CCIT Report #500 August 2004

Silent Attack Hindering in Drum

Gal Badishi[1] Aran Bergman[2] Nadav Lavi[3] Isask'har Walter[4]
Electrical Engineering Department, Technion – Israel Institute of Technology

Abstract

Gossip based multicast is a scalable and reliable protocol for dissemination of information within a group of interconnected users.
Upon receiving or producing a message the process sends (*pushes*) it to a small constant subset of processes which is randomly selected out of the group of members. Some implementations of gossip based multicasts poll a small subset of processes for new information, effectively *pull*ing, instead of *push*ing it. In this manner, each message is eventually delivered to every process, with a high degree of probability [1].
An intrusion-tolerant version of a gossip-based multicast algorithm, developed by G. Badishi, I. Keidar and A. Sasson [2], employs several schemes in order to minimize the effect of DoS attacks on a member.
One possible attack on this protocol is one in which a malicious (or a malfunctioning) process acts normally, but actually does not forward any useful messages, and replies to *pull* requests with null entries, thus affecting the performance of the protocol. In this project we suggest a failure detector for such a malfunction or attack and a way to overcome it.

Introduction

"This Discourse, of human indifference, it's shouting out its urgently preparing for the worst, this conversation is at an end my brother, and this time the fear is kicking in, my enemy."

- This Discourage. The Silent Attack

Gossip-Based Multicast Algorithms

As described in [1], *gossip-based multicast protocols* are a class of epidemiologic protocols, which have been introduced as an alternative to the "traditional" reliable multicast protocols. The main motivation is to trade the reliability guarantees offered by costly deterministic protocols against probabilistic reliability guarantees, but in return obtain very good scalability and fault-tolerance properties. The reliability of gossip-based protocols suffers lightly as more processes fail. Furthermore, these algorithms are adaptable, meaning that they support dynamic addition and removal of group members and are also relatively easy to implement and deploy.
Decentralization is the key concept underlying the scalability properties of gossip-based broadcast protocols. In contrast to sender-reliable protocols or receiver-reliable protocols, gossip-based multicast protocols are part of the class of peer-to-peer protocols. While retransmission requests in traditional algorithms can be handled by any process but lead to the re-broadcasting of a message, gossip-based protocols rely on interaction between peers.
In typical gossip-based algorithms, messages are disseminated by having every process periodically exchange information with a randomly chosen subset of processes inside the system (*view*). In each gossip-round, a process may send messages to the processes in its view (*push*-based protocols) and may also request messages from processes in the

[1] badishi@techunix.technion.ac.il
[2] aranb@techunix.technion.ac.il
[3] nadavl@techunix.technion.ac.il
[4] zigi@techunix.technion.ac.il

view (*pull*-based protocols). Each message is gossiped for a number of rounds. These gossip rounds are initiated locally by every process and no global synchronization is required.

A possible reliability characteristic of gossip-based multicast protocols is the probability that a message reaches a randomly chosen process within *n* rounds. The redundancy in sending messages to multiple processes increases the reliability in the face of message loss and process crash failures. In this manner, gossip algorithms may achieve high probability of spreading the message to the entire multicast group.

In order to maintain a set of active processes that can be chosen as gossip partners, gossip protocols are often complemented by *membership protocols*. These protocols may be a part of the gossip protocol, or implemented separately.

Drum

Basic gossip-based multicast protocols are vulnerable to attacks on the system. Messages can be forged, senders impersonated, and processes may be flooded with pull and push requests. The Drum system, presented in [2], offers improved immunity by deploying several attack hindering schemes.

Digital signatures are used in order to authenticate the integrity of messages and the identity of senders. However, due to the complexity of generating and verifying signatures, further measures should be taken in order to prevent attackers from easily launching a *denial of service* (DoS) attack by forcing processes to exhaust their CPU while generating and verifying signatures of useless messages.

Since every process has a limited amount of available resources (buffers, communication bandwidth) in each round, a group of attackers might choose to overload the push or pull mechanisms of one or more victim processes, thus causing them to malfunction. Therefore, Drum uses a combination of pull and push operation in order to disseminate messages. Moreover, Drum limits the amount of resources a process utilizes in each round: Only a certain amount of resources is dedicated to sending messages in a push operation and another amount is dedicated to sending messages in reply to pull requests. Similarly, the amount of resources used by a process to receive and process messages pushed by other processes, or to receive messages that were sent in response to pull requests is limited.

Drum also makes use of randomly chosen communication ports in order to increase the complexity of possible attacks. Further discussion of this issue is available in [2].

Silent Attack

One possible attack on this protocol is one in which a group of malicious (or malfunctioning) processes act "normally" (i.e., a crash failure detector will consider then correct), but actually do not forward any new messages and reply to pull requests with null entries. Since any other process cannot distinguish between these processes and correct ones using only Drum operations, the performance of the protocol might be affected. In this work we quantify the degradation of Drum's performance due to such attacks using measurements of a real system. We suggest a failure detector for such a scenario and ways to mitigate the effects of such a behavior on the system. We operate under the strong assumption that an attacker knows our failure detector's inner workings.

Assumptions

- All of the assumptions presented in [2] hold for this algorithm.
- The workings of the failure detector are well known, so that the attacker can exploit them to maximize the attack's effect.

- There can be more than one attacking process. All the attacking processes can share information and coordinate their attack.
- Each process has a crash failure detector.

Proposed Algorithm

Insuring delivery using the push mechanism seems hard or impossible. The reason is that in order to identify a process which does not comply with Drum we need to gather information from all of the processes in the system. We need to ask each process which processes delivered messages to it using push operations. After we gather enough information for a long period of time, we can perform statistical calculations and reach a conclusion as to which process is, statistically, not following the protocol. This is highly inefficient, depends on the message generation rate in the system and is very slow. Moreover, an incorrect process can cause correct processes to suspect other correct processes, by falsely reporting incorrect statistics.
The problem of detecting a process that does not perform its pull requests according to Drum is equivalent to the problem of detecting a process that does not perform its push-offers, described above.
Due to the above reasons, we propose a failure detector for the correct behavior of the pull-reply mechanism.

SAH – Silent Attack Hindering

The main assumptions and characteristics with regards to the SAH mechanism are:

- Each process holds a list of suspected processes. The objective of the attacker is not to be included in any of these lists, so as to maximize its effect.
- If possible, the attacker will try and cause a correct process to add correct processes to its suspects list.
- The proposed algorithm insures correct delivery of messages using the pull mechanism. The list of suspected processes is, therefore, a list of processes that do not perform the pull-reply operation as the Drum protocol stipulates. This may be due to a malicious attack, or as a malfunction of the process. Either way – When a process randomly selects its $view_{pull}$ (the view for the pull operations) it excludes the suspected processes, since it is highly probable that the suspected processes will not deliver any new messages in their pull-reply.
- The push operation of the suspecting process does not change due to the list of suspects. The only thing that is affected is the pull operation. This is due to the fact that the failure detector gathers information only about the correct behavior of the pull-reply mechanism for each process. It cannot know anything about the behavior of the push mechanism of the process. The suspected process might be a malfunctioning one, but not necessarily an attacking process, thus we need to propagate information to that process as well. The only mechanism that can achieve that is the push mechanism. Moreover – since the probability of false detection is not zero, we need to examine the behavior of suspected processes, so that we can remove them from the suspects list if, indeed, they are not malfunctioning.

Overview

The algorithm is based on sharing information regarding the messages that a process holds and then cross-checking the process' behavior. After the initiating process p completes its push-operation to the processes in its $view_{push}$ (the view used for the push operation), it asks one of these processes q for a list of the messages that it holds. Since p just delivered a set of messages to q, it knows what messages q should have at the very least (if it does not hold these messages, process *p* might suspect *q*). It then forwards this information as is (including *q*'s signature) to a randomly chosen set of processes. Consider one of these processes, denoted *w*. This process, in turn, performs a regular pull-request on *q*. It includes a list of the messages that *w* holds, as would a regular request. The only difference is that *w* omits a message which it knows *q* holds, so that *q* will be forced to respond to the pull-request with at least one message. Otherwise, *w* will suspect that *q* is a silent attacker. The message flow is illustrated in Figure 1.

Looking at the algorithm from the attacker's point of view, it cannot distinguish between a regular pull-request and one that is aimed at detecting silent attackers, thus it is forced to react the same way to all pull-request messages. Furthermore, the attacker does not know which of the messages it holds is the one the requesting process expects. Since one of the goals of the attacker is not to be suspected by any of the processes, it has to respond to any pull-request as a regular process would.

An attacking process cannot forcefully create a scenario in which a correct process, *w*, adds another correct process, *q*, to *w*'s suspects list. If *q* is a correct process, then the list of the messages it holds will be a correct one. The attacking process cannot change the message containing that list, since it is signed by *q*. Nor can he fabricate a list of its own, for the same reason.

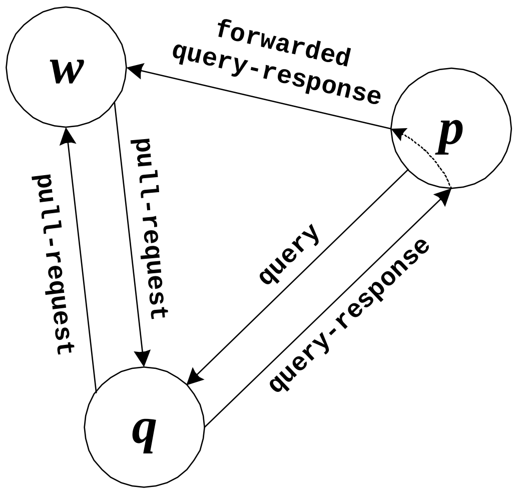


Figure 1 – The messages in the *Drum*/SAH algorithm

Detailed Description

The algorithm description makes use of the following notations:

Members -	The set of all processes in the system.
$CrashSuspect_p$ -	The set of the processes that process *p* suspects of crashing. This list is maintained by the crash failure detector of each process.
$AttackSuspect_p$ -	The set of processes that process *p* suspects of being silent attackers.
$Msgs_p$ -	The messages that process *p* holds in its buffer.
$prMsg_q$ -	The set of messages that process *q* delivered during the pull-reply.

