Proc. of II Workshop on Tests and Fault-Tolerance (II WTF 2000), pp. 76-81, July 2000, Curitiba, Brazil, Brazilian Computer Society

Failure Detection in Asynchronous Distributed Systems

Raimundo José de Araújo Macêdo

Laboratório de Sistemas Distribuídos – LaSiD
Departamento de Ciência da Computação
Universidade Federal da Bahia
Campus de Ondina, CEP: 40170-110, Salvador-BA, Brazil
e-mail : macedo@ufba.br

Abstract

Being able to detect failures is an important issue in designing fault-tolerant distributed systems. However, the actual behaviour of a system limits the ability to provide such a mechanism. From one extreme of the spectrum, synchronous systems (i.e., with bounded message transmission delay and processing times) allow for the construction of perfect failure detection based simply on local timeouts. At the other extreme, accurate failure detection cannot be developed for asynchronous systems (i.e. systems with no bounds on message transmission delays and processing times), unless some extra properties can be guaranteed, such the ones specified in a seminal article by Chandra and Toueg [1]. The present paper discusses the requirements and describes the implementations of failure detectors for two important fault-tolerant mechanisms meant to asynchronous environments: process group membership and <>S Failure Detector based distributed consensus [1]. These implementations are based on a mechanism called the Time Connectivity Indicator, introduced in this paper.

Key Words : failure detectors, fault-tolerance, distributed systems, consensus, membership

1. Introduction

A distributed system is defined as a collection of processes running on a set of networked, possibly geographically spread, computers. Nowadays, mainly after the widespread use of the World Wide Web, the dependence of society on such systems have become commonplace. This reality have pushed researchers to find out techniques for building reliable distributed systems which can deliver the specified services despite failures of some of its components. System Diagnosis[14,15], Byzantine Agreement[16], and Processor Membership[17] are three well known techniques which have successfully been used in the past to develop such reliable or fault-tolerant systems. Although these techniques are closely related, they have distinct specifications and purposes. System Diagnosis aim at determining the sets of fault-free and faulty processors[1]. The set of faulty processors, called the syndrome, is diagnosed through a two-phase procedure: the test phase where processors are tested, followed by a diffusion phase where test results are exchanged among fault-free processors and the syndrome established. Processor membership is similar to system diagnosis in that both must maintain a list of which processors are faulty and which are not. However, in the processor membership approach, there is not a explicit test (which can cover time and value faults). A processor is taken as faulty if it fails in transmit and forward messages in a timely manner. The goal of Byzantine Agreement in its turn is to mask processor faults by having enough processors doing the same task. A majority vote scheme is used to achieve consensus between the faulty-free processors which allows the fault results be masked out. Thus, Byzantine Agreement is primarily involved with fault masking

[1] The techniques are equally valid for a distributed system of PROCESSES

whereas system diagnosis with fault detection. In order to keep the specified level of reliability of a Byzantine Agreement mechanism, faulty processors should eventually be replaced by fault-free ones. Therefore, a fault detection mechanism is always required in any of these techniques (though, it not crucial for the Byzantine Agreement case).

Unfortunately, because the above mentioned techniques require a synchronous behaved environment, they cannot be directly applied to distributed systems running on settings such as the Internet, where message transmission delay and processing times cannot be known and bounded. For these environments, called asynchronous, new techniques have emerged. Amongst them, process group membership[5] and distributed consensus[1] are prominent. They can be thought of as the counterparts of Processor Membership and Byzantine Agreement for Synchronous Systems, respectively. While timeout values can be used to implement failure detection in the Processor Membership and Byzantine Agreement mechanisms, they cannot directly be taken as an accurate indication to such failures in the asynchronous world.

Although many protocols for group membership and distributed consensus have been proposed in the recent years, the actual implementation of failure detectors have deserved little attention in the literature. Yet, failure detection is a fundamental issue for process group membership and distributed consensus. This paper discusses the requirements and describes the implementations of failure detectors for the two mentioned fault-tolerant mechanisms meant to asynchronous environments: process group membership and <>S Failure Detector based distributed consensus [1]. The implementations presented are based on a mechanism called the Time Connectivity Indicator, briefly introduced in this paper, and fully described in other documents[20,19]. The remaining of the paper is organised as follows: section 2 and 3 discuss the requirements to implement failure detectors for group membership and distributed consensus, respectively. Section 4 outlines the design of a software tool used to estimate timeout values, and section 5 shows how this tool can base the development of the failure detectors discussed in sections 2 and 3. Finally, section 6 presents some concluding remarks.

2. Failure Detection in Group Communication

A process group is an entity which an application process refers to without knowing the number and location of the members which form it. Essentially, in a group communication mechanism, a message sent by a process must be addressed to all processes in the group and message delivery must be atomic: either all processes receive the message or no one receive it.

When a group G is initially formed, the group membership service working on behalf of a group member installs an initial view $V_i^1 = \{P_1, P_2, ..., P_n\}$ of the group members. As process crashes occur, the Group membership Service will install subsequent views V_i^1, V_i^2,..., V_i^r by reaching "consensus" with all the other fault-free[2] processes on these new group views to be established.

