detector depends, to some extent, on the application itself, e.g., what is unacceptable behavior by a node. Yet, the LOCKSS system is generic enough in being applicable to any distributed storage solution, while PeerReview applies modularly to any distributed application which allows auditing actions on application logs.

LOCKSS (Lots of Copies Keeps Stuff Safe): The LOCKSS system by Maniatis et al [31] provides a protocol for maintaining consistency of replicas - LOCKSS is implemented in the context of a digital library archive, where archival units, or AUs, are the basic blocks that are replicated across multiple nodes. The challenge is the following - even though the AUs are immutable, attacks by either adversaries or bit-rot, may cause some of the replicas of the AU to become corrupted as time progresses. The goal of the LOCKSS system is to (1) maintain the correctness and consistency of these replicas, and (2) to enable detection of an ongoing attack, especially when a large number of replicas are in disagreement with one another.

LOCKSS meets the above challenges by (1) building a continuously-changing (churned) overlay among nodes, and (2) using this overlay to execute periodic polling on the replicas of the AU (to check for and correct their consistency). We do not describe here the intricate details of the protocol, viz., the actual quotas on how much of the list is churned for each of the above actions, etc. - the reader is encouraged to read [31] for all details and adversary attacks on the protocol.

In brief, the protocol works in the following manner. Each node n:

1. Maintains two types of neighbors - inner circle neighbors (more trusted) and outer circle (less trusted) neighbors. The inner circle at any time consists of a random subset of other nodes that have agreed with the recent *votes* of this node n. In addition to these two circles, node n maintains a list of *friends* - other nodes on whom it places a very high level of trust.

2. Initiates a *Voting procedure* periodically. This is done by querying the inner circle neighbors, each of which in turn nominates a few nodes for n's outer circle. n then chooses a small random subset from each

nomination and asks these nodes to vote. Each vote is classified as either "agreeing" or "disagreeing" with n's own vote. This calculation is based on the hash of the replica of the AU in question, i.e., the entire contents of the AU replica are hashed to generate a signature and this signature is matched. (In addition, the LOCKSS protocol also marks each vote as either valid or invalid, based on a proof of computational effort. For our purposes, it is suffices to know that an invalid vote will result in the offending voter being ignored and removed from the neighbors lists at n). Finally, if V total votes were requested and received, then (a) if the number of agreeing votes is at least $V - D$, then the poll was poll was successful and n retains its replica; (b) if the number of agreeing votes was no more than D, the poll was a failure and n repairs its replica (from a random disagreeing neighbor); (c) the number of agreeing votes is between D and $V - D$: in this final case, n raises an *alarm* (the effects of an alarm are described below).

3. *Churns* its neighbor lists. After each vote, inner circle neighbors are who disagreed or have not voted for a while are eliminated, a random subset of the remaining nodes is left in, a few random recently agreeing nodes (no voting) from the outer circle are brought, and finally a few random friends are brought in. The goal of this churning of neighbors is to ensure that malicious nodes do not manage to gain a foothold on the neighbor list of a node n for too long a time.

The authors of the LOCKSS system then show that under a variety of adversary attacks, if most of the replicas of the AU are good (resp. bad), then most polls will end successfully (resp., in failure). *However,* and most importantly, it takes a very long time for an AU with predominantly good replicas to transition to a state with predominantly bad replicas. Hence the *alarm* condition raised in the specification above will have enough time (and there will be enough alarms) to detect this shift. In the authors' words – "The rate at which at an attack can make progress is limited by the smaller of the adversary's efforts and the efforts of the victims." Put in another way, LOCKSS slows down the conversion of good replicas into bad replicas (which occurs due to the presence of malicious nodes) so much that victims (i.e., good nodes) are able to take corrective action in fixing the bad replicas. Thus, even a delayed and slow human response to such an alarm would restore the correctness of the system, since the adversaries are slowed down considerably by LOCKSS.

Notice that even though the above protocol does not explicitly *detect* Byzantine nodes in the system, it is able to detect *disagreeing* votes. In the case of alarms being raised for the AU, compromised nodes can be detected easily (via their proposed hashes for the AU) and thus repaired. Logs of the votes obtained can be used to detect faulty nodes (albeit perhaps with human involvement), and if a particular group of nodes happens to be raising alarms, a local spoofing alarms could be raised in order to audit local nodes in the neighborhood of those nodes.

