PuppetCast: A Secure Peer Sampling Protocol

Arno Bakker and Maarten van Steen
Department of Computer Science
Vrije Universiteit Amsterdam
The Netherlands
{arno,steen}@cs.vu.nl

Abstract

PuppetCast is a protocol for secure peer sampling in large-scale distributed systems. A peer sampling protocol continuously provides each node in the system with a uniform random sample of the node population, and is an important building block for gossip-based protocols for information dissemination, aggregation, load balancing and network management. Existing peer sampling protocols are either very vulnerable to attacks by malicious nodes, do not scale to large systems or provide only a static sample of the population. PuppetCast continues to operate when 50% (or more) of the nodes are acting maliciously, is shown to scale to systems of significant size and continuously provides new samples.

1. Introduction

Gossip-based protocols have many applications in large-scale distributed systems. They can be used for information dissemination [5], aggregation [9], load balancing [7], network management [16] and others. A *Peer Sampling Service*, identified as an important building block of gossip-based protocols [10], continuously provides each node in the system with a uniform random sample of all nodes in the system. Contacting a truly randomly selected node greatly improves the effectiveness of gossip-based protocols, making a good peer sampling service essential to their success.

Several implementations of peer sampling services exist [10, 3, 2]. Only one of these implementations, Brahms, can currently resist attacks by more than a few malicious nodes and scale to large networks of nodes [3]. However, Brahms provides each node with a static sample of the population, and was shown so far to resist only 20% malicious nodes when there is churn. In this paper we present *PuppetCast*, a secure and scalable peer sampling protocol that continues to operate when 50% (or more) of the nodes are actively attacking their peers and provides a new sample of the system every cycle of the protocol. This means that with PuppetCast every node in the system will expeditiously receive the address of every other node in the system, a property which a number of higher-level gossip-based protocols depend on (e.g. [9]).

PuppetCast builds on the work of Jelasity *et al.* [10] who propose to implement peer sampling for gossiping via a protocol that itself is gossip-based. The basic idea is that each node maintains a small partial view of the membership of the network. Periodically, each node selects one of the nodes in its partial view and sends that node its current partial view. In reply, the contacted node sends its partial view back. Both nodes then combine their own view and the received view into a new partial view of the network by means of a *view merge function* that overwrites their old one.

It has been shown that this method of gossiping and merging partial views leads to a situation in which each node has a partial view that is a good random sample of the node population (for a specific class of view merge functions) [10]. Furthermore, this method is robust against failures of nodes and point-to-point connections between nodes and thus is able to function in dynamic network conditions.

However, this implementation of peer sampling is also extremely vulnerable to attacks by malicious nodes. For example, it takes only a small group of malicious nodes to poison the partial views of all other nodes, precisely because of the epidemic nature of gossip protocols. If a small group of malicious nodes continuously advocates the same partial view, the partial views of all other nodes in the network quickly converge to that view. The nodes then no longer see a proper random sample of the network. Worse still, the graph formed by the nodes and the links in their partial views will have come to resemble a star topology. If the central nodes in this topology are also malicious and jointly leave the system, the graph disintegrates into a set of disconnected nodes and the peer sampling process stops [12].

The root cause of this vulnerability is that *malicious nodes control the partial view they send to others*, and it

is very hard for the receiving nodes to detect their malicious intent in time. PuppetCast removes the vulnerability by denying nodes any choice in the partial view they provide to others. In particular, PuppetCast replaces a node's single partial view with an *internal view* that is used by the node to select which node to exchange views with, and an *external* view that is sent to others. This external view of a node is assigned by a *trusted authority*. During view exchanges nodes will accept an external view from a node only when it has been assigned to that node by this trusted authority (using public-key cryptography). As malicious nodes can no longer choose what node addresses to include in the view they send, fast spreading attacks such as just described are no longer possible in PuppetCast.

The trusted authority introduces a central component in the system, responsible for handing out random samples of the population to nodes. We show that this central component is neither a performance bottleneck nor a single point of failure, unless targeted specifically by denial-of-service attacks. During other attacks, PuppetCast puts only a light load on the trusted authority, even in dynamic network conditions. In particular, even when 50% of the nodes is malicious and churn is high, a single trusted authority server can support a system of 100,000+ nodes. Furthermore, the functionality of the trusted authority can be partitioned over multiple servers which can, in turn, be replicated to improve fault tolerance. PuppetCast can also operate without a trusted authority in a degenerative state.