SEND(*src*,*dest*,*msg*) -	A primitive that sends a message *msg* with *src* = source port and *dest* = destination port
qport -	A well-known port of each process on which it listens for queries.
qfport -	A well-known port of each process on which it listens for forwarded query-responses.
digest(*Msgs*)	A list containing the IDs (sequence number and source ID) of the messages in *Msgs.*

Every process *p* performs the following algorithm:

Task 1:

```
Every once in a while, after completing a push operation do
      src <- choose a random port number
      SEND(src, qport ,"query"$_p$) message to a randomly chosen
            process $q \in view_{push}$
      Wait for "query-response"$_q$ on port src OR $q \in$ CrashSuspect$_p$
      If $q \in$ CrashSuspect$_p$ then exit
      Forward the message as is to the qfport of a set of R randomly
         chosen processes from (Members \(CrashSuspect$_p$ $\cup$
         AttackSuspect$_p$))
```

Task 2:

```
Upon receiving a "query-response"$_q$ on qfport do
      m <- randomly chosen message from (Msgs$_p$ $\cap$ Msgs$_q$)
      Msgs <- Msgs$_p$ \ m
      Send a pull-request to q with digest(Msgs)
      Upon receipt of "pull-reply"$_q$(prMsgs$_q$) do
            If m $\notin$ prMsgs$_q$ then
                  Discredit(q)
```

The function of discrediting a process can mean many things. Some of the proposed methods for discrediting a process:

1. Add it to the list of suspected processes, so that the process will never be chosen for the $view_{pull}$ in the suspecting process.
2. Decrement a *confidence indicator* that is attached to each of the process IDs in the set. When this confidence indicator is less than a configurable threshold, the process associated with that confidence indicator will not be chosen as a member of the $view_{pull}$.
3. Decrement a *confidence indicator* that is attached to each of the process IDs in the set. The members of $view_{pull}$ and $view_{push}$ chosen in a random method which ensures that processes with lower *confidence indicator* will have less of a chance to be included in the view.
4. Exponential Backoff – similar to the former method, each of the process IDs in the set has a *confidence indicator* (*CI*) attached to it. When a process is detected as a silent attacker, the CI attached to the ID of that process is incremented by 1 and the process will not participate in pull operations for the next 2^{CI} rounds.

The discrediting function should be complemented by a function that updates the credit of a process that performed a pull-reply correctly. Otherwise, once a process is identified as an attacker, it will stay in the suspects list forever. Since there are falsely suspected processes, especially in an erroneous network, and since processes might malfunction for a limited period of time, this is not acceptable. Gaining back credit could be achieved by a timeout mechanism, i.e. once a process is suspected, its credit will be increased after a timeout, the length of which could be determined by the number of times the process was suspected in the past. Another method is to keep checking the behavior of suspected processes. If they were falsely suspected, and are behaving correctly later, their credit will be updated accordingly.

Possible Attacks

One possible attack that can be employed is one in which the attacking process, *g*, wants to "frame" other correct processes. In order to do so *g* collects digests from a number of processes it want to "frame", using query messages, and holds the query-responses for a period of time that insures that all the messages in those digests are supposed to be flushed out of the *MsgBuf* of all the processes due to the TTL parameter (see [2]). Then it forwards these digests to as many processes as it whishes. Since the messages in these digests are no longer in the *MsgBuf* of these processes, they will be suspected of being silent attackers.

Another attack is one in which an attacking process, *g*, would like to refrain from performing correct pull-replies (i.e. sending back new messages), but does not want to be discredited for it. It can do so by faking the progress of its own round counter or timer. Since each message is supposed to be stored by a process for a limited time, the attacker can abuse this behavior. When *g* receives a pull-request, it acts as if its round counter or timer advanced far beyond the number of rounds that the messages are supposed to be saved. That way, to any process querying *g*, this would seem a legitimate reason not to forward any messages.

The attacker could use the following tactics: on pull-requests it does not return any message (thereby, having a negative effect on the dissemination of information). To keep from being detected as a silent attacker, each time the attacking process needs to send a digest (e.g., as a response for a push-offer), it acts as if for every sender in the system (we assume that the number and identity of the senders is known to the attacker) it already received all of the sender's messages, and already discarded them from its buffer.

Code Overview

In this and the following sections, the terms below were used:

snitch message -	Forwarded query-response.
snitch operation -	
"check" -	The pull-request that is initiated upon receiving a snitch.

Drum Code Overview

As explained, the Drum protocol belongs to the family of gossip protocols. The aim of Drum is to reliably deliver messages between processes over the network.
In order to deliver the messages between processes Drum uses pull and push methods. Each process listens on two different ports to pull-requests and push-offers.
The Drum code is composed of 15 files:

FRAG.java	The main class of the Drum protocol. Creates threads to perform the protocol. The name is FRAG for historical reasons ☺
Gossiper.java	Activate and manage the puller and pusher. Activate buffer management routines every round.
Puller.java	One of the two classes that do the actual gossiping. This class is in charge of the pulling procedure.
Pusher.java	The second class that does the actual gossiping. This class is in charge of the pushing procedure.
PullWaiter.java	In charge of receiving a response on a recently sent pull-request.
PushWaiter.java	In charge of receiving a response on a recently sent push-offer.
PullReceiver.java	In charge of receiving pull requests and sending back responses.
PushReceiver.java	In charge of receiving push offers and sending back responses.
MsgBuf.java	Hold and manage the buffer of stored messages recently received.
MsgGaps.java	Hold and manage the buffer of information on messages in MsgBuf.
FRAGMsg.java	The *Drum* messages. (The name FRAG is for historical reasons).
RoundThread.java	Counts the rounds.
Certificate.java	The process certificate.
AsyncSender.java	The thread that sends the Drum messages.
Configuration.java	All the parameters needed for Drum protocol.

Focusing on Drum main methods pull and push, the Puller and Pusher send several pull-requests and push-offers with random views in each round.
Upon sending those offers and requests, a PullWaiter or a PushWaiter is spawned, according to the sending operation, for a short and limited lifetime (which is configurable). The Gossiper initiates the Puller and Pusher which send the pull-request and push-offer messages (types of FRAGMsg) using the AsyncSender.
In order to receive the pull-requests and push-offers the PullReceiver and PushReceiver listen on the pull and push ports respectively.
Upon receiving a valid push-offer, the digest of the stored messages is sent back to the offering process and a PullWaiter is spawned.
Upon receiving a valid pull-request, PullWaiter checks to see if MsgBuf holds messages that are not contained in the received digest, and chooses a random subset of these messages. This subset is sent to the requesting process.

Code Modifications and Additions

To implement SAH using the Drum source code, several classes and methods were added to the code. The main structure of the Drum was preserved and the modifications were done according to it.

The credit mechanism implemented in our code is as follows:
A counter value (scale) is associated with each of the processes in the set. The counter is initialized to 50 at the beginning of the experiment. Each time a "check" fails for a process, its counter value is decremented, and each time a "check" succeeds, the counter

value is incremented. When the counter values falls below a threshold, the process is declared as a silent attacker. If a process is suspected as a silent attacker, its counter value should exceed a different, higher, threshold, to be considered a correct process.

Another mechanism implemented in the code is a crash failure detector. This failure detector is similar to the credits used for the silent attacker failure detector. Again, a counter (scale) is associated with each process in the set. When a process fails to respond to messages sent to it (such as pull-request and push-offer), its counter value is decremented. If the response comes in a timely fashion or a message is initiated by that process, the counter value is incremented. There are two threshold values for this mechanism. If the counter value falls below the lower threshold, the process is considered to be inactive (crash failure). Once a process is suspected of being inactive, its counter should exceed the upper threshold to be considered active.

Below is a summary of the changes made to the Drum code:

Additions to the *Drum* code

1. Add a snitch port to the existing ports.
2. Add a set of parameters necessary for SAH implementation and operation.

Nsnitch	Number of digests to send to each snitch recipient.
Nsnitchrecipients	Number of snitch recipients.
Nacceptsnitch	Number of snitch messages to accept in a single round.
Rroundsbetweensnitch	Number of rounds between snitch operations.
Rwaitsnitchreply	Number of rounds to wait for a snitching procedure.
Naceptsnitchreply	Number of pull-response messages to accept after as part of a "check".
Rroundsbetweensameattacker	Minimum number of rounds between consecutive "checks" to the same process.
AtckHighTH	Attacker’s upper threshold.
AtckLowTH	Attacker’s lower threshold.
LiveHighTH	Liveliness upper threshold.
LiveLowTH	Liveliness lower threshold.

3. The ability to send a snitch to other processes was added to the Gossiper.
4. A buffer was added to hold digests received from other processes.
5. Every *Rroundsbetweensnitch* rounds the Gossiper randomly selects *Nsnitch* snitch messages from the digest buffer, and sends every message to a random *Nsnitchrecipients* processes selected from the current view.
6. A SnitchReceiver class was added to deal with snitching messages.
7. The SnitchReceiver accepts *Nacceptsnitch* snitch messages at most in each round.
8. Upon receiving and accepting a snitch message, the digest is extracted from the message and a pull-request message addressed to the checked process is created with a special digest according to the local digest and the extracted one. This message is sent using the AsyncSender. In addition a SnitchWaiter thread is spawned.

