A Failure Detection Service Based on Epidemic Dissemination for Peer-to-Peer Networks

Leandro P. de Sousa 1, Elias P. Duarte Jr. 2
Federal University of Paraná (UFPR) - Dept. Informatics
P.O. Box 19018 Curitiba 81531-980 PR Brazil
1 leandrops@inf.ufpr.br
2 elias@inf.ufpr.br

Abstract—Failure detectors were first proposed as an abstraction that makes it possible to solve consensus in asynchronous systems. A failure detector is a distributed oracle that provides information about the state of processes of a distributed system. This work presents a failure detection service based on a gossip strategy. The service was implemented on the JXTA platform. A simulator was also implemented so the detector could be evaluated for a larger number of processes. Experimental results show that increasing the frequency in which gossip messages are sent gives better results than increasing the fanout. Other results include CPU and memory usage, fault and recovery detection time, mistake rate and how the detector performs when used in a simple election algorithm.

I. INTRODUCTION

Several distributed applications involve some kind of agreement between their components [1]. Processes must reach a consensus whenever they need to decide on the same value given an initial entry consisting of a set of possible values. As both processes and communication channels can fail in real distributed systems, a basic condition for distributed processes to reach an agreement is that each process must know the state (working or failed) of the other processes in the system. In some types of distributed systems, this can be hard or even impossible to implement. That is the case of asynchronous systems: in this type of system, processes and its communication channels can behave arbitrarily slowly, making it impossible to distinguish slow and failed processes. Lynch and others proved in [2] that consensus is impossible in an asynchronous system in which even a single process can fail by crashing. This result is known as the FLP impossibility.

As way of avoiding the FLP impossibility and thus solving the consensus problem in asynchronous systems, Chandra et al. proposed abstractions called unreliable failure detectors [3]. A failure detector is a distributed oracle that provides information about the state of processes of a distributed system. Failure detectors can make mistakes, i.e. fault-free but slow processes can be erroneously considered to be suspect. Chandra and Toueg proposed two properties to classify failure detectors: completeness and accuracy. Completeness requires that if a process has crashed then it is suspected by the failure detector, while accuracy restricts the mistakes that the detector can make. Even though it is impossible to implement perfect failure detectors in completely asynchronous systems, consensus algorithms using unreliable failure detectors can complete successfully if the detector output can be trusted for a long enough period [4]. Also, solutions built around failure detectors are simpler and more generic, as failure detectors encapsulate the timing properties of the system [3].

This work presents the specification and implementation of a distributed failure detection service based on epidemic dissemination. The detection protocol is probabilistic, and uses a gossip strategy [5]. To use the detector, a process must implement the service and then participate in a dissemination group. At any moment, the process can query its local detector and receive a list of processes suspected to have failed. The detection service was implemented as a prototype in the P2P JXTA platform [6]. A simulator was also implemented, using the SMPL library [7].

Experiments were conducted comparing two different strategies for service configuration. The first strategy increases the fanout of the dissemination and the other increases the frequency in which gossip messages are sent. Results show that increasing the frequency in which gossip messages are sent gives better results than increasing the fanout. Results are also given for CPU and memory usage, fault and recovery detection time, and mistake rate. In order to show how the detector performs from the application point of view, a simple election algorithm was also implemented and is described.

The rest of this paper is organized as follows. Section 2 presents related work in the field of failure detector implementation. In Section 3, the proposed detection service and the gossip algorithm on which the detector is based are described. In Section 4, experimental results are given after a description of the service implementation and simulation. Finally, Section 5 concludes the work and discusses future work.

II. RELATED WORK

Perfect failure detectors cannot be implemented in completely asynchronous systems due to the FLP impossibility. However, the study of failure detectors is of great importance to the implementation of solutions to problems in distributed systems. In [3], the authors show that unreliable failure detectors can be implemented in systems with partial synchrony. In the system model given by the authors, there are upper limits for the delay in messages and for the execution time of processes, but these limits are not known and are only valid after some unknown time, known as GST (Global Stabilization Time).
Although real networks and distributed systems do not directly correspond to the theoretical partially synchronous model, these systems frequently alternate between periods of stability and instability. During an instable period, the system is completely asynchronuous, but for the whole stable period the system presents timing properties that enable failure detectors to work reliably. If these stable periods last long enough, problems such as consensus can be solved [4].

