A Failure Detection System for Large Scale Distributed Systems

Andrei Lavinia*, Ciprian Dobre*, Florin Pop*, Valentin Cristea*
*Computer Science Department, Faculty of Automatic Control and Computers Science, University POLITEHNICA of Bucharest
(e-mails: me_andrei_lavinia@yahoo.com, {ciprian.dobre, florin.pop, valentin.cristea}@cs.pub.ro)

Abstract

Failure detection is a fundamental building block for ensuring fault tolerance in large scale distributed systems. In this paper we present an innovative solution to this problem. The approach is based on adaptive, decentralized failure detectors, capable of working asynchronous and independent on the application flow. The proposed failure detectors are based on clustering, the use of a gossip-based algorithm for detection at local level and the use of a hierarchical structure among clusters of detectors along which traffic is channeled. In this we present result proving that the system is able to scale to a large number of nodes, while still considering the QoS requirements of both applications and resources, and it includes the fault tolerance and system orchestration mechanisms, added in order to assess the reliability and availability of distributed systems in an autonomic manner.

1. Introduction

Large scale distributed systems are hardly ever “perfect”. Due to their complexity, it is extremely difficult to produce flawless designed distributed systems. While until recently the research in the distributed systems domain has mainly targeted the development of functional infrastructures, today new requirements have emerged for large scale distributed systems; among these requirements, fault tolerance is needed by more and more modern distributed applications, not only by the critical ones. The clients expect them to work despite possible faults occurring.

Although the importance of fault tolerance is today widely recognized and many research projects have been initiated recently in this domain, the existing systems often offer only partial solutions that follow a particular underlying distributed architecture. Traditional fault detection solutions, in particular, fail to work properly in the context of large scale distributed systems because of the large number of monitored processes involved, the high probability of messages being lost, the dynamic nature of the underlying topologies and the unpredictable latencies of the message deliveries.

In this paper we present an innovative solution to solving the requirements involved in obtaining a failure detector designed for large scale distributed systems. The solution is based on the results presented in [1]. The failure detection system is particularly designed for highly-dynamic large scale distributed systems. Its architecture allows applications to specify different QoS detection levels, while offering scalability, generality and non-intrusive characteristics.

The failure detection consists in monitoring processes and stations throughout the systems and detecting errors in the shortest time according to the fault tolerance requirements of the distributed applications. The interpretation of the monitoring information is based on level of suspicion and the progressive detection feature brings a new approach in terms of predicting the time of arrival of the next heartbeat message, as well as interpreting and updating the level of suspicion. The system is able to deal with both transitional and permanent errors.

The implementation of the system is based on two strategies. The first one uses a hierarchical approach based on dynamic clustering to solve the scalability issue. The second strategy leverages the gossiping technique in order to remove wrong suspicions and to decrease the time needed to detect errors.

The solution proposed in this paper combines the advantages of existing approaches to minimize their limitations in order to effectively treat problems such as the explosion of messages, scalability, flexibility, dynamism and adaptation to the various requirements.
of fault tolerance requirement coming from applications and the variable network conditions.

The rest of this paper is structured as follows. Section 2 presents related failure detection solutions for distributed systems. The next section presents the proposed architecture and key elements of the implementation of a robust failure detector, highlighting the proposed models and protocols. We present next results demonstrating the validity and performances of the proposed solution. Finally, in Section 5 we present some conclusions and future work.

2. Related Work

A failure detector is widely recognized as an oracle that can intelligently suspect processes to have failed [2]. In distributed applications, failure detection is generally implemented through the use of directly invoked local services (unreliable local failure detectors). The general strategy consists in attaching to each processes of a distributed application a failure detection module.

The failure detection module works asynchronous and independent on the application flow and is responsible with monitoring a subset of the processes in the system and maintaining a list of those it currently suspects to have crashed. A process can query its local failure detector module at any time. Internally, the failure detector module maintains a list of suspect processes that he suspects are crashed. The suspect processes list is permanently updated such that, at any time, new processes can be added and old ones removed. The failure detector is considered unreliable because is allowed to make mistakes, to a certain degree [2]. A module can erroneously suspect some correct process (wrong suspicion) or can fail to detect processes that are already crashed. At any given time two failure detector modules may have different lists of processes.