PeerReview: The PeerReview system [20] shares some common design characteristics with the LOCKSS system designed above. However, unlike it, PeerReview provides for explicit Byzantine failure detection with interesting completeness and accuracy properties (see below). Specifically, PeerReview ensures that a correct node will never be declared as being faulty (assuming that the node is indeed responsive). *This is a major difference from the fail-stop failure detectors of Section 2.*

The PeerReview protocol has each node monitor application protocol-compliance of all other nodes in the group. Thus, it is potentially expensive and inefficient (it involves $O(N^2)$ messages in the system), however it is a good first-cut at this extremely different problem. Among the several assumptions made by PeerReview, the most important ones are: (1) Messages sent by correct nodes are eventually received by the recipient (if it is correct); (2) the application protocol for which compliance has to be checked is a replicated state machine [40].

First, each node n maintains a log of all its previous protocol actions, and uses this to sign messages. Top-level hashes of the log are taken periodically and on-demand - such *authenticators* are piggybacked on top of all messages sent out by n. The log, in other words, is maintained as a hash chain. All messages need to be acknowledged (acknowledgement messages also carry authenticators). Each message sent by n, besides the authenticator, also contains a short *proof* that the latest message is in fact the latest action in the local log. Finally, node n periodically forwards to other nodes the authenticators it knows for other nodes; this ensures eventual dissemination of any authenticator.

Second, each node n is periodically *audited* by other nodes j. Node j can show that n is faulty if either (1) it has an authenticator and a log both from n, both signed by n, but disagreeing with each other, or (2) a signed log segment from n that fails a conformance check. During the audit phase, node j can begin to *suspect* n if the latter is either unresponsive or non-compliant. Otherwise, node j performs a consistency check to see if the log matches the recent authenticators it has for n (this is for rule (1) above). Then, node j extracts all authenticators from the log segment and forwards them to all other nodes - this ensures eventual dissemination of these authenticators to all other correct nodes. Finally, j performs a conformance check (for step (2) above). This is perhaps the most computationally expensive operation in the protocol. j instantiates a local copy of the application state machine, replays all inputs form the log, and checks whether outputs match the ones in the log.

Notice that any deviation based on the above checks can be forwarded to other interested nodes, who can then verify for themselves whether node n is in fact faulty or not, either by repeating the checks for itself or by contacting node n directly to re-do the checks.

This helps PeerReview ensure a nice variant of the Accuracy property - *no non-faulty node will be suspected or detected by another non-faulty node.* Notice that 100% accuracy is impossible - faulty nodes can suspect non-faulty nodes at any time. The Completeness property is not guaranteed either, but an interesting variant of it is. While it is possible that a faulty node may in fact escape detection forever, it is true that if there is a faulty node in the system, at least one node will be detected eventually. Thus a finite number of bad nodes can affect the good nodes for only so long.

PeerReview has been implemented and found to perform well in practice - readers are referred to [20] for more details. However, at the time of writing this article, it still remains to be seen what alternative Byzantine failure detectors can be designed. Further, whether this detector class can be made scalable at all remains a million dollar question!

4 Availability Detectors

After having discussed detectors for *online* individual node-level characteristics (crash and Byzantine), we next transition to the problem of *availability detection*. The failure model considered here is the *crash-recovery* model, where a node can leave or fail away from the system, and then rejoin later with the same node identifier.

The availability detection problem is to estimate the *short-term* or *long-term* up/down characteristics of each node n. The earlier detectors informed other nodes of the immediately recent failure of a node n - that is not our goal here; instead, tracking the up/down characteristics of n is our goal.

Availability detection is an absolutely essential component in the design of many peer to peer storage systems, e.g., [4, 10]. In these systems, the availability histories of nodes are used to select the best set of nodes to hold replicas for a given object, in order to increase the system-wide availability of the object. In these systems, availability detection is sometimes tied to an *availability predictor*, which predicts the future availability of node n based on its history.

Availability detection is also useful in trying to satisfy reliability predicates, where the reliability of an application protocol (e.g., multicast) at a recipient node is tied to the availability of that node, e.g., [37].

Below, we describe different types of availability detection schemes. Notice that detection schemes typically have two sub-components: *who monitors node n*, and *how the availability history of n is maintained at other monitoring nodes.* We discuss both these issues below. Further, in cases below where availability prediction is possible, we briefly describe this as well.