The remainder of this paper is organized as follows. Section 2 describes the PuppetCast protocol. In Section 3 we present the threat model we assume, identify the attacks it enables, and discuss how PuppetCast withstands these assaults. Section 4 analyzes the performance of our solution in two experiments. In Section 5 we discuss related work and we conclude in Section 6.

2. PuppetCast

To maintain the properties of a peer sampling service, the trusted authority must assign external views that are uniform random samples of the node population. To this extent, the trusted authority securely maintains a (more or less) complete membership of the network. When a node joins the system it registers at the trusted authority and gets assigned an external view. With this external view the node bootstraps its internal view and commences gossiping. As the external views are random samples of the node population, the peer sampling process with PuppetCast works as before.

In the case where few nodes join and leave the system the external views remain usable for a long time. As a result, other nodes need not refresh their assigned external view, implying there is only a light load on the trusted authority. To support dynamic network conditions where nodes frequently leave, PuppetCast introduces the concept of a *death certificate (DC)*, which is a statement by a node, say A, that it is leaving the system. A sends its DC to the trusted authority, and to each *publishing node* P_j that has A in its external view. Each P_j will subsequently extend its external view with A's DC when sending that view to others. The death certificates mechanism thus extends the lifetime of external views, which therefore do not need to be refreshed by the trusted authority, thus reducing its load in changing conditions.

The death certificates mechanism does not cover the cases where nodes crash unexpectedly, or where malicious nodes refuse to send death certificates themselves when leaving, or malicious nodes not sending them along with their external views. To handle these cases PuppetCast uses mandatory periodic reregistration, requiring a node to refresh its external view periodically.

2.1. The protocol

We consider a collection of nodes where every node can communicate with every other node via a network such as the Internet. Each node has a unique public/private key pair that has been certified by a *certification authority* [14]. The public key (or secure hash thereof) of this pair acts as a unique *node ID* for the node.

When a node A wants to participate in the peer sampling service it contacts the PuppetCast *bootstrap service (BS)* which acts as the trusted authority, as shown in Figure 1. The node authenticates itself to the BS using its key pair and a standard identification protocol [14] and sends it a `register` message. If the node's key has been certified by an accepted certification authority, the BS registers the node as a member of the peer sampling service. Registration of node A consists of storing its node ID, network address and an expiration time T_X in the BS database. From this membership database, the BS then takes a random sample of d (node ID,network address) pairs (d is typically 20). To this list it adds the node ID and expiration time T_X of node A and digitally signs this structure to create an external view for A. If the BS encounters expired entries in the database, it removes them. A node never appears in its own external view.

Once A has received its external view, it contacts all of the nodes $C_i (i = 1 \ldots d)$ listed in the view to register itself as a *publishing node* for C_i. Node A and its so-called *client node* C_i run the identification protocol to authenticate each other, and A sends an `I-am-publisher` message containing its external view to prove it has been assigned to be publishing node for C_i. C_i verifies that A's external view is valid; that is, it is signed by the bootstrap service, the contained node ID is that of A, and that the expiration time T_X

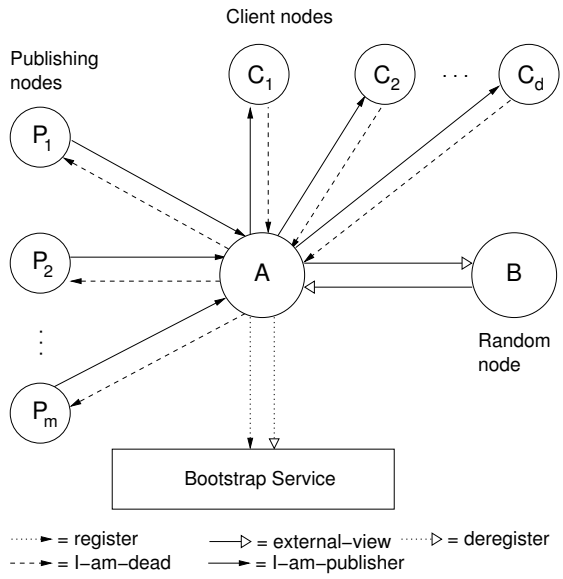

Figure 1. Overview of PuppetCast

has not yet passed. If the received view is valid, C_i records A as one of its publishing nodes by storing its node ID, network address and the expiration time T_X of the view.