The most common implementation of local failure detection is based on the heartbeat strategy. In this strategy every failure detector module periodically sends a heartbeat message to the other modules, to inform them that it is still alive. When a module fails to receive a heartbeat from another process for a predetermined amount of time (timeout) it concludes the remote process crashed. There is a tradeoff, however, for the timeout values being considered. If the timeout is short then crashes are detected quickly, but there is a high chance of suspecting of being crashed processes that takes a longer time to respond (due to a possible high load for example). Conversely, if the timeout is long, the chance of wrong suspicions if
A distinctive category of detectors is represented by the accrual failure detectors. The family of accrual failure detectors consists of detector modules that associate, to each of the monitored processes, a real number value that changes over time. One example of an implementation of an accrual failure detector is the $\phi$-failure detector [5]. The $\phi$-failure detector samples the arrival time of heartbeats and maintains a sliding window of the most recent samples. The window is used to estimate the arrival time of the next heartbeat. A similar approach was also proposed in [8]. However, the proposed failure detectors are poorly adapted to very conservative failure detection because of their vulnerability to message losses. In practice message losses tend to be strongly correlated (i.e., losses tend to occur in bursts). A proposed accrual detector designed to handle this problem is the $k$-failure detector [9]. The $k$-failure detector takes into account both messages losses and short-lived network partitions, each missed heartbeat contributing to raising the level of defined suspicion according to a predetermined scheme.

An important issue with failure detectors is their scalability. An approach that focuses on the scalability of failure detection was proposed in [8]. However, the proposed system assumes simpler failure semantics such as crash failures.

3. Architectural and Implementation Details of the Failure Detector

The proposed architecture is based on the idea of making the fault detector available as a service to applications. This means that any distributed application can use the failure detection capabilities of the failure detector. For that the system is composed of several failure detection agents running inside the distributed environment, each being responsible for the monitoring of a subset of processes and the update of the applications, through specialized modules, about the status of the monitored processes. The system is composed of four layers (Figure 1).

The communication layer is responsible with the clusterization mechanism. This layer is responsible with creating a scalable, fault tolerant and dynamic communication infrastructure for the upper-layer agents. Within a large scale distributed system the solution that envisions complete peer-monitoring of agents (each agent being responsible with the monitoring of every other agent in the system) is not feasible in terms of communication costs and scaling capabilities. The network and host resources are limited, while the geographical distribution of the system can be quite large. Therefore, we adopted a solution that groups agents in clusters, based on the geographical position of the workstations on which they are running, combined with costs (such as communication delays, load of the workstation, balancing cost, etc.) that together minimize the time needed to propagate updated among adjacent modules.

![Fig. 1. The architecture of the system.](image)

The system is therefore composed of several clusters. Within each cluster an agent has a coordination role, being responsible with the administration of the topology (adding/removing nodes as agents enter or exit the system, dividing or unifying clusters), with the communication management inside the cluster and with processing and disseminating messages inter-clusters, among cluster coordinators.

For fault tolerance within each cluster one agent acts as redundant secondary replica of the coordinator. This eliminates the possible single-point-of-failure represented by the existence of the unique coordinator. The replica is asynchronously notified of any operation that could result in a modification of the topology. The replica becomes the coordinator when other nodes discover that the old coordinator failed. The coordination exchange is executed quickly and without loss of information - the unfinished operations (assuming for example that the last coordinator crashed unexpectedly) are evaluated and re-executed in order to bring the cluster to a consistent state.

At the cluster level the coordinator acts as a gateway, representing the only point of entrance within the cluster. This forms an hierarchical interconnecting network that ensures an optimal control scheme.

At this level there is also a registry and lookup (LUS) service that is used to store information about the modules running inside the system. At initialization a module tries to estimate the best cluster to which to connect. For that the agent will interrogate the LUS service to find current coordinators, will order the received list based on the round-trip time between him and the agents and based on the cluster sizes, and in the
end will choose the nearest cluster having the smallest dimension.

The number of coordinators and the size of their associated clusters are essential to maintaining efficiency. If the number of agents is too high compared to the number of coordinators, the cluster-level information becomes hard to manage, whilst if the number of agents is much smaller than the number of coordinators, the number of information and control messages exchanged between clusters increases greatly.