In order to allow real applications to use and take advantage of failure detectors, they might require some additional properties besides the eventual properties described in [3]. Completeness is said to be weak if eventually every process that crashes is permanently suspected by some correct process. Accuracy can be eventual: there is a time after which mistakes do not occur. Applications have timing restrictions, and detectors that are too slow may not suffice. For this very reason, [8] proposes some metrics for the quality of service, or simply QoS, of failure detectors. The metrics proposed by the authors are mostly used to describe the speed and accuracy of detection. In other words, they describe how fast a detector reports a failure and how well it avoids mistakes. The authors also propose a failure detector, called NFD-E, that can be configured in accordance to the needs of the application using it. A probabilistic system in which the detector can be implemented is also described.

In [9], a group membership protocol is proposed, called SWIM. The SWIM protocol is divided in two parts: a failure detector and a dissemination protocol for membership information. This failure detector was initially proposed in [10]. The detector uses a randomized ping strategy, where each process periodically tests another process, which was selected randomly. Information about group membership and process failures are piggybacked in the ping messages sent by the detectors.

Recently [11] an implementation of a failure detection service was reported for a P2P storage system that tries to improve the detection QoS by using monitoring together with a prediction model.

In [12], a detection algorithm that uses a gossip strategy for failure detection is proposed. In the algorithm, processes periodically send gossip messages to a group of other processes, which are chosen randomly. Failure detection is done through a heartbeat mechanism. Each gossip message contains the heartbeat value for the sending process and the last heartbeat values it has received for every other process. If a detector does not receive new information about a process for some given time, this process is considered to have failed. The authors also suggest some improvements that can be applied so that the algorithm can be used in real networks such as the Internet, where processes are located in multiple domains. The detection service proposed in this paper is based on the algorithm proposed on [12]. Processes monitor each other using heartbeats, which are disseminated through gossip messages (epidemic dissemination). The complete algorithm is shown on Figure 1.

Every JXTA-FD instance executes the following:

```
|| Initialization: table ← new HBTable
heartbeat ← 0
timeOfLastBcast ← 0
start tasks ReceiverTask, GossipTask, BroadcastTask and CleanupTask

|| ReceiverTask: whenever a gossip message m arrives
for all <ID, hbvalue> ∈ m do
  table.update(ID, hbvalue)
end if
if m is a broadcast then
  timeOfLastBcast ← current time
end if

|| GossipTask: repeat every GOSSIP_INTERVAL units of time
if table is not empty then
  numberOfTargets ← min(FANOUT, table.size())
  targets ← choose numberOfTargets random elements from table.get_ids()
  for all t ∈ targets do
    send gossip message to t
    heartbeat ← heartbeat + 1
  end for
end if

|| BroadcastTask: repeat every BCAST_TASK_INTERVAL units of time
if shouldBcast() then
  send gossip message by broadcast
  timeOfLastBcast ← current time {not necessary if the process receives its own broadcasts}
end if

|| CleanupTask: repeat every CLEANUP_INTERVAL units of time
for all id ∈ table.get_ids() do
  timeFromLastUpdate ← current time - table.get_timestamp(id)
  if timeFromLastUpdate ≥ REMOVE_TIME then
    remove id from table
  end if
end for
```

Fig. 1. Detection algorithm used by the JXTA-FD service.

The system on which the algorithm is assumed to run can be represented as a complete graph, each process can directly send and receive messages from every other process. The system is asynchronous; in particular, the system has probabilistic properties for message delay and process failures. Only crash failures are considered. Failed processes can rejoin the system with a new identity. Each process executes an instance of the detection algorithm.

The algorithm is divided in three distinct tasks that execute in parallel: ReceiverTask, GossipTask, BroadcastTask and CleanupTask. An initialization procedure must also be
executed. The following sections describe in detail these tasks as well as the data structures used.