9. The checked process is recognized according to the certificate in the snitch message.
10. A SnitchWaiter class that analyzes the respond of the attacker to the pull-request was added. The snitch on the attacker will be confirmed if the attacker doesn't cooperate, meaning the attacker does not send back the messages that were in its digest and were deleted from the special digest sent to him
11. adjustDigest method creates the special digests according to the checked process' digest received from the snitching process and the local digest.
12. A database (hash table) that holds the history of the checking procedure was added.
13. A method that manages the scales of every process in the system according to results of the snitching procedure.
14. A database (hash table) that holds the scales of the processes in the system.
15. The Puller was modified to support the use of the scale database.

Changes in the existing *Drum* code

1. In Configuaration.java:
    a. Add a snitch port, PORT*_OFFSET_SNITCH_RECEIVER*.
    b. Add snitch parameters: *Nsnitch, Nsnitchrecipients, Nacceptsnitch, Rroundsbetweensnitch, Rwaitsnitchreply, Naceptsnitchreply* and *Rroundsbetweensameattacker*.
    c. Add scale parameters: *AtckHighTH, AtckLowTH, LiveHighTH* and *LiveLowTH*.
2. In Gossiper.java:
    a. Add a method *SendSnitch(RecipientView,AttackerView, rawData)*
    b. Add to run() method the activation of *SendSnitch* method in every *Nroundsbetweensnitch*.
    c. The Gossiper will activate *SendSnitch* every *Nroundsbetweensnitch* rounds. The *SendSnitch* will send a snitch on each attacker included in AttackerView to *Nsnitchrecipients* of the processes in RecipientView.
3. In MsgGaps.java:
    a. Add a new method *findSameMessages(Vector checkedSerials)*
    The *findSameMessages* method is activated on a local gap vector to create a new gap vector that will hold all messages located in both the local vector and the *checkedSerials* vector.
4. In FRAG.java:
    a. Add an *AttackerScale* database.

New Classes

`SnitchReceiver.java`

This class is responsible for receiving snitch messages. Upon receiving a snitch message (maximum of *Nacceptsnitch* messages per round) the *handleSnitch* method is called. This method validates the message and creates a pull-request to be sent to the checked process with a special digest created using the *adjustDigest* method.Before sending the special pull-request a *SnitchWaiter* is spawned.

`SnitchReceiver.java Variables`

`Hashtable attackersChecked`

Hash table which holds for each process that was checked the round number of the last "check". This data is used in the validation processes in *handleSnitch* method. A process will be checked only if more than *Rroundsbetweensameattacker* elapsed from the last checking procedure.

`SnitchReceiver.java Methods`

`handleSnitch(FRAGMsg msg)`

Validate the message.
Check if *Rroundsbetweensameattacker* rounds passed.
Update *attackersChecked* database.
Create a special digest using *adjustDigest* method.
Send the digest to the checked process using the *sendPullRequest* method.

`sendPullRequest(attacker,digest,rawData)`

This method is adopted from the *pullReceiver* class.
Send a pull-request to the checked process with the special digest.
Spawn a *SnitchWaiter* on the selected port.

`adjustDigest (AttackerDigest)`

Adjust the local digest according to the checked process' digest.
This method will return a new and special digest.
The operation of this method can be described as:

$$L \setminus (\textit{one of}\,(L \cap A))$$

Here L is the local set of messages and A is the checked process' set of messages.
The special digest hold the messages that are located in both the local set and the alleged attacker set. From this intersection one message is randomly selected and deleted. The selection algorithm is composed of three phases, the first random selection of the source process, the second phase a gap is selected and in the third phase one message from the selected gap is deleted.
In order to be consistent with previous pull-request operations all messages located only in the local database are added to the digest, this way the attacker will have more difficulties to know when the message is a real a pull-request or one triggered by a snitching operation. If $L \cap A = \emptyset$ the snitch is canceled.

`snitchWaiter.java`

This class waits for a certain process to respond with a certain message on the pull-request sent due to a snitch. After receiving *Naceptsnitchreply* messages or *Rwaitsnitchreply* rounds elapsed, it checks whether the expected message is located in the buffer. Thus it decides whether the checked process' scale should be decremented or not. The *snitchWaiter* runs for a short and limited time.

`SnitchWaiter.java Methods`

`SnitchWaiter(Drum,port,Certificate,Rounds,messages,expectedMessages)`

This method listens to the given port for a given number of rounds and expects to receive from the process with the given certificate a certain expected message. If it doesn’t receive the expected message during its lifetime, the process' scale is decremented. The checking procedure is done by the *handleSnitchReply* method. According to the result of this method the scale of the checked process is adjusted.

`handleSnitchReply(Vector msgsOfAttacker)`

This method checks whether the deleted message was received, if the message wasn't received the interrogated process is declared as an attacker, if it was received the process is declared as normal.

`Snitcher.java`

The Snitcher class implements the basic snitch elements and various operations that are related to logging and debugging. Its methods were separated from the Drum's implementation in order to enable easy integration when a new Drum version is released.

If *Rroundsbetweensnitch* rounds have elapsed since the last snitch operation, the *sendSnitches* method is called for each host in the snitch view (a randomly group of chosen hosts). This method sends *Nsnitch* messages (that are actually previously stored push reply messages) to each of the hosts. The messages are sent unchanged, since there is no need to validate the signatures or to sign it again.

`Snitcher.java Variables`

`Hashtable pushReplyTable`

This hash table stores the digests that were received from other hosts in reply to push offers. This digests are later sent as snitch messages.

`Snitcher.java Methods`

`snitch(Vector viewSnitch)`

This method checks whether *Rroundsbetweensnitch* round have passed since the last snitch operation. If so, it uses the *getRandomDigests* method to randomly select *Nsnitch* digests, and calls the *sendSnitches* method to send each digest to every host in *viewSnitch*.

`sendSnitches(Certificate cert, Vector snitchs)`

This method uses the asyncSender object to send each snitch message in the *Snitchs* vector to the host that owns the certificate *cert*.

`addDigest(String senderID, FRAGMsg msg)`

This method adds the push reply message *msg* to the hash table that stores push reply messages (*pushReplyTable*).

`getRandomDigests(int numDigest)`

Returns a vector of *numDigest* randomly chosen digests from *pushReplyTable*. If the table holds less than *numDigest* digest, all available digest are returned.

The following methods are used for logging and debugging, and may be omitted without damaging the snitching capabilities:

`initLogs(String filename)`

Initializes both log files (.exp and .log) by writing a message to the files indicating the host's own ID.

`gapsToString(Vector gap)`

Converts the *gap* vector to a string for logging purposes.

`digestToString(Hashtable digest)`
Converts each digest in the hash table *digest* to a string for logging purposes.

`writeToLog(StringBuffer text, int verboseLevel)`
Writes the *text* to the host's .log file, if the verbose level is higher (numerically equal or lower) than *Configuration.VerboseLevel*. The *text* is also written to the screen if the screen verbose level is numerically equal or lower than *VerboseLevel*.

`writeToExpLog(StringBuffer text)`
Writes the *text* to the host's .exp file, if the verbose level is higher (numerically equal or lower) than *Configuration.VerboseLevel*.

`writeToLogRound(StringBuffer text, int verboseLevel)`
Similar to writeToLog, but adds the current round number and the host's ID to the printed text.

`getOwnDigest()`
Returns the host's own digest table, which holds the message gaps for each known sender in the system.

`TTLsToString(Vector TTLs)`
Returns a string that corresponds to the vector *TTLs*, which indicates the TTL value of each message of a particular sender that is currently stored.

`TTLsTableToString(Hashtable MsgBufTable)`
Converts the hash table *MsgBufTable* to string, for logging purposes.

`attackerScaleToString(Hashtable scaleTable)`
Converts the hash table *scaleTable* to string, for logging purposes.

`sendersToString(Hashtable sendersTable)`
Converts the hash table *sendersTable* to string, for logging purposes.

`attackerScale.java`

The AttackerScale class manages two values a host keeps for each of the other hosts. The first value indicates the liveliness of the host. This value is incremented when a host communicates with another host, and is decremented when the other host fails to reply. If the value is equal or less than *Configuration.LiveLowTH* the host is considered dead. The host will be considered alive again when the value equals *Configuration.LiveHighTH*. The second value reflects how much the other host is suspected to be an attacker. It is updated whenever a host is tested using the "snitch" mechanism. The value is increased if the host replies with the message that was artificially removed and decremented otherwise. If the value is equal or less than *Configuration.AtckLowTH* the host is considered an attacker. The host will be considered correct again when the value equals *Configuration.AtckHighTH*. Each of the counters also stores the last round number in which the counter was updated and whether that update was an increase or decrease operation. These parameters enable employing complex strategies to determine which hosts are dead or attackers.

`SnitchReceiver.java Variables`

`Hashtable SendersScale`

A hash table that stores the two scales of each of the other hosts.

`Hashtable attackersView`

A hash table that stores the Ids of each of the hosts that are considered to be attackers.

`Hashtable deadSenders`

A hash table that stores the Ids of each of the hosts that are considered dead.