4.1 Group-Based Master Detectors

The Total Recall system [4] uses a *master node* in a group (of replica-holding nodes) to detect the availability of the nodes holding replicas, and to maintain availability history, as well as to do availability prediction in order to select the best set of replicas. The master node is selected on a per-object basis, and it is responsible for monitoring (via pings or heartbeats) the availability of two types of nodes: *inode storage nodes* and *data storage nodes.* Highergranularity availability information is maintained for the former set of nodes (and those lost replicas repaired eagerly by the master), while the latter set has lower-granularity availability detection (and those lost replicas are repaired lazily).

4.2 Group-Based Distributed Detectors

Carbonite [10] and HBHC [41] each use more distributed schemes than the previous one, but once again, both these schemes work within small groups of nodes (holding replicas of a given object).

Carbonite's availability detection works by creating a spanning tree (of height $O(log(N))$) rooted at each node in the group, and containing other nodes at its leaves. The spanning tree is created using the routing algorithm of the underlying p2p DHT (distributed hash table). Each node periodically sends out heartbeat messages to its children, and the heartbeat is propagated down the tree to its leaves. If a heartbeat missed, the monitoring node triggers repair for every object stored on the node detected as being down. In a manner, this is indeed a crash-recovery protocol, but we include it in this section (rather than in Section 2) since it is used by Carbonite to measure the availability history individual nodes.

HBHC [41] is another system for replica maintenance. The availability monitoring in HBHC is also fully distributed within the replica group. In brief, each node periodically pings each other node in the group, i.e., it is an all-to-all pinging scheme. This information is also periodically disseminated to all other nodes in the group itself, using a gossip-style (epidemic-style) dissemination [13].

4.3 System-Based Detectors

AVCast [37] is a system that links the multicast reliability at recipient nodes to their availability. The availability monitoring is done on a system-wide basis, i.e., without assuming replica groups. Thus it is a general scheme. To start with, availability monitoring can be done either by having each node individually report its own availability, or by using the overlay structure itself to decide which nodes monitor the availability of a given node n. The former approach is infeasible since nodes can lie about their own availability, while the latter scheme does not generalize easily since in power-law overlays (e.g., Gnutella [48]), higher-degree nodes would have a higher monitoring overhead.

Instead, AVCast's detector says that a node m will monitor another node n if the condition $Hash(m, n) < K/N$, where $Hash$ is a consistent hashing function with range $[0, 1]$, m and n are id's of the nodes, K is a small fixed constant, and N is the approximate system size (a fixed quantity at all nodes). Note that if the actual system size stays within a constant factor of N, each node will have an expected $O(K)$ other *random* nodes monitoring its availability, via ping and reply messages. Besides load balance, this scheme is *verifiable*, i.e., any third node can verify (using the hash condition above) if two nodes m and n are in fact related by a monitoring relationship. Thus, it is very difficult for a node n to cheat others by either reporting a higher availability for itself, or by colluding with other nodes. [37] describes further optimizations in the algorithm, where the value of K is changed adaptively - the reader is encouraged to read the paper for details.

4.4 Types of Availability History

While the detectors of Sections 4.1-4.3 merely maintained a straightforward history of the availability of a given node n, and calculated its availability as the average of all previous availability-test points (i.e., times at which the availability of the node was explicitly measured), other approaches to *maintaining history* are possible. [35] and [37] discuss some of these very well in the context of the goal of *availability prediction*, and we describe some below. Notice that most of these history-maintenance schemes can be used orthogonally along with the availability monitoring schemes above. However, we do not discuss such integration issues here.

RightNow: This is just the current up/down status of the node.

Aged: [37] uses an aged detector, where the last k availability tests on a node are weighed in an aged manner, with more recent availability tests weighed exponentially heavily compared to older tests. This aged equation is used to estimate the availability probability of node n. This aging rule is similar to the aging-based prediction of run-times of tasks in Operating Systems.

SatCount: In this [35], the availability of a node is marked as one of 4 values (using a 2-bit counter), based on its history. These values are -2 (Strongly Offline), -1 (Weakly Offline), +1 (Weakly Online), +2 (Strongly Online). This categorization is based on the results of the past k availability-testing points for node n.

de Bruijn Graph-based: For each node, the last k points of availability testing are maintained, with the most recent tests in the lowest significant bit. A left-shift operation is done with each new test. Using this as a basis, a state machine based on a de Bruijn graph can be set up among the 2^k possible availability states for node n (for the k-bit availability history). In a de Bruijn graph, each of the 2^k states is linked to lead into the 2 other states obtaining by left-shifting it. [35] then describes how to predict availability of a node n based on this – either by following the most likely path from the current availability state, or by following multiple paths, or by using a linear predictor (this work best for short-term-stable availability behavior) which are based on digital signal processing approaches, and finally a hybrid detector that combines all the above using an adaptive tournament scheme. Readers are encouraged to read [35] for more details. Using availability traces collected from two different clusters, the authors showed that in practice, the hybrid detector and predictor work very well for home and office clusters, and moderately well for geographically distributed clusters like PlanetLab.