A. Data Structures

The HeartBeat Table, or HBTable is the most important of the data structures used by the detection algorithm. This table is used to store the heartbeat values received from other processes, and also records the last local time instant each entry was updated. Each heartbeat is represented as a positive integer. HBTable is implemented as a hash table that uses the process identifier as its key, called ID, plus a tuple of two integers, the heartbeat and the timestamp corresponding to the last update. The notation \(<hbvalue, tstamp>\) is used to represent this tuple.

A HBTable provides five basic operations: update(ID, hbvalue), get hbvalue(ID), get_timestamp(ID), size() and get_ids(). The update(ID, hbvalue) operation, when executed, verifies if the hbvalue received is larger then the one stored for the given ID. If it is, the new value is stored and the timestamp is updated with the current time. If the HBTable does not keep an entry for the given ID, a new entry is created with the hbvalue and the timestamp for the current time. The get hbvalue(ID) and get_timestamp(ID) operations return, respectively, the hbvalue and timestamp for a given ID. Finally, size() returns the number of entries in the table and get_ids() returns the set of IDs stored as table keys.

Each instance of the algorithm uses one HBTable and two integers: localHB is the current heartbeat value for the local process and timeOfLastBcast is the time of the last broadcast message received. timeOfLastBcast is used to decide when a process should execute a new broadcast, as described below.

B. Initialization

When the algorithm starts, an initialization procedure is executed. This procedure initializes the data structures and starts the other routines of the detection algorithm. First, a HBTable is created and the localHB and timeOfLastBcast values are set to 0. Next, the other tasks of the algorithm are started and execute in parallel.

C. ReceiverTask

The ReceiverTask routine is executed every time a gossip message is received, including broadcast messages. Each gossip message is composed of a set of \(<ID, hbvalue>\) tuples, each one representing the heartbeat value for a specific process. When a gossip message arrives, the update(ID, hbvalue) operation of the HBTable is called for each one of the tuples, updating the heartbeat and timestamp information in the table. When the received message is from a broadcast, the timeOfLastBcast value is updated.

D. GossipTask

The GossipTask routine is executed periodically, every GOSSIP_INTERVAL time intervals. It is responsible for sending gossip messages to other processes. At each execution, it checks whether HBTable is empty. If it is, there is nothing to be done, as no other process is known to the detector. If the entries in the table, FANOUT processes are chosen randomly from the set of known processes (or less if there are not enough entries in the HBTable). A gossip message is sent to each one of the chosen processes. The gossip message sent is built from the information stored in the HBTable. For every entry in the table, a \(<ID, hbvalue>\) tuple is added to the message. A tuple containing the local process ID and localHB value is also included in the message. After the messages are sent, the localHB value is incremented.

E. BroadcastTask

The BroadcastTask routine is executed periodically in order to allow system processes to find each other after they start up, and also to improve the speed in which the output of the detector stabilizes after the occurrence of multiple simultaneous failures. Broadcast messages are sent occasionally: each time a gossip message is to be sent, there is a chance that the message is broadcast to every process in the system, instead of a few random processes. The probability that a broadcast is performed is computed using the service parameters and the time the last broadcast message was received (timeOfLastBcast). This probability must be defined so that it avoids too frequent or too many simultaneous broadcasts.

The JXTA-FD service uses the broadcast probability proposed in [12], \(p(t) = \frac{t}{BCAST\_MAX\_PERIOD}BCAST\_FACTOR\), where \(t\) is the number of time units from the last broadcast received and BCAST_MAX_PERIOD and BCAST_FACTOR are algorithm parameters described in the following. Every time BroadcastTask is executed, a broadcast message is sent with probability \(p(t)\). In this way, the mean time between broadcasts depends on the frequency in which the gossip routine is executed (controlled by the BCAST_TASK_INTERVAL parameter), the number of processes in the system and the algorithm parameters. BCAST_MAX_PERIOD is the maximum interval between each broadcast: as \(t\) approaches this value, the probability \(p(t)\) approaches 1. BCAST_FACTOR is a positive floating point number and controls how close to BCAST_MAX_PERIOD the broadcasts tend to occur. The higher the BCAST_FACTOR value, the closer to BCAST_MAX_PERIOD the broadcasts are sent.