After A has contacted its client nodes, it copies the list of (node ID, network address) pairs to its internal view. Next, it selects a random pair from this internal view, for example, the pair identifying node B. Node A then contacts node B and they authenticate each other, after which A sends its external view to node B and *vice versa* in an `external-view` message. Both nodes verify that the external view is valid. If the received view proves correct, each node merges the external view with its internal view using a view merge function called `Zipper+View0Init` to create a new internal view. If A fails to contact B, it removes B's address from its internal view. Subsequently, node A waits T_C seconds (the *cycle time*) and randomly selects another peer from its internal view to exchange views with, starting a new cycle of the protocol.

2.2. View refreshes and nodes leaving

Node A continues to do so until either its external view expires or it decides to leave the peer sampling service. When its external view expires, node A recontacts the bootstrap service, and sends its old external view. The BS then verifies that the external view has expired and creates a new external view for A. When node A decides to leave the peer sampling service, it first creates a death certificate for each of its publishing nodes $P_j (j = 1 \ldots m)$. A death certificate for P_j consist of A's public key, P_j's node ID and the expiry time T_X of P_j's external view (to limit the DC's lifetime). It is digitally signed by node A using its key pair. Second, A sends the relevant death certificate to each publishing node in a `I-am-dead` message. When a publishing node receives the DC, the node checks if it is valid and, if so, stores it. A death certificate is valid from the perspective of a publishing node when it contains its node ID and the correct expiry time of its current external view. Finally, the leaving node A deregisters from the bootstrap service with a `deregister` message.

The next time a publishing node P_j of A sends its external view to another node B it includes A's death certificate. Upon receipt, B checks that the external view and the death certificate are valid. A death certificate for A is valid if it is signed by A, sent by P_j and the timestamp T_X equals that of P_j's external view. If valid, B removes the entry for A from the external view before merging it with its internal view. If P_j sends along death certificates which are not valid its external view is ignored.

When publishing nodes have received death certificates from more than half of the nodes in their external view, they need to recontact the bootstrap service. Before doing so, they wait a random amount of time to negate a certain denial-of-service attack, as we explain below. After this time has expired, they contact the BS and pass their external view and associated death certificates. The BS checks if it was time to refresh and if so, creates a new external view for the publishing node.

If at one point in time, node A discovers that none of the nodes in its internal view are responding, it recopies its external view to its internal view. If afterwards it still cannot reach any node it contacts the BS to obtain a fresh external view. The BS creates one, but records that node A made an unwarranted request in this new external view. A is allowed to make only a few such requests in a certain period.

2.3. View merging

The `Zipper+View0Init` view merge function merges the internal view of node A with the external view of node B as follows. The first entry in the new view of A is filled with the address of B, if and only if A initiated the view exchange with B. The subsequent entries are copied from the internal view or external view alternatingly, what we refer to as a zipping method. Whether the merge starts with the first entry from the internal view or external view is chosen randomly. If one view contains too few entries, the new view is completed with entries from the longer one. If B initiated the exchange of views, A does not use B's address but merges the views via a pure zipping method. Finally, the new view becomes the new internal view for A. Note that the node sending the external view cannot

control the contents of the receiving node's internal view other than through the external view it sends, which is composed by the trusted bootstrap service, making this a safe view-merge function.

2.4. Bootstrap service implementation

The bootstrap service can be implemented by one or multiple servers depending on the number of nodes participating and the fault-tolerance requirements. The workload can be partitioned securely over multiple servers by using the certified keys of the nodes. Each server is then made responsible for an equal part of the key space, and a node must contact the server in whose partition its certified key falls. The server can check whether a node contacted it rightfully after the authentication protocol has run, ignoring illegal requests.

To ensure that nodes in different partitions meet each other, a bootstrap server exchanges samples from its database with the bootstrap servers of the other partitions. It then merges these samples into its so-called *trusted view*. When creating external views, it takes one part from its own database and one part from its current trusted view. Server replication can be used to make the bootstrap service more robust.