The solution consists in allowing the number of coordinators to dynamically adjust as the number of agents increases (new agents join the system) or decreases (agents exit the system). Thus, when the number of agents in a cluster is too large, a secondary coordinator is elected among agents. The current cluster is divided into two clusters of equal size and the agents in the new cluster are notified of the new coordinator. Then the new coordinator registers itself as a coordinator and establishes a new replica inside the newly formed cluster.

Fig. 2. The creation of a new cluster.

When the number of agents in a cluster becomes too small the cluster is eliminated by merging it with an adjacent cluster. The coordinator of the newly merged cluster issues a request to the other cluster coordinator. This results in a merge of the known topologies, followed by the notification of the members of the updated topology. The old cluster coordinator also becomes the secondary replica of the newly merged cluster.

Fig. 3. The merging of two clusters.

At the monitoring level each detection module is responsible with proper monitoring and logging of monitoring data. A registered application may require starting or stopping of monitoring of a certain process. The monitoring is based on a pull model. The agent periodically sends to the agent associated with the monitored object heartbeat messages and samples the timestamps of the received answers. Periodically, at each $T_{\text{monitor}}$ seconds, he traverses the list of monitored objects and for each object he sends a heartbeat message to the associated agent and, based on the received response, it updates the associated suspicion level. An update of the suspicion level also triggers the logging of the current status.

We also assume that processes can experience permanent or transient failures, and that they can experience high loads. A detection module can decide that a process has failed due to several factors, such as the failure of the workstation on which the agent was running, the loss of a direct communication link with the monitored object, an increase in traffic resulting in the delayed responses from the questioned agents, or a high load of the workstation where the agent is running resulting in an increase in the response time of the agent which can be incorrectly detected as a failure.

These aspects lead to wrong suspicions of failures, even when using a sophisticated detection algorithm. In an unstable network, with frequent losses of messages or with processes that fail often, the detection algorithm is almost irrelevant, because the agents can not distinguish between permanent and transient failures. The third level of the architecture attempts to solve this deficiency by using a gossiping algorithm. This increases the confidence of an agent in the failure of a process by using a schema to notify agents about the failures of processes they consider running correctly or to remove false suspicions.

The next layer is responsible with the interpretation of the monitored data, meaning the analysis and processing of that data. This results in an update of the status of a particular process, as known by the local failure detector.

Fig. 4. Information flow in the failure detector.

Each heartbeat that was not received contributes partly to the suspicion level of the failure detector. The contribution of a heartbeat $H$ increases from 0, meaning that $H$ is not yet expected, to 1, meaning that
$H$ is considered lost. The suspicion level is computed as a sum of all contributions. But, unlike previously other existing implementations of accrual detectors, the suspicious level in this case is not computed only from the local heartbeat contributions, but also from contributions received from other failure detectors located in the local cluster.

The contribution function is computed as follows. Each failure detector maintains a local suspicious level value $s_{\text{qp}}(t)$, computed as:

$$s_{\text{qp}}(t) = \frac{t-1}{t+1}, \text{ where } t = \frac{t_{\text{now}}}{t_{\text{pred}}}$$

(1)

The heartbeat messages are sampled by the detector in order to estimate the time when the next heartbeat is expected to arrive. For that the detector can use any of several prediction methods [1]. After several tests, we decided on the use of a modified version of exponential moving average (EMA) to correct smoothing factor based on the latest developments in current values.

The gossiping protocol is next responsible with the exchange of information among the nodes of the system. A traditional gossip protocol is based on a probabilistic model in which nodes randomly choose partners with whom they exchange information, leading to uneven distribution of monitoring information. Some agents often receive information messages, while others receive the information rarely enough to wrongly suspect the monitored processes. But, on the other hand, a traditional gossiping algorithm has several advantages: a simple implementation, a fast dissemination of the monitoring data, no need to know the adjacent topology, the independence on the dimension of the clusters or on the topology changes, and it does not depend on a certain agent.

Because the detection system we were not able to use the traditional approach, we developed our own modified gossip protocol. First we were forced to reconsider the limitations imposed by the communication layer: it limits the initial advantage of the independence from the topology changes. In our approach no agent except for the coordinator and the secondary replica should know the topology. This is adequate for minimizing the number of messages exchanges inside the cluster, but also for a traditional gossip algorithm would require building supplementary mechanisms to treat the inconsistency generated by the topology changes. This could further lead to a situation where agents in different clusters would communicate directly. On the other hand, because a probabilistic broadcast of information is not acceptable because of the penalties in the detection time and the accuracy of the detection, we needed to create an optimized algorithm to uniformly disseminate the gossiping information. This is difficult to accomplish in an environment in which each agent know only a subset of agents (the agents from which it received gossip messages).