5 System Size Estimators

Finally, we move to detecting system-wide properties related to failures. Specifically, we discuss different approaches to solving the *System Size Estimation* problem in large-scale distributed systems. Informally, the problem involves finding the "current" (at initiation time) number of non-faulty processes present in a distributed system, given that nodes can join and leave at any point of time. First, notice that an accurate estimate is impossible to achieve - messages have non-zero latencies, and departure or failure of even a single node, immediately after its last message with respect to the estimation protocol, will lead to an inaccurate estimate (and this is likely to occur in large-scale distributed systems).

Such estimation protocols are absolutely necessary in many distributed system, e.g., p2p overlays whose design depends on the value of system size N [19, 33], nodeID assignment schemes [32], for estimating the latency of lookups in some $log(N)-$ p2p overlays. Finally, estimated system sizes can be used to monitor and audit performance of distributed applications, e.g., on PlanetLab, as well as for dynamic partitioning of Grid applications.

Like prior detection protocols, we desire our estimation protocols to be scalable, efficient, fault-tolerant, and practical. In addition, increasing the (probabilistic) accuracy is an important goal. The estimation problem comes in two flavors - *one-shot* detection involves a one-time estimation of system size, while *continuous* detection involves continuously keeping the latest estimate of the system size. Accuracy can then be defined as either the root mean square of the error between the estimated system size and current system size, and/or as the standard deviation of these errors. The former metric measures how *close* the estimate is to the actual size; the latter metric measures how *consistently* the estimated size shadows the actual size.

Protocols for system size estimation come in two varieties - *active protocols* and *passive protocols*. Active protocols need to be initiated by a single node, and involve passing messages around inside the group until the initiator receives back enough responses or information to draw an estimate. This style of protocol is thus a one-shot solution, but can be repeated for a continuous implementation.

Passive protocols, on the other hand, do not involve actively exchanging any estimation messages. Instead, these protocols attempt to *snoop* on messages being otherwise sent by the application or by a membership protocol (such as the ones discussed in Section 2) in order to obtain an estimate. They are by nature continuous estimators.

Below, we discuss a small subset of active and passive estimation protocols. Following the theme of this paper, we choose only protocols that are the most practical, i.e., are implementable and have the least assumptions to hinder their transition into practice.

5.1 Active Estimation Protocols

Bawa et al and Sample&Collide: Both Bawa et al [3] and Massoulie et al [34] use the birthday paradox to estimate the system size. These protocols initiate a random walk within the distributed system - each node uses its neighbor information (provided by any of the group membership protocols such as the ones discussed in Section 2). If the number of nodes in the system is N, it takes an expected number of $\sqrt{2N}$ steps to get back to a node that was already traversed by the random walk. Based on this, a system size estimate is drawn.

Aggregation-Based Protocols: Several aggregation protocols have been proposed for distributed system. These include protocols to calculate sum, average, min, max, etc., of a set of values provided by the nodes in the system. Jelasity et al [22] use one such aggregation protocol to derive an estimation protocol. Basically, their protocol works as follows: once the protocol is initiated, each node keeps an estimate of the current size of the system. At node n, this value is initialized to 1 when the initiating message received. Then, periodically, node n exchanges its value with one neighbor chosen at random, and replaces its current estimate with the average of these two estimates. The authors of [22] then show that the estimate converges in time that is logarithmic in the group size. Several other aggregation protocols such as those by Kempe et al [23] could also potentially be used to derive a system size estimate similarly.