As an example, for a BCAST_TASK_INTERVAL of 1 time unit, BCAST_MAX_PERIOD of 20 time units and a system with 1000 running processes, to get an interval of approximately 10 time units between broadcasts, BCAST_FACTOR should be set to approximately 10.43 [12].

F. CleanupTask

The CleanupTask routine is responsible for the removal of old entries from the local HBTable. At every CLEANUP_INTERVAL time units, entries from the table that have not been updated in less than REMOVE_TIME
time units are removed. This mechanism is important because suspect processes are also used as targets for the gossip messages, and a table with too many invalid entries (crashed processes) will have a bad impact on detection accuracy. The $REMOVE\_TIME$ value is an algorithm parameter.

G. Detector Output

At any moment during the execution of the algorithm, a process can query its detector for the set of processes considered suspect (or the set of correct processes). To determine which processes are suspect, every entry in the $HBTable$ is examined and the time from its last update is computed. If this value is larger than or equal to the $SUSPECT\_TIME$ value, the process corresponding to this entry is considered to be suspect. Otherwise, the process is considered correct.

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

The JXTA-FD service was implemented as a Module for the JXTA platform, version 2.5, using the Java language. Process (or peer) monitoring is done in the context of a Peer Group. The JXTA-FD module must be loaded and started for every group that is supposed to be monitored. Only peers that are executing the module participate in the algorithm. At any moment, a peer can query its detection module for the list of processes considered suspect or correct.

A number of parameters are available for configuring the behavior of the algorithm. The most important are: $GOSSIP\_INTERVAL$, $FANOUT$ and $SUSPECT\_TIME$. The first parameter ($GOSSIP\_INTERVAL$) controls the interval in which gossip messages are sent by the $GossipTask$ routine. The second parameter ($FANOUT$) controls the number of gossip messages that are sent at each interval. Finally, $SUSPECT\_TIME$ represents the interval after which a silent peer is considered to be suspect. In other words, if no heartbeat is received from a peer for a time interval that lasts $SUSPECT\_TIME$ time units, the process state is toggled to suspect. The service parameters for a given group must be specified before the group is initialized.

A. Experimental Results

To evaluate the proposed failure detection service, the empirical study included both experiments executed with the JXTA implementation and a simulator which was implemented using a discrete events library called SMPL [7].

Two strategies for configuring the detector were evaluated. In the first strategy, on each execution of the gossip task, only one gossip message is sent. To increase the detector accuracy, the interval between sending gossip messages ($GOSSIP\_INTERVAL$ parameter) is decreased, i.e. a shorter gossip interval is employed. In the second strategy, the interval between sending gossip messages is fixed, so in order to increase the detector accuracy, more gossip messages ($FANOUT$ parameter) are sent at each interval. The two strategies were compared while using the same bandwidth, that is, the number of $<ID, hbvalue>$ tuples sent in a given time interval is the same for both strategies. These strategies are represented in every figure as $Gossip$ and $Fanout$, respectively.

To simulate the delay and loss of messages, a simple mechanism that drops a percentage of the messages received was implemented. Each message has a chance of being discarded. This mechanism was adopted to simplify the implementation and analysis of the results, given that sufficiently delayed messages have the same impact as lost messages on the detection accuracy.

B. JXTA Implementation Results

The experiments were conducted for a group of peers executing on one host. The mechanism of dropping messages is used to represent network delay and lost messages. The charts shown here are presented for a confidence interval of 95%.

CPU and Memory Usage: The objective of these experiments was to evaluate the use of CPU and memory by the detection service and the JXTA platform. The experiments were conducted on an Intel Core2 Quad Q9400 machine, 2.66GHz, with 4GB of memory. Each experiment was run for 15 minutes in which 10 peers ran the detection service. At every second, both CPU and memory usage data was sampled. Every peer employed the same parameters to configure the service. The time to suspect a given peer, $SUSPECT\_TIME$, was set to 5 seconds. The time for removal of old entries in the process table, $REMOVE\_TIME$, is 20 seconds. Each peer queries its detection service at intervals of 1 second. The values for the $BCAST\_MAX\_PERIOD$ and $BCAST\_FACTOR$ parameters were, respectively, 20 and 4.764.