The membership database that the BS maintains needs to be more or less complete to be able to draw proper random samples of the population. It does not have to be completely accurate. If a node is no longer registered due to an error, it can still function. As long as its external view is valid it can initiate exchanges with other nodes, and as long as it appears in external views others will initiate exchanges with it. Hence, the system can also operate without the BS for a period of time.

3. Threat model and attacks

We assume the following threat model: Malicious nodes will attempt to interfere with peer sampling by (1) sending valid protocol messages but with malicious contents or (2) refusing to send required protocol messages, feigning a crash. We do not consider network-level denial-of-service attacks against other nodes or the bootstrap service. We initially assume that at most 50% of the nodes that can join the peer sampling service are malicious.

We discuss the attacks possible against PuppetCast by considering the harm each message in the protocol can do when (1) sent maliciously, (2) not sent or (3) when malicious nodes pretend they did not receive it. We consider `register`, `I-am-publisher`, `external-view`, `I-am-dead` and `deregister` messages in turn.

The first attack is nodes sending malicious `register` messages to the bootstrap service. The correct functioning of PuppetCast depends on the bootstrap service containing a largely accurate view of the node membership. In a Sybil attack [6] malicious nodes would try to poison this membership view by creating many extra node identities and registering as these nodes at the BS. As a result, all external views handed out would eventually contain just addresses of malicious nodes. To counter this attack, PuppetCast requires that an identity is issued by a trusted certification authority. This solution is generally recognized as one of the few effective ones against a Sybil attack [4].

Not sending `register` messages or refusing the sent external view has no effect. It either means the malicious node is not registered or that the node does not receive a new external view to communicate to others.

3.1. I-am-publisher messages

The second attack we protect against is malicious nodes not registering themselves as publishing nodes with their well-behaving client nodes. This attack is equivalent to the attack where malicious nodes do not piggyback the death certificates they received from good client nodes. The result of these attacks is that good client nodes cannot convey to other nodes that they left the system. This implies that nodes that receive an external view from malicious nodes will merge the addresses of these client nodes into their internal view thinking they are still alive. Consequently, the recipient nodes will continue to contact these client nodes, generating useless, but harmless network traffic. The amount of traffic generated depends on how many nodes leave the system (i.e., the *churn rate*) and the number of malicious nodes. In our maximum threat model of 50% malicious nodes, as much as 50% of the views exchanged may contain addresses of dead nodes.

PuppetCast combats this attack by putting an expiry time T_X on external views. This expiry time requires all nodes to periodically reregister at the bootstrap service. The new external views that reregistering nodes receive do not contain addresses of benign dead nodes, thus preventing the attack from affecting the system over long periods of time. The expiry time can be adjusted to the churn rate (as observed by the bootstrap service through the death certificates it receives). In this way the bootstrap service can ensure that nodes refresh more frequently when churn is high.

Malicious `I-am-publisher` messages can be directly spotted as they do not contain a signed external view that identifies the sender as an assigned publishing node. Dropping received `I-am-publisher` messages is equivalent to a malicious node not sending death certificates and is discussed below.

3.2. External-view messages

The third attack that is evidently covered is malicious nodes sending external views with malicious contents, e.g., containing IDs and addresses of fellow malicious nodes. This attack is not possible because nodes accept only external views signed by the trusted authority.

The fourth attack that does not harm PuppetCast is malicious nodes refusing to participate in the basic view exchange. PuppetCast handles this case by simply ignoring nodes that fail to respond in the correct way and select a new random candidate in the next cycle. Dropping received external view messages has no effect.

3.3. I-am-dead/deregister messages

The fifth attack is complementary to the second attack and entails malicious nodes not sending death certificates when they leave. Hence, their address will continue to be merged into the internal views of other nodes. As in the second attack this will generate fruitless network traffic, as the other nodes try to contact these dead nodes. This fifth attack is a double-edged sword, however. The fruitless traffic generated is directed at the hosts the malicious nodes ran on, which have to process it, making it an unattractive attack outside botnets [15]. PuppetCast has no special protection against this attack, other than that its periodic reregistration will weed out the dead malicious nodes from the external views. In a sustained attack, however, malicious nodes could simply reregister when their registration is about to expire and play dead the rest of the time.