Hops Sampling: This scheme [25, 26] involves disseminating a *gossip* message (also called *epidemic* message) into the group, and measuring the average latency of the receipt times of this gossip. Since the dissemination latency of a gossip varies logarithmically with the system size, an estimate can be derived. The basic gossiping model works as follows: once a gossip message is received at node n, this node then periodically (once every T seconds) selects a fixed constant number of gossip targets (nodes) at random and sends them copies of the gossip. In addition, the Hops Sampling approach carries the *hopcount* variable (initialized to 0 by the initiating node); when a node n first receives the initiating message, it notes the hopcount, increments it by 1, and then starts gossiping the initiating message with the new hopcount piggybacked atop it. Finally, after $O(log(N))$ rounds, the initiating node queries a small subset of nodes in the system to sample their hopcounts, relates this to $log(N)$, and obtains a system size estimate. The latency of this protocol is also logarithmic in the system size.

Comparing the above three approaches: Le Merrer et al have compared the above three active approaches quantitatively via simulations [28]. They found that using aggregation (with estimates over last 50 rounds) provides the best accuracy, while Hops Sampling (with the estimate averaged over last 10 runs) comparatively provides lower accuracy. Admittedly, this particular comparison is somewhat unfair as it lacks a common baseline across the algorithms (e.g., the number of messages exchanged, etc.), but it is clear that Hops Sampling uses significantly fewer messages than aggregation, while Sample&Collide uses the least messages and has somewhat middling accuracy. Overall, the comparison does show that these different active protocols define an overhead-accuracy tradeoff. This opens the door for the design of adaptive estimation protocols, e.g., try to achieve a given level of accuracy while trying to stay within an overhead budget.

Other Active Estimators: Awerbuch and Scheideler [2] assign special id's to nodes, and organizes them in a hierarchy in order to enable estimation. Malkhi and Horowitz [30] use a ring-based algorithm for estimation, however this scheme could have a very high error, unlike the above schemes. Finally, several systems have been proposed for estimating the size of p2p overlays, e.g., Stutzbach and Rejaie's crawler-based approach [43].

5.2 Passive Estimation Protocols

Passive protocols for size estimation do not initiate one-shot runs of the protocol. Instead, they snoop on application or membership protocol messages in order to estimate the system size. Further, this class of protocols enables *each and every node* in the system to have an estimate, without restricting this knowledge to a privileged initiating node.

The Interval Density Scheme [25] is one such passive estimation protocol that works by snooping on the messages passed along by a gossip-style membership protocol such as the one by van Renesse et al [45] (discussed in Section 2). Basically, given a membership protocol (such as [45]) that enables each node n in the system to eventually (and perhaps quickly) learn information about the id or IP address of each other node joining into, or failing or departing from the system, the node n can estimate the system size *by only remembering a small fraction of these node ids.*

Each node then uses a consistent hash function (e.g., one based on SHA-1 or MD-5) to hash a heard-of node IP address into the real interval $[0, 1]$. In a nutshell then, node n is interested in only those other nodes whose IP addresses hash into a sub-interval I of the interval $[0, 1]$. If I is of size $O(K/N)$, where K is a constant, and N is the (approximate) system size, then the memory utilization at node n due to this estimation protocol is merely $O(K)$. Further, using snooping on the gossip-style membership protocol, the time for a node join or failure to reflect at the estimate at all other nodes is $O(log(N))$. Finally, the inventors of this scheme showed that if suffices for K to be $O(log(N))$ to derive an accuracy of the protocol that goes to 1 as the actual system size increases towards infinity.

[25] describes several ways to adjust both the size and the center-point of the interval I at node n so as to obtain an accurate detection - the reader is encouraged to see the referenced paper for more details.

Active vs. Passive Approach: The active Hops Sampling approach was compared against the passive Interval Density scheme in [26]. Both these algorithms are available as part of an open-source software called *Peer-*

Counter [25]. Overall, if the group size is more or less static, the passive approach yields a better accuracy. If the group size is highly dynamic, the algorithms perform comparably when one considers the root mean square error. However, the passive scheme has better performance w.r.t. the standard deviation of these errors, i.e., it is able to shadow the variation of system size better.

6 Summary

In this article, we have seen online detectors for several types of problems in large-scale distributed systems. We have seen (1) Heartbeat- and ping-based detectors for crash failures, (2) Implicit and explicit Byzantine failure detectors, (3) Master-based, Group-based and Fully Distributed Availability monitors, and (4) Active and Passive System Size Estimation Schemes. Our focus was on approaches that were practical, yet novel at their core. This continues to be a flourishing area of research, with ideas being implemented in a variety of real systems.

References

[1] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. ACM SOSP*, pages 45–58, 2005.

[2] B. Awerbuch and C. Scheideler. Robust distributed name service. In *Proc. 3rd IPTPS*, 2004.