`attackerScale.java Methods`

`updateSnitchScale(String senderID, boolean isOK)`

Increases the attacker scale of the host *senderID* if isOK is true, decreases it otherwise. This method also stores the current round number and the direction of the update in order to facilitate future improvement in the mechanism that distinguishes between a correct process and an attacker. If the scale falls below *Configuration.AtckLowTH,* the host *senderID* is considered an attacker. If the scale is equal to or grater than *Configuration.AtckHighTH* the host is considered correct.

`updateLivelinessScale(String senderID, boolean isOK)`

This method is identical to the updateSnitchScale method, but it updates the liveliness scale and uses *Configuration.LiveLowTH* and *Configuration.LiveHighTH* as thresholds. Using these thresholds the method decides whether the host is dead or alive and updates *attackersView* accordingly.

`Hashtable getAttacker()`

Returns a hash table that holds the Ids of the processes that are currently considered to be attackers.

`Hashtable getDead()`

Returns a hash table that holds the Ids of the processes that are currently considered to be dead.

`Hashtable getAttackerScale()`

Returns a hash table that holds the scales of each of the other hosts.

New Packages

`SilentAttacker`

The package SilentAttacker implements a silent attacker. It is almost identical to the Drum’s implementation, but some minor modifications and additions were necessary to enable the required attacker’s behavior. The most notable modification is to the PullReceiver.java class: Instead of replying to a pull request as a correct host would, the attacker calls the method sAttacker.deliverMessage for each of the messages that should be sent. This method decides whether to send the message or to drop it, thus implementing a silent attack.

`SAttacker.java`

In addition to Drum's classes, SAtacker.java implements some of the silent attacker's behavior. It holds the list of the attacked processes and controls the attacking probability.

`SAttacker.java Variables`

`Hashtable attackedTable`

A hash table that stores the Ids of the hosts that should be attacked.

`SAttacker.java Methods`

`addVictim(String ID)`

Adds host *ID* to the list of processes that should be attacked.

isVictim(String ID)

Returns true if host ID is attacked, false otherwise.

`deliverMessage(String sourceID,String destID)`

This method decides whether to send a message created by host *sourceID* as part of a push reply to host *destID*. The decision can be based upon any of the parameters available in the Drum code. In the current implementation, this method always returns false, which implements a simple silent attacker that never responds to a pull request.

Implementation Issues

- Response to a pull-request even if there are no messages
  A problem that we discovered during our implementation is that a process that doesn’t have new messages to send upon reception of pull-request will be detected as a dead process and thus its scale will be decreased. A way to solve this is to add a special message that indicates that the process has no new messages. This solution was implemented in the Drum code during our implementation of SAH.

- Gaps
  During the implementation of SAH on the Drum code, we detected a possible problem with the implementation of the digests sent through the pull and push operations. Due to the fact that messages are deleted from the buffer using age based purging, these deleted messages no longer exist in the buffer but they need to be included in the digests. Therefore, virtual gaps are created and inserted into these messages. We discovered that these virtual gaps can influence our decision algorithm. This happen when the *adjustDigest* method opts to discard a message from the virtual gap, thus the interrogated process can’t send the necessary message back and it is declared as an attacker even when it is a correct process. To correct this problem the digest presentation was changed and instead of a virtual gap we have added the *minAcceptableSerial* parameter.

- Scale for attackers
  In our implementation we used a linear scale, meaning that when a process is declared an attacker or a correct process its scale is decreased or increased by one respectively. Another scale that can be implemented is an exponential scale

in which when a process is declared an attacker its scale changed exponentially and when the process is declared normal its scale is changed linearly. This kind of implementation is well known from Van-Jacobson AIMD (additive increase multiplicative decrease) strategy used in TCP window algorithms.
The implemented linear scale is much more forgiving, compared with the AIMD scale, to false declarations that can occur due to network problems or overload in the pullReceiver method. But on the other hand its reaction to detection is slow, and an attacker will be discarded from the normal view only after several detections. Another improvement that can be made is to give priority to processes already declared as attackers, this way these processes would be checked more often and their scale would be changed more aggressively.

- The detection algorithm and scales
  Another option to improve the detection algorithm is by discarding more than a single message. This way the scale can be modified by the percentage of messages return by the checked process.

- Threshold
  Obviously the thresholds can affect the performance of the SAH implementation. According to the thresholds processes are discarded from and added to the normal view. Future work must include several experiments on the affects of the threshold on the SAH performances.

- Smart coordinated attack by two processes.
  Two attackers, *a* and *b*, that coordinate their attack can overcome our detection algorithm and influence the Drum/SAH performance. This attack is possible if both attackers are sources and create their own, bogus, messages. The processes update each other on the new messages created locally. In addition both of the processes do not cooperate during the other push and pull operation triggered by normal processes, that is to say, they discard messages created by other processes. However, they do respond to pull-request, and can even initiate push-offers. This way both of the attackers hold the messages created locally and the messages of the second attacker. When *a* wants to increase *b*'s credit in the scale of a correct processes, *p*, it forwards *b*'s digest to *p*. *p*, in turn, will perform a "check" on *a*, and receive a response which is correct. This kind of attack needs further investigation.

- TTL
  An option to attack the system using the new feature of *minAcceptableSerial* is by using this parameter and adjusting it in the attacker’s digest to show that all messages were deleted from the buffer. A countermeasure is to implement the use of TTL, counting the rounds passed since the message was created, this way an attacker won’t be able to use the *minAcceptableSerial* and impersonate as a normal process. This method must be investigated in the future.

Experiments

The main goal of the analysis is to check the impact of various network parameters and processes' parameters.
The experiment parameters that can affect the operation of the algorithm include the total number of processes, the ratio between the number of attackers and the total number of processes and the influence of our suggested algorithm parameters such as thresholds and the decision algorithm. In addition, the packet loss rate of the network is

an important parameter that can affect not only the communication between processes but can also make the distinguishing between attackers to other normal processes harder.
Using specific fixed network architecture we will first evaluate how these parameters affect the performance of the non-modified algorithm (*Drum*) then we will compare the results to our suggested algorithm.
Using the results of this comparison we will be able to evaluate the impact of the parameters on our suggested algorithm and continue the tests on several modified versions of the algorithm.

Emulation Results

To measure the performance of our algorithm we used the Emulab TestBed [3].
We performed all our experiments using the following parameters and conditions (unless otherwise noted):

- All links are error-free links ($\varepsilon = 0$)
- The *push* fanout is 3
- Practically, there is no bound on the number of messages that a process sends another process in a *push* or a *pull* operation.
- The number of rounds between consecutive messages is 5
- The length of the simulation is 1000 rounds (200 messages for each source)

In all experiments where the SAH mechanism was active, the following parameters were used:

- The initiating process forwards the digest to only 1 other process ($R = 1$)
- The number of accepted snitch message at each process is 1
- Snitching is performed every round
- The number of digests sent is 1

The snitching mechanism is used by the "checking" process to gather information it does not have from the process it is checking. Therefore, performing a "check" on a process is somewhat identical to performing a *pull* operation. To compare the performance of a system with attackers using the SAH algorithm and the same system without using the SAH, we used the following parameters:

- When using the SAH mechanism, the fanout of the *pull* operation is 2
- When the experiment did not include activating the SAH mechanism, the fanout is 3

To verify that, indeed, the comparison between a system with a *pull* fanout of 3 is identical to a similar system with the SAH activated, and a *pull* fanout of 2, we performed some experiments to compare the two configurations. The following graph shows the average number of processes that received a message vs. the number of rounds from the creation of the message. In this graph, a single source was used. Experiments with different number of sources and different number of processes show the same results, and are presented in Annex A.

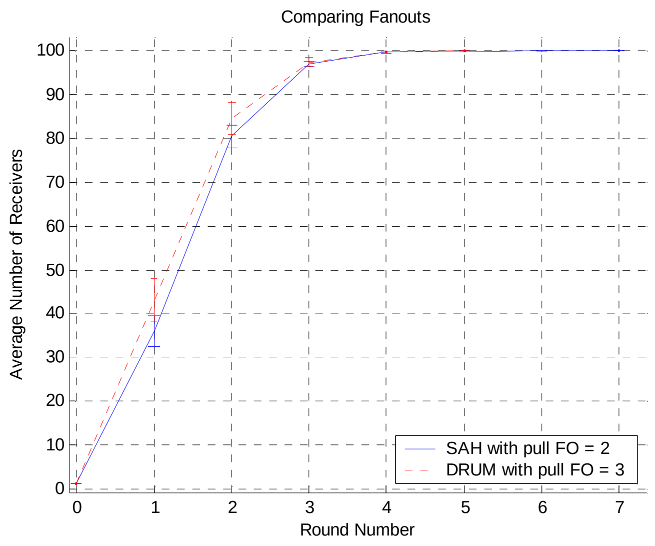


Figure 2 - Fanout Comparison

We can conclude from the above comparison, and the graphs that are presented in Annex A, that we are performing a fair comparison using the above fanout configuration, regardless of the number of processes and number of sources in the experiment. However, this conclusion is only viable when there are no attackers in the system. The effect of the SAH mechanism on the information propagation rate when there are attackers in the system is investigated in the section describing the silent attack effect.

When there are no attackers in the system, all the processes participate in the snitching mechanism. This means that every process sends a digest of some other process (chosen randomly and uniformly) to one process, chosen randomly and uniformly from the set of participating processes. If each process received all the digests and performed checks upon receiving each of them, the average number of "checks" that would be performed by each process would be 1. Since each "check" is the same as performing a *pull-operation*, we get that the performance of a system with a *pull* fanout of 3 closely resembles the performance of the same system with a *pull* fanout of 2 but with the SAH mechanism active.