The designed solution consists in partitioning the cluster in groups of agents of a predetermined size (power of two). At each group level we apply a simple gossip algorithm, having the following properties. Each agent has a local view which changes dynamically, depending on the topology updates. Each agent from the same group has the same local view. Each agent also maintains a local list with gossiping information. A record of this list contains the identifier of the agent, the associated suspicion level and the timestamp of the last update. Each agent also maintains an internal counter to indicate the current round. The counter is reset when reaching the number of rounds necessary to disseminate the gossiping information to all members of the group. Periodically, at each $T_{\text{gossip}}$ seconds, each agent updates the local list by modifying the suspicion level for each entry, according to the formula (1). Then it chooses a partner from the local view, based on a Round-Robin algorithm, to which it sends the local list. Periodically, at each $T_{\text{broadcast}}$ second, each agent sends a broadcast request to the coordinator with the local gossiping information. The coordinator then broadcasts further the received data to all agents in the other groups. Upon receiving of a gossip message, each agent combines the local list with the one received, choosing the minimum suspicion level received on each entry (if the received value was modified at a later time compared with the last local update). At each $T_{\text{gossip}}$ each agent also verifies the local list. If there are no informing messages from the agent from the group for more that $T_{\text{cleanup}}$ seconds and the associated suspicion level is sufficiently large it considers that the agent failed.

Finally, the last layer is represented by the service capabilities being provided to various applications running on top of the large scale distributed system. As in case of accrual failure detectors, we provide a complete decoupling between monitoring and interpretation. The failure detection architecture follows the SOA approach, the applications being able to send requests regarding current suspicious levels of failures for certain processes from the failure detectors services using a standardized service approach. Also, this approach has the advantage of coping well with various existing service-based middleware platforms.

The architecture is designed to scale well and provide timely detection. For that, we combine the advantages of several proposed failure detection solutions. We believe that, in order to cope with the
large scale nature of today’s distributed systems, a failure detector must scale well and also the probability of false detections must not be influence by the number of monitored processes. For that, the gossip-based protocol provides several advantages: the probability that a member is falsely reported as having failed is minimized; the algorithm scales well and delivers minimum detection times and network loads. We combined these properties with those introduced by the accrual detectors. Such detectors provide a lower-level abstraction that avoids the interpretation of monitoring information (Figure 5).

A value that represents a suspicion level is associated with each process, which is then left to the application to be interpreted. In this way a real-time application could take quicker decision on processes being considered failed, while application requiring a high-level of confidence in their decisions (such as a data warehouse synchronization service) might require higher level of confidence that a process really failed. By setting an appropriate threshold, applications can then trigger suspicions and perform appropriate actions.

4. Results

In order to evaluate the developed detection system we considered two scenarios. The first one tested the actual functionality of the system, while the second evaluated its performances. For the scenarios we started from the idea that an agent must correctly detect failures in a reasonable time (scenario 1) and use a minimum amount of resources on the workstation where it runs, plus it should maintain a low number of messages sent for monitoring and control (scenario 2).

The first scenario consisted of a number of small set of simple tests that evaluated the correct detection of failures. The idea is to simulate both permanent (forced crash of an agent) and transient (temporary interruptions in the network connections to simulate the loss of messages) faults. We also tested certain configuration values for parameters such as the sliding window sizes used for computing storage times of the heartbeat messages and the generation rate for the gossip messages.

The testbed involved three stations, two located in Switzerland and one in Romania. In the beginning we assumed two agents running on each of the nodes in Switzerland, and on the node in Romania we started the Jini service. For this test we assumed a dimension of the gossip group of four.

The four agents form a cluster and also a group of gossip. At one point, an agent was broken to see how the agents would react to its failure. The study showed how the window size influences the evolution of the suspicion level, because the smoothing factor used for prediction is closely related to the number of current analyzed values. In addition, it was important to assess the impact that a particular rate of spread of rumors has on the time of detection.