[3] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani. Estimating aggregates on a peer-to-peer network. Technical report, Stanford Univ., 2003.

[4] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total Recall: System support for automated availability management. In *Proc. Usenix NSDI*, 2004.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

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

[7] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proc. 30th International Conference on Dependable Systems and Networks (ICDSN/FTCS)*, 2000.

[8] B. Chor and C. Dwork. Randomization in byzantine agreement. *Advances in Computing Research*, 5:443–497, 1989.

[9] B. Chor, M. Merritt, and D. B. Shmoys. Simple constant-time consensus protocols in realistic failure model. In *Proc. 4th ACM PODC*, pages 152–160, 1985.

[10] B.-G. Chun and et. al. Efficient replica maintenance for distributed storage systems. In *Proc. Usenix NSDI*, pages 45–58, 2006.

[11] E. Damiani and et al. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proc. 9th ACM CCS*, 2002.

[12] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly-consistent Infection-style process group Membership protocol. In *Proc. IEEE DSN*, pages 303–312, 2002.

[13] A. J. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proc. 6th ACM PODC*, pages 1–12, 1987.

[14] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *EDCC*, pages 71–87, 1999.

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

[16] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 2001.

[17] A. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, Feb. 2003.

[18] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. SCAMP: peer-to-peer lightweight membership service for large-scale group communication. In *Proc. 3rd NGC*, pages 44–55. LNCS 2233, Springer, 2001.

[19] P. B. Godfrey and I. Stoica. Heterogeneity and load balance in distributed hash tables. In *Proc. IEEE Infocom*, 2004.

[20] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *Proc. Usenix HotDep*, 2006.

[21] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay toplogy management. *Self-Organising Systems: ESOA*, LNCS 3910:1–15, July 2005.

[22] M. Jelasity and A. Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proc. 24th ICDCS*, 2004.

[23] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *Proc. 44th IEEE FOCS*, 2003.

[24] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[25] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. Demers. Decentralized schemes for size estimation in large and dynamic groups. In *IEEE NCA*, 2005.

[26] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. Demers. Active and passive techniques for group size estimation in large-scale and dynamic distributed systems. Manuscript currently under preparation, 2006.

[27] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM TOPLAS*, 4(3):382 – 401, Jul 1982.

[28] E. Le Merrer, A. M. Kermarrec, and L. Massoulie. Peer to peer size estimation in large and dynamic networks: A comparative study. In *Proc. 15th IEEE HPDC*, pages 7–17, 2006.

[29] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acqusitional query processing system for sensor networks. *ACM TODS*, 2005.

[30] D. Malkhi and K. Horowitz. Estimating network size from local information. *ACM Information Processing Letters*, 88(5):237–243, 2003.

[31] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, 2005.

[32] G. S. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *Proc. ACM PODC*, pages 197–205, 2004.

[33] G. S. Manku, M. Bawa, and P. Raghavan. Symphony:distributed hashing in a small-world. In *Proc. 4th USITS*, pages 127–140, 2003.

[34] L. Massoulie, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh. Peer counting and sampling in overlay networks: random walk methods. In *Proc. ACM PODC*, 2006.

[35] J. W. Mickens and B. D. Noble. Exploiting availability prediction in distributed systems. In *Proc. Usenix NSDI*, pages 73–86, 2006.

[36] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *HotNets-I*, 2002.

[37] T. Pongthawornkamol and I. Gupta. Avcast : New approaches for im-

plementing availability-dependent reliability for multicast receivers. In *Proc. IEEE SRDS*, 2006.

[38] M. O. Rabin. Randomized Byzantine generals. In *Proc. 24th IEEE FOCS*, pages 403–409, 1983.

[39] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, 2001.

[40] F. Schneider. The state machine approach: A tutorial. Technical Report TR86-800, Cornell University, 1986.

[41] T. Schwarz, Q. Xin, and E. L. Miller. Availability in global peer-to-peer storage systems. In *Proc. WDAS*, 2004.

[42] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM Conference*, pages 149–160, 2001.

[43] D. Stutzbach and R. Rejaie. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. In *Proc. IMC*, pages 49–62, 2005.

[44] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalabel technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[45] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. Middleware '98*, pages 55–70. Springer, 1998.

[46] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[47] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proc. ACM SOSP*, pages 253–267, 2003.

[48] The Gnutella protocol specification. http://www9.limewire.com/.