The sixth attack is all malicious nodes leaving at once and sticking to the protocol for leaving. This means that all malicious nodes send death certificates to their publishing nodes and deregister at the bootstrap service. The latter causes a direct spike in the load on the bootstrap service. This is equivalent to a direct network-level denial-of-service attack by malicious nodes, as is the attack in which all malicious nodes try to register at once. We do not consider these denial-of-service attacks in this paper.

We do offer protection against some side-effects. The malicious nodes sending death certificates at a massive leave may cause a well-behaved publishing node's external view to become unusable (if more than half of its client nodes declared itself dead). In our threat model of 50% malicious nodes, which would all leave in this attack, it is likely that all of the remaining external views become unusable as they contain a random sample of the population, and 50% of it is leaving. When its view becomes unusable, PuppetCast requires that the node obtains a new view. However, to prevent overloading the bootstrap service in this case, each node waits a random amount of time before reregistering, thus dividing the load over time. So malicious nodes cannot cause other nodes to contact the bootstrap service *en masse*.

There is another case where PuppetCast offers some protection against overloading of the bootstrap service. Malicious nodes may contact the bootstrap service too often for a refresh of their external view. This is a denial-of-service attack that will swamp the service's download links and use up processing capacity. PuppetCast provides some measures for preserving processing capacity and the service's uplinks. In particular, the bootstrap service will consider a request only when its sender is not on its blacklist and the request is legal. There are four cases when a node is allowed to contact the bootstrap service: (1) when it is not yet registered there; (2) when its external view expires; (3) when more than half of the nodes in its external view declared themselves dead and (4) when all of the nodes in its internal and external view appear dead. The bootstrap service can verify whether a request is legal based on the information the node is required to send along. If a node sends illegal requests too often it is put on the blacklist. Depending on how often it makes these requests the bootstrap service either refuses to answer after authentication is complete or even stops accepting messages from its network address.

4. Performance analysis

We evaluate the performance of PuppetCast in two experiments, discussed in turn. In both experiments we study the properties of a system of 10,000 nodes of which 50% are malicious. The first property of the system we look at is the average number of dead links in a good node's internal view. This is a measure of how successful the malicious nodes are in attacking the good nodes. In addition we study the load on the bootstrap service. The load on the bootstrap service is primarily determined by how often nodes refresh their external view, as determined by the expiry time T_X. Therefore we run the experiments with several different *refresh intervals* v. The larger the interval the less load there should be on the bootstrap service. The expiration of external views of nodes is uniformly distributed over the refresh interval, as follows. When a node requests its first external view the expiry time is set to the current time T plus a random value in the range $1 \ldots v$. Subsequent external views get an expiry time $T + v$. Times and intervals are expressed in cycles of the protocol.

We also compare the obtained results to those for two basic gossip-based protocols from [10], `rand-healer` and `rand-swapper`. `Rand-healer` uses a view-merge function that keeps the freshest entries, and `rand-swapper` uses a function by which nodes effectively swap the links they are exchanging. As a result, `rand-healer` will more quickly remove the dead links from its view during churn or after failure than

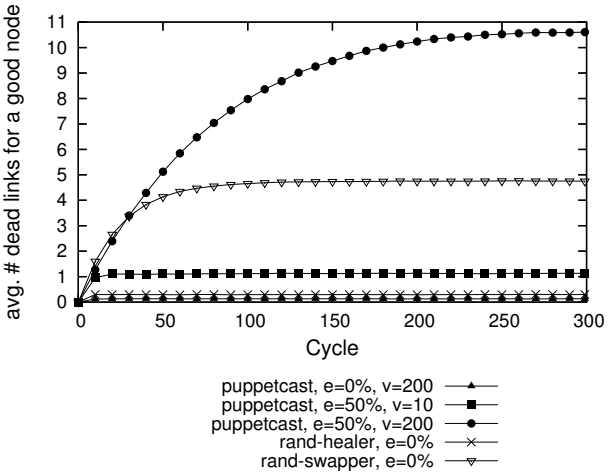

Figure 2. Average number of dead nodes in a good node's internal view in the churn experiment.