The graph is only slightly lower when using the SAH mechanism, since the average number of "checks" that each process performs in each round is slightly less than 1. This is due to the fact that the number of "checks" that a process performs in each round is bounded by 1. Since there is a probability that no snitch message will reach a given process in a given round, the average must be lower than 1. Another reason is that the received digest ("snitch") is the digest of a process that was chosen to be polled in the *pull-operation* in the same round. In that case, no new information is available to the checking process.

Silent Attack Effect

To inspect the effect of silent attackers on a system we performed several experiments with different number of sources. The results of the experiments with a single source are presented in Figure 3.

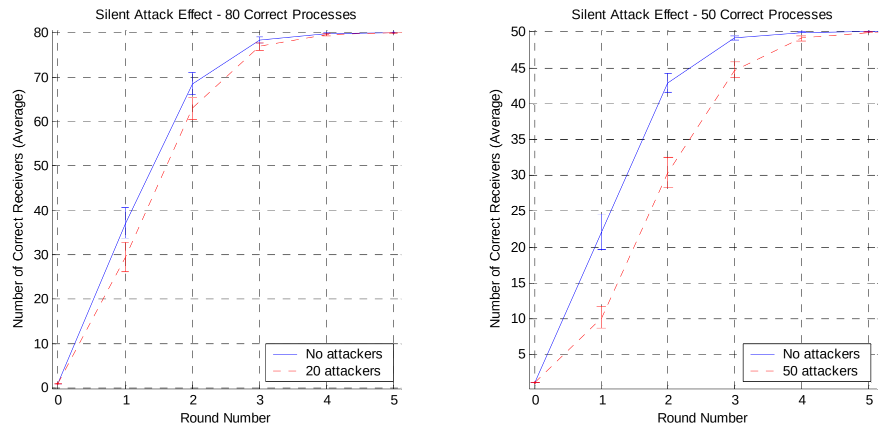


Figure 3 - Silent Attack Effect

The results of the experiments with 20 and 50 sources appear in Annex A.
It is clear that the effect of a small percentage of attacking processes is negligible. Even when 20% of the total number of processes are silent attackers, only a mild effect can be noticed. When this number increases to 50%, the effect is clearly evident. Experiments with different number of sources show the same results.

The following graphs show the same effect but with a different total number of processes. The results are normalized to the total number of processes.

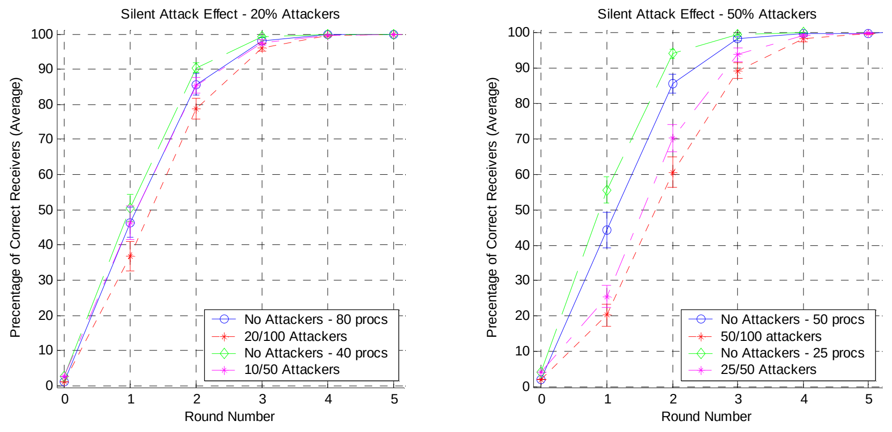
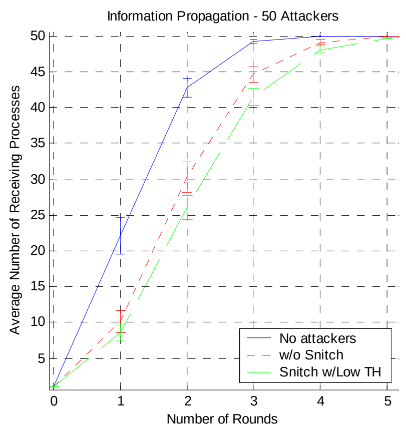


Figure 4 - Silent Attack Effect with 100 and 50 Processes

To inspect the effect of the silent attack on information propagation when the SAH mechanism is active, but while no attackers are suspected, we ran experiments with thresholds set to 0. During the experiment the counter associated with each suspected processes do not reach 0, so none of the attacking processes are actually declared as such. Figure 5 shows the information propagation rate with 20 and 50 attackers, when the SAH is active, but with thresholds set so that it is ineffective.



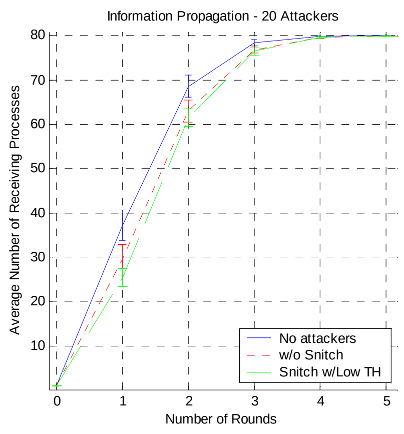


Figure 5 - Silent Attack Effect with SAH enabled

As seen in the above graphs, the performance of a system with the SAH mechanism active is slightly poorer than the performance of a system with no snitching. This stems from the fact that the comparison is made using different values of pull fanouts. In the system with no snitching, the pull fanout is 3, while the other system uses a pull fanout of 2. The results do not follow those presented in Figure 2, Figure 12, Figure 13 and Figure 14 since the attacking processes do not participate in the SAH mechanism. For the case of a system with 50 attackers, this means that only 50 processes send digests. Each of those processes sends one digest ("snitches") to one other process chosen randomly from the list of unsuspected processes. In this experiment none of the processes is suspected, so the digest is sent to one of the processes in the system. On average, this means that each process receives 0.5 digests ("snitches") every round. The outcome of this analysis is that the effective pull fanout when the SAH is active is about 2.5, which is lower than the pull fanout of the same system without the SAH, resulting in poorer performance.

Detection Rate

Figure 6 depicts the detection rate of the SAH mechanism. The graph shows the average percentage of attackers detected vs. the round number of the experiment. The average is over the correct processes, and the experiment begins with all the processes initialized with a counter value of 50.

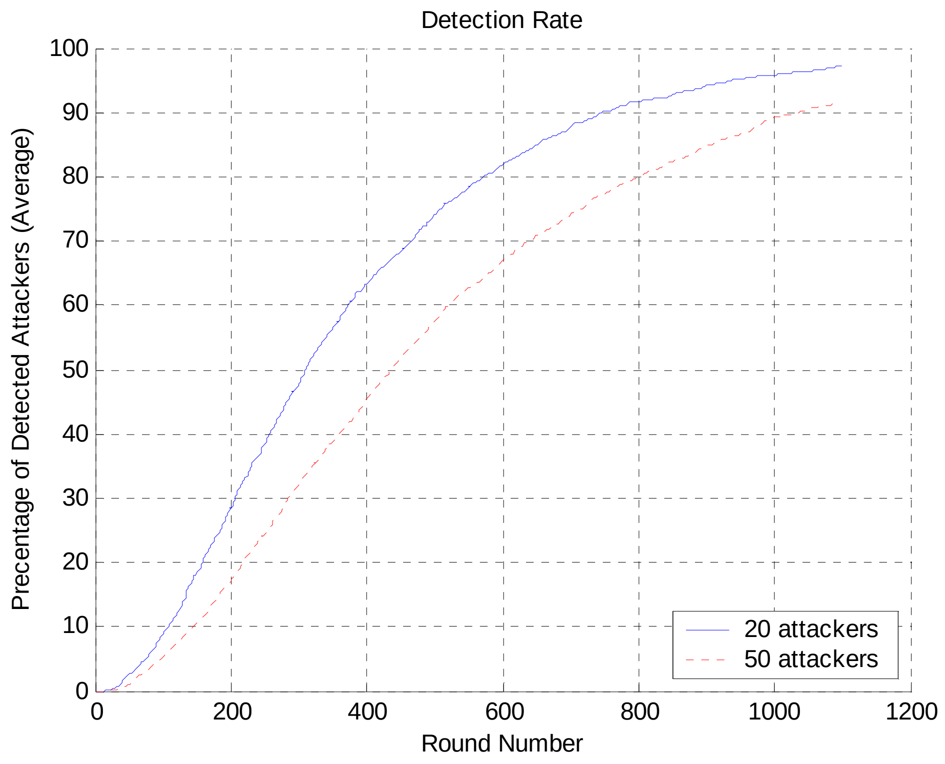


Figure 6 - Detection Rate

Similar graphs were produced with different number of sources. They are presented in Annex A. All show that the number of sources does not influence the rate at which attackers are identified.