Figure 2(a) shows how the detector parameters affect CPU usage. The chart shows that the used of CPU is proportional to the bandwidth used by the detection service. From these results one can conclude that there is just a little difference can be seen between the two strategies. It is possible that, given a very large number of peers, the difference could be more expressive. Figure 2(b) shows that the memory usage is also proportional to the bandwidth used by the detector.

These results show that the JXTA platform uses a considerable amount of resources, given that this should be a basic service upon which other more interesting applications are built. It should be noted that each peer also runs on its own JVM and that causes considerable overhead.

Mistake Probability: These experiments were executed to evaluate the impact of the detection parameters and the number of failures on the number of mistakes the detector makes. A mistake occurs when a working peer is considered to be suspect by some other peer. In these experiments peers never failed. In this way, every suspicion by some detector is a mistake. The scenario for these experiments is the same as the previous ones, tests are executed for 10 peers and the service parameters are the same.

Figure 3(a) shows the probability of a given query returning a mistake for message loss rate of 30%. It is possible to see that increasing the bandwidth used by the two strategies also increases the accuracy of the detection. The probability of a mistake for a bandwidth value of 12.5 is approximately
Fig. 2. (a) Impact of service parameters on CPU usage. (b) Impact of service parameters on memory usage.

0.02700 for the Gossip strategy and 0.03019 for the Fanout strategy. For a bandwidth value of 25, the values are 0.00015 and 0.00035, respectively. For a bandwidth value of 50, no mistakes were made. As the chart shows, for such a small group of peers, the difference between the two strategies is not very expressive. Even so, the Gossip strategy presented better accuracy, making approximately 50% less mistakes for a bandwidth value of 25.

Figure 3(b) shows the impact of message loss on number of mistakes made by the detector. The results show that the Gossip strategy is a little more resilient to message losses. Given loss rates of 20% and 30%, the number of mistakes is approximately 10% smaller than the number for the Fanout strategy.

Detection and Recovery Time: These experiments have the objective of verifying the difference in the detection time and the recovery time for the two proposed configuration strategies. The detection time is the mean time between the failure of a given process and the time it takes for another process to suspect it. The recovery time is mean time between a process recovering from a failure and the time it takes for another process to stop suspecting it. The recovery time can also be seen as the time it takes for a new process to be discovered by another process.

The tests were executed with the same configuration as the previous experiments. There is no message loss. In a given moment, one peer ceases its execution. It resumes its execution 10 seconds later.

Figure 4(a) shows the detection time for different values of bandwidth used. The results show a wide variation in detection time. For low bandwidth values, the variation is probably due to the large number of mistakes. A given peer might already be suspect when it actually fails. The chart also shows that the detection time for the Fanout strategy is approximately 20% smaller for a bandwidth value of 50.

Figure 4(b) shows the recovery time for different bandwidth values. In this case, the Gossip strategy is superior, having
Fig. 4. (a) Detection time, for different bandwidth values. (b) Recovery time, for different bandwidth values.

A recovery time approximately 50% smaller than the other strategy, for bandwidth value of 50. This difference is probably due to the higher frequency of updates. It can also be seen that the recovery time is directly affected by the bandwidth used.

C. Simulation Results

The simulation experiments were executed so that the detection algorithm could be evaluated for a larger number of processes and without the overhead of the JXTA platform on the results.

The experiments were conducted for a group of 200 peers. Some parameters are fixed for all the experiments. The SUSPECT_TIME value is 5 time units and the REMOVE_TIME value is 20 time units. The detectors are queried every 0.25 time units. The BroadcastTask routine is executed every 1 time units, and the values BCAST_MAX_PERIOD and BCAST_FACTOR are 20 and 8.2, respectively. This causes a broadcast to be executed approximately every 10 time units. The Gossip strategy keeps the FANOUT value as 1 and decreases the GOSSIP_INTERVAL, while the Fanout strategy keeps the GOSSIP_INTERVAL as 2 time units and increases the FANOUT value.

Mistake Probability: This experiment was run to evaluate the detector accuracy for different values of bandwidth. No peer fails during this experiment, in this way every suspicion is a mistake.

Fig. 5. (a) Mistake probability for different bandwidth values. Experiment done with 50% of the message being dropped. (b) Mistake probability for different percentages of message loss.