`rand-swapper`. On the other hand, `rand-swapper` balances the load of the nodes more evenly. We assume both protocols are operating in the absence of malicious nodes, and their results thus represent the ideal case. For all protocols we use an (internal and external) view size d of 20.

4.1. Churn experiment

The churn experiment is taken from [10]. In this experiment we study the properties of a system of 10,000 nodes over 300 cycles of the protocol. Each cycle 1% of the nodes (=100 nodes) gracefully leave the system and are replaced with 100 new nodes. The percentage of malicious nodes in the initial network as well as the nodes joining and leaving is 50%.

The good nodes leaving will send death certificates to their publishing nodes and the bootstrap service. When good nodes have received death certificates for more than 50% of the links in their external view, they refresh it at the bootstrap service, uniformly distributed over the next 10 cycles. The 50 malicious nodes that leave each cycle will not send death certificates or deregister from the BS. In addition, none of the malicious nodes will pass on death certificates they received from good nodes. Otherwise the malicious nodes adhere to the PuppetCast protocol.

Figure 2 shows the average number of dead links in the internal view of a good node. The numbers are the average of ten runs, all with a standard deviation of less than 0.14. The degree of a good node is 20 on average. The variable e in the labels indicates the percentage of malicious nodes in the system. For PuppetCast at e=50%, the number of dead links first increases. Each cycle 50 malicious nodes leave without telling their publishing nodes or the bootstrap service. The 50 good nodes that leave do tell their publishing nodes, but if those are malicious the death certificates will not be passed on. As a result, the publishing nodes start to spread external views that contain more and more dead links. Furthermore, the bootstrap service starts to assign external views to new nodes that contain dead links from the start. As more of these external views are received by a good node and merged into its internal view, the number of dead links in that view increases.

After a while, PuppetCast's mandatory periodic reregistration causes the number to stabilize. This is because a newly handed-out external view contain less dead links than an old one, for two reasons. First, the new view no longer contains any addresses of good nodes that already left. Second, the registrations of the malicious nodes that left without telling expire and their addresses are no longer included in external views. If we run the same experiment with the `rand-healer` and `rand-swapper` protocols in the absence of malicious nodes, they manage to stabilize quickly at a low number of dead links. In such a benign environment, PuppetCast performs even better than `rand-healer`, due to its use of death certificates.

The speed and level at which the number of dead links stabilizes in PuppetCast depends on the refresh interval v. The shorter the interval, the faster the number stabilizes and at a lower level. Unfortunately, short intervals put a high load on the bootstrap service. In addition, if the interval becomes too long the number does not stabilize. It turns out that up until $v = 200$ periodic registration removes as many entries for dead malicious nodes from the BS database as there are malicious nodes that leave (50).

The interval $v = 200$ is therefore the one interval that puts the lightest affordable load on the bootstrap service. This load can be calculated as follows. Each cycle, 100 new nodes register at the BS, and 50 good nodes leaving the system send a `deregister` message. In addition, we measured that at $v = 200$ 51.1 nodes reregister themselves at the BS either due to the mandatory periodic reregistration or because they received death certificates for more than 50% of their client nodes. The total load on the BS is therefore very low at 201.1 requests per cycle for a 10,000-node network. For example, given that cycle times (T_C) are expected to be in the range of 10 seconds for real Internet applications, the total load in this scenario is just 20.1 requests per second. A more complete analysis of the load on the BS at different refresh intervals is found in [1].

As the load is so low it is possible to use shorter refresh intervals, trading load for a lower average number of dead links in the system. However, $v = 200$ is a viable refresh

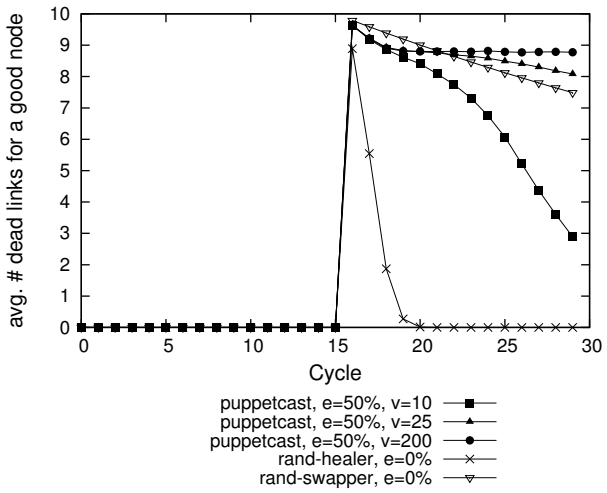

Figure 3. Average number of dead nodes in a good node's internal view in the catastrophic-failure experiment.

interval as it supports the system in the worst-case scenario of 50% bad nodes. The percentage of bad nodes is actually more than 50% because we implicitly allowed a Sybil attack to take place, making $v = 200$ even more viable, see [1].