We also investigated the influence of the total number of processes on the behavior of our failure detector. The following figure demonstrates the results:

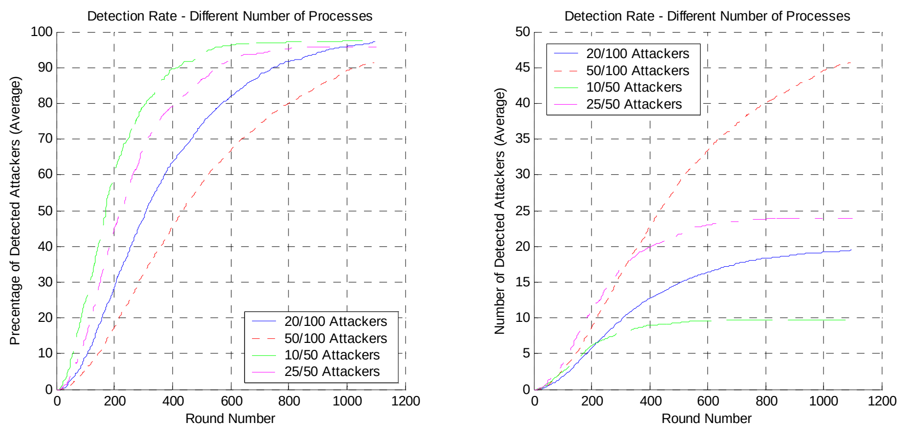


Figure 7 - Detection Rate with Different Number of Processes

As can be seen, when the total number of processes is lower, the rate at which attackers are detected is higher. We can see that the percentage of attackers is not the only parameter that influences the detection rate. Actually, looking at graph that depicts the *number* of detected attackers, rather than the graph that depicts the *percentage* of detected attackers, we can see that more attackers are detected in the beginning of the experiment in a system with a smaller number of processes when compared with a system with the same percentage of attackers, but twice as many processes. To explain this phenomenon, a more precise analysis should be made.

The graphs do not reach the maximum detection of 100% for the following reasons: In the experiment with 100 total processes, the experiment's duration was not enough to reach the maximum average number of detected attackers. In the experiments with a total of 50 processes, the maximum average number of detected attackers is limited, since the source cannot detect any attackers. This is because the source cannot send a modified digest to a suspected process without the other processes identifying the modified digest (it cannot omit its own message). For that reason, the maximum average number of detected attackers in experiments with a single source is given by the following formula:

$$\frac{\text{\#Correct Processes -1}}{\text{\#Correct Processes}} \cdot \text{\# Attackers}$$

In the above experiments this number is presented in the following table:

Total number of processes	Number of Attackers	Max. average % of detected attackers	Max. average number of detected attackers
50	10	97.5%	9.75
50	25	96%	24
100	20	98.75%	19.75
100	50	98%	49

The influence of transmission errors was investigated only when a single source is used and the total number of processes is 100.

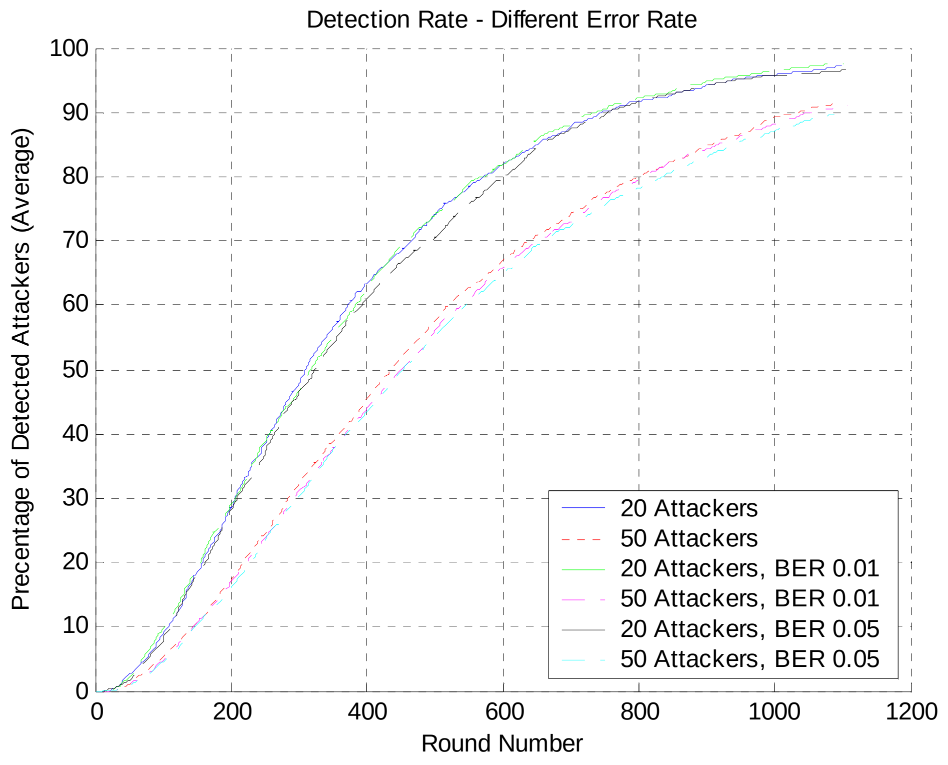


Figure 8 - Detection Rate vs. BER

The results show that the effect of reasonable error rates on the detection rate is minimal. We also notice that as the error rate increases, the percentage of detected attackers in each round is somewhat decreased. This is the results of lost "snitches". If "snitches" were not lost, the detection rate would be the same as the detection rate in a system with no errors.

False Detection

Several experiments were performed to investigate the false detection of the failure detector. The following graphs show the results of the experiments with different BER and different number of attackers. The total number of processes in the experiments is 100.

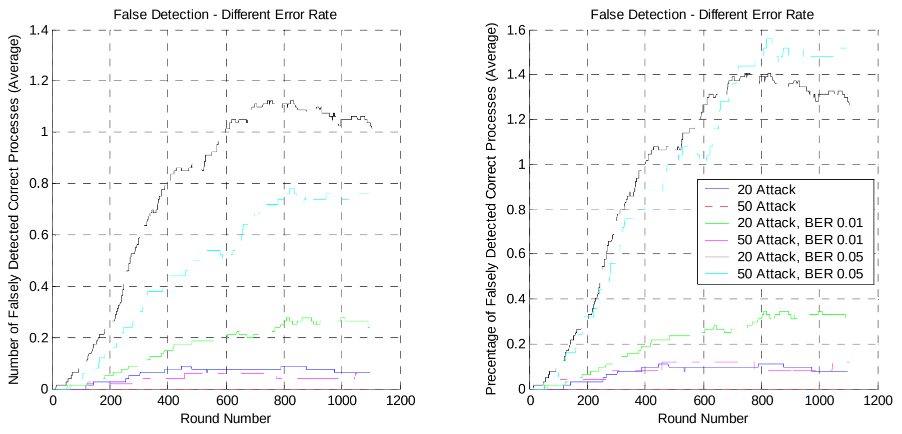


Figure 9 - False Detection with BERs

The number of correct processes falsely suspected of being silent attackers is bigger when the BER increases, which is to be expected. We also see that the number of processes falsely detected is bigger when the number of attackers is smaller. There are two things that contribute to this behavior. First, when there are fewer attackers, there are more correct processes that are checked by the SAH mechanism. Assuming that the probability of falsely detecting any process is identical, this results in a higher average. Second, when there are more correct processes, there is a higher rate of "checks" that are performed in the system. This also contributes to the higher average.
We see that for all cases investigated, the average number of correct processes, even when the BER is 0.05, stays under the reasonable value of 1.2

Failure Detector Performance

The following graphs depict the information propagation rate when the failure detector is active and compare it with the performance of a system without a failure detector. The information propagation is also compare to an ideal system, in which only the correct processes exist.
The information propagation is sampled 3 times during the simulation: at the start of the simulation, where the failure detector still does not affect the performance, at the middle of the experiment and at the end of it. The graphs shown are averages of 50 messages at the beginning, middle and end of the experiment.
Figure 10 shows the outcome of running an experiment with 50 attackers.

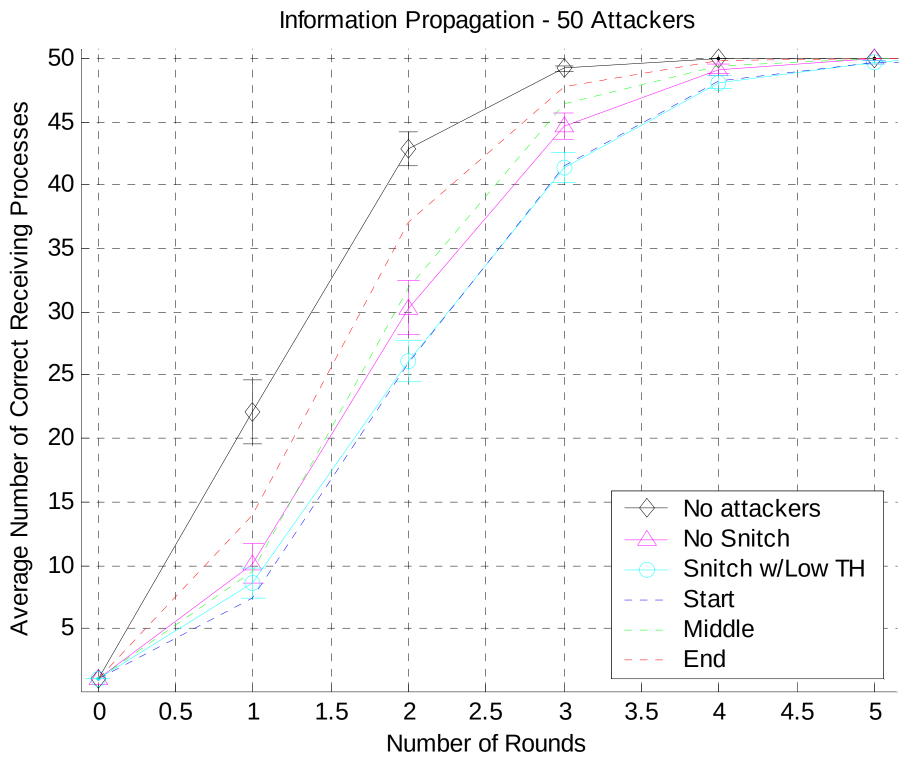


Figure 10 - 50 Attackers, 50 Correct Processes

As can be seen above, the information propagation rate is improved as the experiment advances. At the beginning of the experiment, the rate is identical to the experiment with a low threshold for detecting an attacker. This is expected, since at the beginning of the experiment no attacker, or a very small number of attackers, is detected. We can see that the performance at the end of the experiment is improved. However, the performance is not identical to the performance of a system with 50 correct processes and no attackers. This is also expected, as the push mechanism is not affected by the SAH mechanism. In addition, on average, half of the "check" operations are performed on attackers, which do not contribute to the propagation of information in the system.
Figure 11 depicts the results when running the experiment with 20 attackers.