The first test showed that the evolution of the suspicion level associated with the agent dropped to the level of the three functional agents. For that we considered different valued for the dimension of the window (5, 10, 15) and various generation rates for the gossip messages (4, 6, 8, 16 seconds). The results showed that the failure was correctly detected in less than 20 minutes, which is a realistic result for a system where suspicions must follow a balanced grow to allow stations affected of transient faults to eventually recover from errors. The results also show an increase in the detection time as the rate of generating gossip messages increases.

Therefore, the detection time can be adjusted accordingly by adjusting the rate of spreading rumors. In addition, the graph in Figure 6 show that for a
window size of 15 and a low gossip interval we obtained the lowest detection time, while for larger rates of gossiping a smaller window size is better to be used, especially when we are interested in a faster speed of detection. Choosing a particular value for a configuration parameter largely depends on what we want to achieve through detection.

These results showed that the level of suspicion reaches, in a reasonable time, the threshold of 0.7, which indicates a fairly high probability of failure. The evolution beyond this threshold is slower, which ensures the settlement of workstations that fail often. Therefore, the algorithm for updating the level of suspicion successfully ensures greater confidence in a failure of a process.

The second test evaluated the evolution of the suspicion level based on monitoring data. Here we considered three applications using the detection service: one application registers itself with the system, while the other two monitor it. At certain moments the applications query their detection modules about the status of the third monitored process.

The second series of experiments assumed the use of LISA, a lightweight dynamic service that provides complete system and application monitoring [10]. In this case the objective was to evaluate the network traffic and load of the system on which the agents are running in order to demonstrate the non-intrusive characteristic of the failure-detection system.

Fig. 8. The results for the evolution of the network traffic.

These tests were performed using a gossiping rate of, in terms, four seconds and eight seconds, and demonstrate that the overhead caused by running the agents is very low, both in terms of network traffic and CPU usage or system load.

Fig. 9. The results for the load.

The plot in Figure 8 shows that the peak traffic for this test case did not exceed 1.20 Mbps, for a gossiping rate of eight seconds. The observed increase in traffic corresponds to the entry of new agents in the cluster, which led to the reconfiguration of the groups and the transmission of data to the affected members. When the detection system stabilizes there is a decrease in the network traffic, even in the context of the enlargement obtained from gossiping, in large systems this is the only way to deal with transient errors, so it is important that the provided information is correct and as stringent as possible.
of the gossip group. Also, the CPU remains idle at a rate of 97.05%.

As a conclusion, the experimental results show that the detection system does not affect the performance of the system on which it runs. There are small increases in the load of the system observed for reconfigurations (when an agent enters or leaves the cluster), but even in this case we demonstrated a small load on the workstations, even for frequent changes of topology. In addition, the network is not overloaded with control and monitoring messages.

5. Conclusions

As society increasingly becomes dependent of distributed systems (Grid, P2P, network-based), it is becoming more and more imperative to engineer solutions to achieve reasonable levels of dependability for such systems. Failure detection constitutes a fundamental abstraction for fault tolerant distributed systems.

In this paper we presented a failure detection system that combines the power of existing approaches: fast propagation of information as offered by gossip-based failure detection approaches together with the decoupling of monitoring and interpretation as offered by the accrual failure detection solutions. The solution is based on the use of prediction functions and a new alternative of computing the contribution function. The approach has several advantages, among which we mention a better estimation of the interarrival times of heartbeat messages and an increase level of confidence in the suspicions of processes being lost.

The approach considers both the various networking conditions of large scale distributed systems and the different QoS detection requirements coming from various applications. In our approach the interpretation of the suspicion level is left to the distributed application using it. In this way multiple applications, having different QoS requirements, use the same failure detectors in different ways. The application could take either conservative (slow and accurate) or aggressive (fast, but inaccurate) decisions.

In the future, we aim to fully deploy this solution in various existing system and compare the obtained performances against other existing solutions. We also plan to extend the architecture in order to include not only detection capabilities, but also means to allow application to automatically asses various recovery and masking (redundancy) mechanisms.

Acknowledgments

The research presented in this paper is supported by Romanian national project “DEPSYS – Models and Techniques for ensuring reliability, safety, availability and security of Large Scale Distributed Systems”, Project “CNCSIS-IDEI” ID: 1710.

6. References