4.2. Catastrophic failure

In this second experiment (also from [10]) we observe a 10,000-node network for 30 cycles. At cycle 15, 5000 of the nodes, randomly chosen, crash unexpectedly. This means that none of the nodes (good or malicious) is able to send death certificates or deregister from the bootstrap service. Hence, this experiment shows only the recovery power of PuppetCast by means of its periodic reregistration, as death certificates do not help here. There are also no sensible attacks possible.

The property of the system we are interested in is again the average number of dead links in a good node's internal view, which is shown in Figure 3. The numbers are the average of ten runs, all with a standard deviation of less than 0.10. The catastrophic failure causes half of the links in the internal view to become dead. As in the churn experiment, the speed at which PuppetCast recovers depends on the refresh interval. By tuning this parameter PuppetCast can be made to recover quickly like `rand-healer` or more slowly following `rand-swapper`. To approximate the speed of `rand-swapper`, the refresh interval has to be between $v = 10$ and $v = 25$. In this scenario of catastrophic failure and our uniform distribution of refreshes over the interval, the load on the BS is equal to $\frac{10{,}000}{v}$. This implies a load of between 400 to 1000 reregistration requests per cycle on the bootstrap service, which is a light load. Longer refresh intervals are also viable, as a catastrophic failure such as described here is likely to be rare.

4.3. Scalability

The load PuppetCast puts on the bootstrap service in both experiments is very low. For a system of 10,000 nodes in the churn scenario the load is just 201.1 requests per cycle, which can easily be handled by a single server (fault tolerance considerations apart). Furthermore, the load on the BS appears to scale linearly with the size of the network in this case. In an experiment we ran with a network 10 times the original size (=100,000 nodes) the load on the bootstrap service increased ten fold [1], which can also still be handled by a single server. We therefore claim PuppetCast is suitable for building a peer sampling service for very large numbers of nodes.

We have also conducted a number of churn experiments with 90% malicious nodes. Even in these extreme conditions PuppetCast continues to function and prevents a good node's internal view from being totally polluted. For example, the average number of dead links in a good node's internal view was (on average) 1.6 for $v = 10$ and 12.5 for $v = 200$ at cycle 299 (for 10 runs of the experiments). Furthermore, we measured that the percentage of good links in a good node's internal view is indeed 10% as desired. However, it is not clear that samples with such a small percentage of good links are useful to the higher-level protocols that use PuppetCast for sampling.

In addition to the properties of a good node's internal view we also verified that the graph formed by the nodes and the links in their internal views still has the desirable properties of a balanced indegree (meaning the load is distributed evenly amongst nodes), low average path length and a low clustering coefficient, see [1].

5. Related Work

[8] provide some sketches of how to detect and neutralize malicious nodes in gossip-based protocols. However, these indicated solutions require that malicious intent can be derived from the messages the nodes send. This is not possible in peer sampling where the messages are just lists of node addresses.

[11, 12] propose solutions to protect peer sampling protocols against the attack described in the Introduction, the so-called hub attack. Unfortunately, the proposed solutions can defend only against a percentage of malicious nodes smaller than 0.25 percent of the population.

Fireflies [13] is a secure membership protocol that provides a probabilistic view of the complete membership of non-malicious nodes. In addition, the protocol provides a set of neighbors per node. A peer sampling service can be built from this protocol by randomly selecting peers from the full membership or by using the neighbor set if that is a proper random sample (which is not clear). The authors claim the protocol scales to thousands of nodes. We have shown that PuppetCast is much more scalable in the scenarios discussed.