Figure 5(a) shows how the bandwidth used affects the quantity of mistakes made by the detector. 50% of the messages are dropped. The chart shows that increasing the frequency of the gossip messages has a much larger impact on the detection accuracy than increasing the FANOUT. In some cases, the Gossip strategy is an order of magnitude better than the Fanout strategy.

In Figure 5(b), the impact of message loss on the accuracy of the detector is shown. Bandwidth is equal to approximately...
550 (FANOUT is 5 and GOSSIP_INTERVAL is 0.4 time units). The chart shows that the Gossip strategy is much better in preventing mistakes for every value of bandwidth used.

These results, together with the results from the JXTA experiments, show again that the Gossip strategy is much superior to the Fanout strategy in terms of detection accuracy. They also show that the difference between the two strategies becomes even larger as the number of processes in the group increases.

**Leader Election**

This experiment shows the use of the proposed detection service from the application point of view. A simple simple leader election algorithm that uses the detector was implemented. This simulation uses the same parameter values as the other experiments. The election is distributed and probabilistic, each process elects the process (including itself) with the smallest identifier that is not suspect as the leader of the group. At every time unit, each peer queries its detector to find the current leader. For the election to be considered successful, every process must choose the same leader. No process fails during this experiment.

Figure 6(a) shows the probability of a given election being successful for different amounts of bandwidth. Messages have a 50% chance of being dropped. The chart shows that the Gossip strategy is better than the Fanout strategy. The result also shows that the a higher amount of bandwidth (and thus lower mistake probability) is required to ensure a probability that the election succeeds.

In Figure 6(b), the election success probability is shown for different percentages of message loss. The bandwidth value is constant and equal to approximately 650 (FANOUT is 6 and GOSSIP_INTERVAL is 0.332 time units). The figure shows that message loss has a large impact on election results. Also, the Gossip strategy has, again, better results than the Fanout strategy.

These results show that in this probabilistic leader election, the higher the bandwidth used, the higher the chance of a successful election. This is expected, as it reduces the mistake probability. Results also do not leave a doubt that the Gossip strategy has better results than the Fanout strategy, both using exactly the same amount of bandwidth.

**D. Discussion**

The decision of using the JXTA platform was motivated by the resources the platform was supposed to offer for peer-to-peer application development. In particular, being able to use the relay and rendezvous mechanisms, as well as multicast support (propagated pipes) were highly anticipated. Reality proved to be very different from what was expected of the platform. Although we tried hard, going through the available documentation (which was mostly outdated) and mailing lists, the JXTA relays and rendezvous could not be made to work correctly, communication among hosts in different networks was simply not possible. One option we had (which was used while the service was being implemented) was to connect to public relays and rendezvous provided by the JXTA community, but those were not available when the experiments were executed. Since we ran our peers on a single host, we had another problem due to our choice of using the Java implementation: each peer required its own JVM. Since the amount of resources needed for each process in this case is very high, our JXTA experiments had to be limited to a small number of peers.

V. CONCLUSIONS AND FUTURE WORK

This work described the specification, implementation and evaluation of a failure detection service based on a gossip strategy. Processes that implement and run the service can query its detector for information about the state of other processes in its detection group. The proposed service was implemented on the JXTA P2P platform as a proof of concept. A simulator was also implemented using the SMPL library for experiments with a larger number of processes.

The detection service was evaluated through experiments on both the JXTA platform and the simulator. Experimental
results show that the gossip algorithm scales well as the number of processes in the group grows, and that it is robust in terms of amount of mistakes it makes. Results also show that increasing the frequency in which gossip messages are sent gives much better detection accuracy than increasing the fanout of the algorithm; both strategies use exactly the same amount of bandwidth. As the number of processes grow, the difference becomes even more significant.

Future work thus includes implementing the failure detection service on another platform, one that would allow processes running on multiple hosts in different networks connected through the Internet to be monitored. Several Internet applications involve agreement problems, and future work also includes the implementation of a consensus algorithm, such as Paxos, [13] based on the proposed failure detection service.

ACKNOWLEDGMENTS

This work was partially supported by grant 304013/2009-9 from the Brazilian Research Agency (CNPq).

REFERENCES