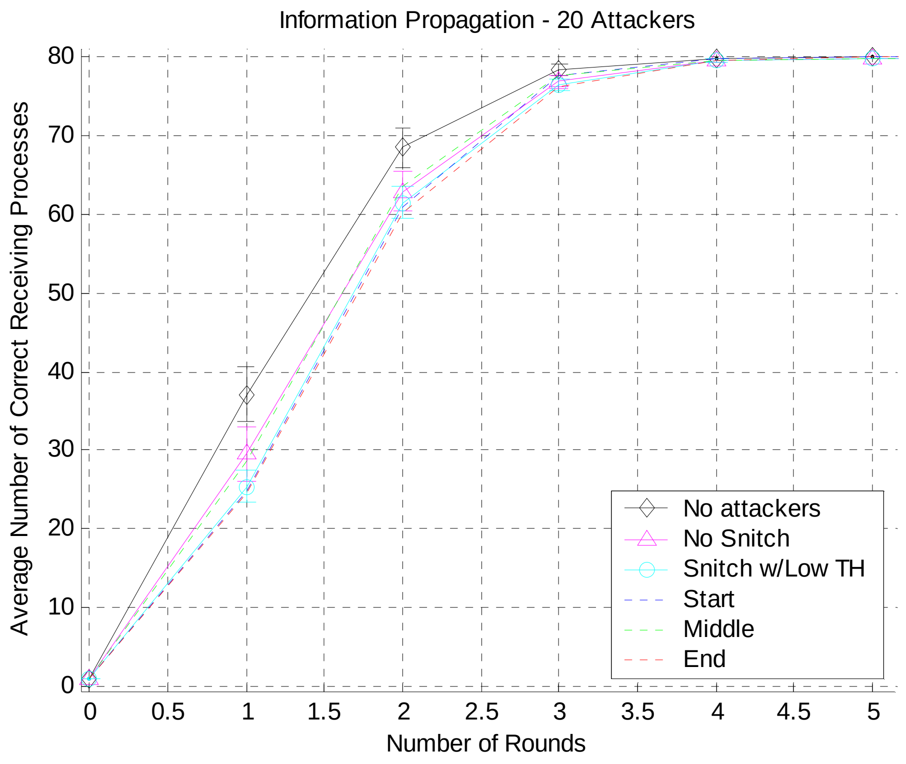


Figure 11 - 20 Attackers, 80 Correct Processes

Further Improvements

In Drum/SAH we have implemented new detectors for gossip based protocol. These detectors will find both dead processes (processes that do not function at all) and silent attackers, which smartly exhaust the system resources by misleading normal processes and keeping its attacks stealth. We analyzed Drum/SAH performances and saw the differences between Drum with no snitching mechanism and Drum/SAH. Although Drum/SAH improves the system performances the snitch and detection mechanisms can be extended to support detection of more intelligent silent attacks. We will conclude with several improvements issues:

- Different Thresholds
  Examining the effect of the thresholds is important and explained in the implementation section. Further tests and analysis will help to understand the best way to adjust these parameters to a gossip based system.
- Scale function
  In our implementation we used a simple linear scale function. Other scale functions such as exponential functions or hybrid functions such as AIMD should be implemented and analyzed.
- Improvements and adjustments to the detection mechanism
  As explained in the implementation sections there are many ways to improve the performance of the detection mechanism which will result in a better overall system performance. Several ways to do that are: improving the digest adjustment function to discard several messages; adjusting the scale according to the cooperation of the checked process; implementing a priority mechanism for snitching on known attackers.
- Different attacker strategies

An important issue that must be further investigated is the ability of the implemented detection mechanism to detect other attacks, not tested in our work, and extend the detection mechanism to support the detection of the attacks that are invisible to the implemented detector.

- Crash failure performance and effects
  The performance of the crash failure detector should be investigated.
- Dynamically change the number of *pull* Fanout
  As seen in the experiments we performed, when there are attackers in the system, the performance of a system with a pull fanout of 3 is not identical to a system with SAH mechanism and a pull fanout of 2, since the average number of snitches per correct processes is less than 1. An improvement to our algorithm could be to change the pull fanout dynamically, to accommodate the "missing" pull-requests.

Conclusions and Summary

In our work we measured the effect of silent attack under various circumstances. We saw that the effect of silent attacks on Drum protocol is minor unless the percentage of the attacker is relatively high. In those cases we show that implementing our algorithm for detecting silent attackers indeed improves the message propagation latency over time. Future work on the subject might include implementing the improvements we suggest in this paper, and further experiment with systems whose properties change over time. As an example, one might consider a system with 20 attackers for the first hour of operation and 50 attackers for the second hour.

References

1. P. Eugster, S. Handurukande, R. Guerraoui, A. M. Kermarrec, and P. Kouznetsov "Lightweight probabilistic broadcast". In Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001), July 2001.
2. G. Badishi, I.Keidar and A.Sasson, "Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast", submitted for publication.
3. B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks", appeared at OSDI 2002, December 2002.

Annex A

Fanout Comparison

The following graphs show the comparison between the information dissemination rates of the two *pull* fanouts (2 with the failure detector active, and 3 when it is not) used in the simulations:

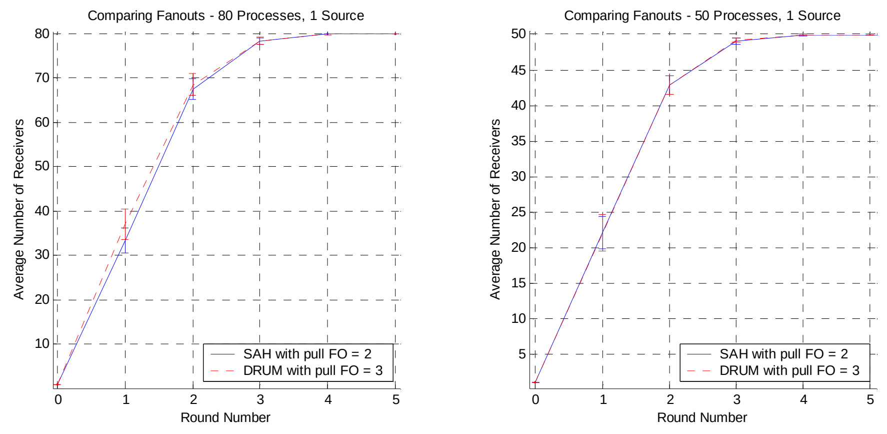


Figure 12 - Comparing Fanouts with a Single Source

Figure 12 and Figure 2, presented above in page 17, cover all the experiments we ran using a single source, with different number of processes in the experiment. Figure 13 depicts the results when there are 20 sources in the system.

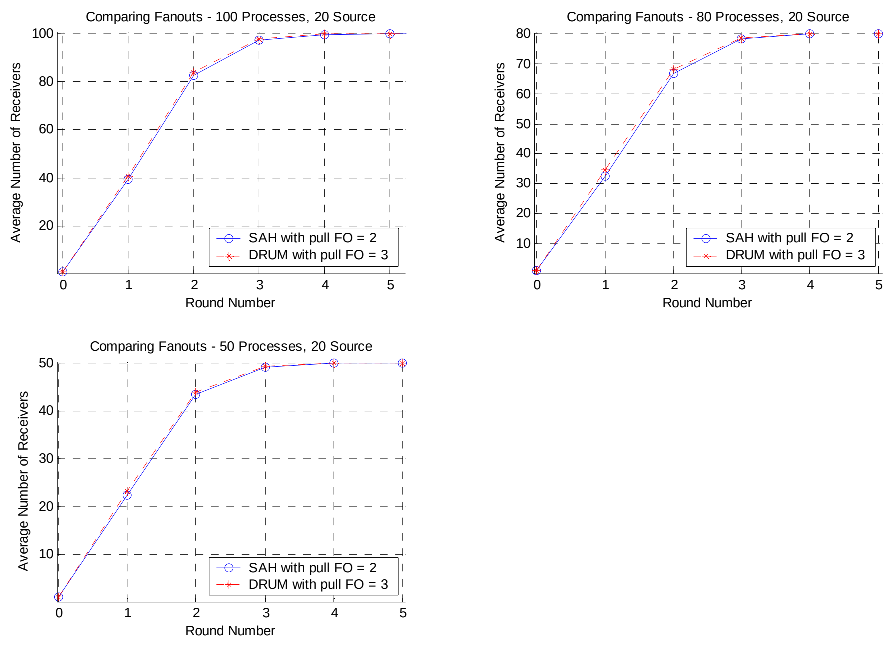


Figure 13 - Comparing Fanouts using 20 sources

The same results can be observed when 50 sources are used:

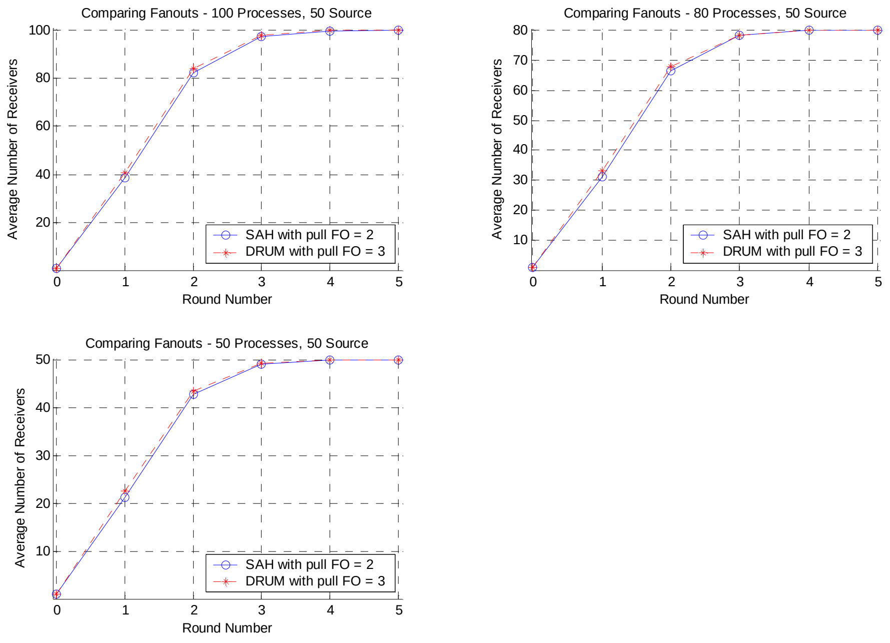
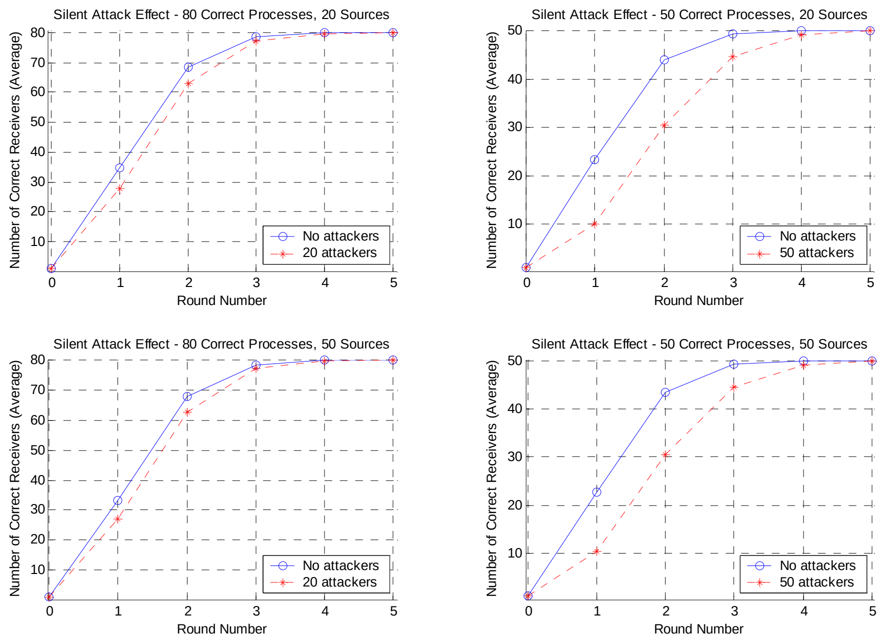


Figure 14 - Comparing Fanouts using 50 sources

Silent Attack Effect

The following graphs display the effect of the silent attack on the information dissemination rate. Several experiments were run with 20 and 50 attackers and with 20 and 50 sources. In all the experiments, the total number of processes is 100. This includes the attacking processes as well as the correct processes.



Detection Rate

Figure 15 presents the results of the experiment run with 20 and 50 sources. The total number of processes in all experiments is 100.

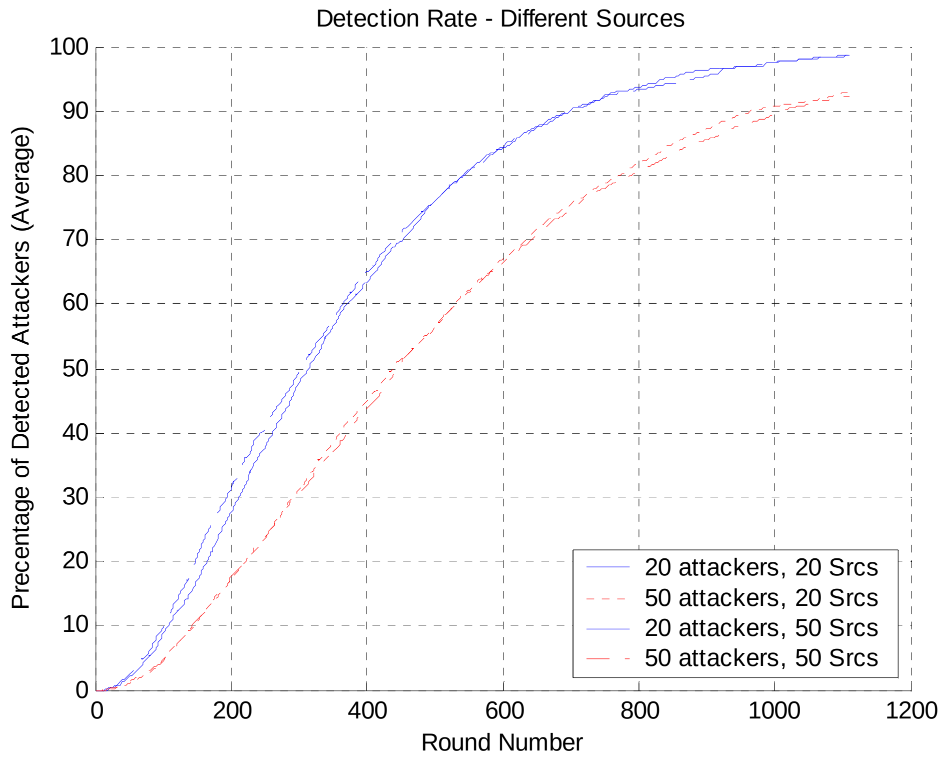


Figure 15 - Detection Rate with 20 and 50 Sources

Annex B

The results with a different number of processes (50 instead of 100), produced false detection figures that were too small to analyze. The maximum average number of processes detected was 0.1, in an experiment with a total of 50 processes, 10 of which were silent attackers.

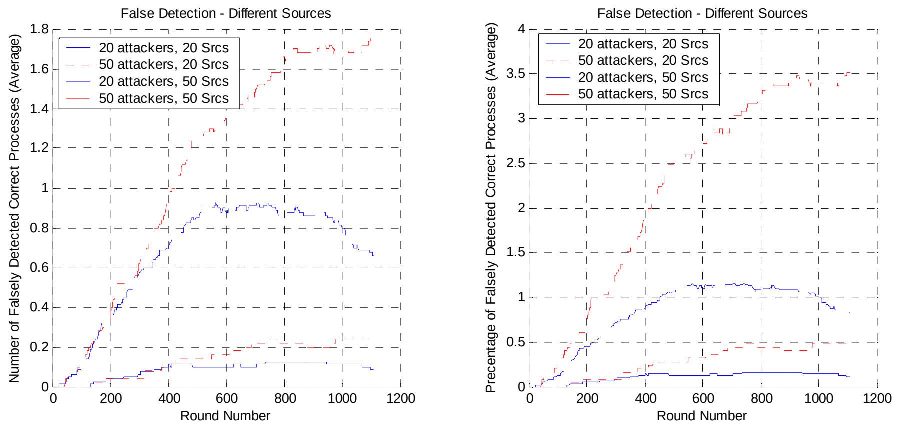
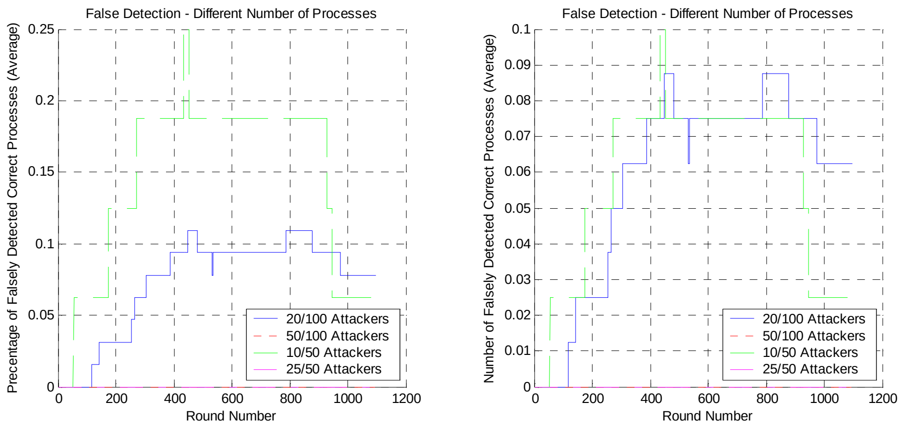


Figure 16 - False Detection with Different Number of Sources

As can be seen from the above graphs - the false detection is higher when the number of sources increases.


False Detection - Different Number of Processes
Precentage of Falsely Detected Correct Processes (Average)
Round Number
20/100 Attackers
50/100 Attackers
10/50 Attackers
25/50 Attackers
False Detection - Different Number of Processes
Number of Falsely Detected Correct Processes (Average)
Round Number