The Brahms [3] protocol provides each correct node with a uniform random sample of the system. It was shown to resist 20% of nodes in the system being malicious when there is churn, and it is fully decentralized. However, the samples of the population it provides are rather static, only changing when either a better matching node (according to a certain distance function) is found or one of the nodes in the sample goes down. It is therefore not directly suitable in securing higher-level protocols that depend on continuously receiving new samples (e.g. [9]). PuppetCast also scales to large systems, can resist higher numbers of malicious peers under churn, and, moreover, provides new samples incessantly, ensuring the higher-level protocols will expeditiously receive the address of every other node in the system.

6. Conclusions

In this paper we presented PuppetCast, a protocol that continuously supplies a node in a distributed system with the addresses of a random selection of other nodes in the system. Such a peer sampling protocol can greatly improve the effectiveness of higher-level, gossip-based protocols for information dissemination, load balancing, network management and other important tasks. Unlike previous peer sampling protocols, PuppetCast is highly resistant to attacks by malicious nodes, scales to large systems and provides dynamic samples. In particular, we showed that PuppetCast continues to operate even when 50% (or more) of the nodes in the system are malicious and attacking. Furthermore, we showed that it only puts a light load on the central trusted authority it is based on during these attacks. We therefore claim that PuppetCast can scale to systems of significant size.

References

[1] A. Bakker and M. van Steen. Security Aspects of the P2P-TV Client: A Secure Peer Sampling Service. D1.14 Deliverable, Freeband I-Share project, Enschede, The Netherlands, Feb. 2008.

[2] Z. Bar-Yossef, R. Friedman, and A. Kama. RaWMS - Random Walk based Lightweight Membership Service for Wireless Ad Hoc Networks. In *Proc. 7th ACM Int'l Symp. on Mobile Ad Hoc Networking and Computing (MobiHoc'06)*, Florence, Italy, May 2006.

[3] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer. Brahms: Byzantine Resilient Random Membership Sampling. In *Proceedings 27th ACM Symp. on Principles of Distributed Computing (PODC'08)*, July 2008.

[4] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure Routing for Structured Peer-to-peer Overlay Networks. In *Proc. 5th Symp. on Operating System Design and Implementation (OSDI'02)*, page 2990314, Boston, MA, USA, Dec. 2002.

[5] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. 6th annual ACM Symp. on Principles of Distributed Computing (PODC '87)*, pages 1–12, Vancouver, British Columbia, Canada, Aug. 1987.

[6] J. R. Douceur. The sybil attack. In *IPTPS '01: Revised Papers from the 1st Int'l Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[7] M. Jelasity, A. Montresor, and O. Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Proc. 1st Int'l Workshop on Engineering Self-Organizing Applications (ESOA'03)*, Melbourne, Australia, Apr. 2003.

[8] M. Jelasity, A. Montresor, and O. Babaoglu. Detection and Removal of Malicious Peers in Gossip-Based Protocols. In *Proc. 2nd Bertinoro Workshop on Future Directions in Distributed Computing*, Bertinoro, Italy, June 2004.

[9] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. on Comp. Sys.*, 23(3):219–252, 2005.

[10] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based Peer Sampling. *ACM Trans. on Comp. Sys.*, 25(3), Aug. 2007.

[11] G.-P. Jesi, D. Gavida, C. Gamage, and M. van Steen. A Secure Peer Sampling Service as "Hub Attack" countermeasure. Technical Report UBLCS-2006-17, Dept. of Computer Science, University of Bologna, Italy, May 2006.

[12] G.-P. Jesi, D. Hales, and M. van Steen. Identifying Malicious Peers Before It's Too Late: A Decentralized Secure Peer Sampling Service. In *Proc. 1st IEEE Int'l Conf. on Self-Adaptive and Self-Organizing Systems (SASO'07)*, Boston, MA, USA, June 2007.

[13] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable Support for Intrusion-Tolerant Network Overlays. In *Proc. Eurosys'06*, Leuven, Belgium, Apr. 2006.

[14] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, USA, 2001.

[15] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems. *ACM Comput. Surv.*, 39(1):3, 2007.

[16] S. Voulgaris and M. van Steen. An epidemic protocol for managing routing tables in very large peer-to-peer networks. In *Proc. 14th IFIP/IEEE Int'l Workshop on Distributed Systems: Operations and Management, (DSOM 2003)*, Heidelberg, Germany, Oct. 2003.