It is well known, however, that consensus in an asynchronous system is impossible to achieve, even if processes fail only by crashing [3]. This happens because, due to the uncertainties on message transmission time, a functioning process cannot distinguish between a faulty process and slow one. Assuming that a slow process is crashed may lead the group membership service to create virtual partitions. The way partitions are dealt with, characterise the two main solutions to the process group membership problem for asynchronous systems: the primary-partition approaches [5,7,8,9] and the partitionable ones [4,6,10]. In both cases, processes belonging to a specific partition will eventually share a mutually consistent view of that (possibly virtual) partition. Primary-partition membership services are intended for systems with no network partitions, or for systems that support only one partition in the group, the primary partition. In such cases, processes at primary partition are able to make progress, whereas processes in the other partitions should be blocked. Partitionable membership services, on the other hand, allow multiple real (or virtual) partitions. In consequence, a group can be split into multiple disjoint concurrent subgroups. Processes in one subgroup cannot communicate with processes in other subgroups, and they proceed execution as if they were the only ones in the group.

[2] In fact, not suspected as crashed.

Lately, when communication is restored, the service must provide some merging mechanism to rejoin partitioned groups.

The *Group Membership* service (being it partitonable or not) of each process in the group must execute the following main functions: it will first suspect a process which does not seem to be active (i.e., sending messages within a predetermined timeout period) and, secondly, it launches an agreement protocol over the membership, trying to remove the suspected process from the group view. The outcome of this agreement protocol is that all functioning and connected processes agree either to eliminate the suspected process from their group membership views, or to drop the suspicion. If proper timeout values are not used, it may well happen that the group membership service will confirm false suspicions from the first phase, causing the virtual partitioning of the group. Therefore, the formation of virtual partitions, which could lead to multiple sub-groups in partitionable memberships, or the creation of minority partitions in the primary membership (causing the killing of the processes in that partition), can be mitigated by utilising an appropriate failure detection mechanism in the first phase.

3. Failure Detection in Distributed Consensus

The consensus problem can be informally defined in the following way. Each process proposes a value, and all fault-free processes have to agree on a common value which has to be one of the proposed values. The Consensus problem constitutes a basic building block on top of which solutions to practical agreement problems can be designed. A typical agreement problem is the Non-blocking Atomic Commitment (NBAC) where processes have to agree on a common outcome to a distributed transaction (namely, abort or commit). Solving this problem in asynchronous distributed systems where processes can crash is far from being a trivial task. More precisely - as mentioned before - it has been shown by Fischer, Lynch and Paterson [3] that there is no (deterministic) solution to this problem as soon as processes (even only one) may crash. The major advance proposed to circumvent this impossibility result lies in the Unreliable Failure Detector concept, proposed and investigated by Chandra, Hadzilacos and Toueg [1,2]. The weakest conditions that have to be satisfied to solve the consensus problem have been identified [2] and, accordingly, several protocols have been proposed to solve the consensus problem [1,13,18].

A failure detector is basically defined by two properties: a *completeness* property that is on the actual detection of failures, and an *accuracy* property that limits the mistakes a failure detector can make. Chandra and Toueg have defined several *completeness* and *accuracy* properties that allowed them to define eight classes of failure detectors. Among them, the class <>S is the most attractive since it imposes the weakest conditions on the run time environment (therefore, it is the more implementable). This class includes all the failure detectors that *satisfy strong completeness* (eventually, every crashed process is suspected by every correct process), and *eventual weak accuracy* (there is a time after which there is a correct process that is never suspected).

Several consensus protocols have been designed for asynchronous distributed systems equipped with a failure detector of the class <>S [1,11,12,18]. They all are based on the same iterative control structure: processes proceed in asynchronous rounds whose aim is to make them eventually converge to the same value (and then decide on it). Each round r is managed by a predetermined coordinator that tries to impose its current estimate of the decision value as the final decision value. Each process executes rounds sequentially, until it decides (or crashes). A process p progresses from a round r to r + 1 when it has sent a positive acknowledgement to the coordinator of r (accepting its estimation), or when it suspects it (it then sends it a negative acknowledgement). Only when a process p becomes the coordinator of a new round r , it has to synchronise with the other processes: it then waits for messages from the other processes. Those messages ensure that if a value v has been decided by other processes, p cannot start a new round with an estimate of the decision value different from v (agreement property).

The above mechanism will work as far as the properties specified for the failure detector <>S can actually be implemented for all executions of the protocol[3]. A simple timeout mechanism is

[3] In fact, if <>S does not hold, it is shown that though the decision can be postponed indefinitely (*liveness*), agreement (*safety*) will never be violated [1].

sufficient to implement the *completeness* property of failure detectors of class <> S[4]. However, the *weak accuracy* property is harder to implement. The next section introduces a mechanism, called the Connectivity Time Indicator, that will latter be used to implement the properties required by the failure detectors of class <>S.

4 The Connectivity Time Indicator - CTI

The connectivity time, *ct*, between two processes, Pi and Pj, is defined in this paper as the time duration for a message to travel from process Pi to process Pj (or vice-versa) for a given moment of the system live. In systems with varied loads (such as the Internet), *ct* may assume distinct values for a interval of time. It can vary from 0 (if i = j) up to infinite (if Pi and Pj are actually disconnected). The Connectivity Time Indicator – CTI is introduced as the mechanism capable of dynamically delivering the connectivity time. A complete description of CTI and its implementation in the CORBA/JAVA environment can be found elsewhere [20, 19]. In the following, a brief description of the CTI mechanism and related concepts are presented.

While it is impossible to predict the precise future (one cannot guarantee components will not fail and load will be constant), it is required a CTI which hints with the present connectivity time by carefully analysing the current operating system and network loads. That is, instead of taking the communication time from the application level, the connectivity time is estimated by the CTI from the network and operating system load analysis. There is one CTI running in each site of a distributed system and it will be constantly updating connectivity information related to local and remote processes. Before an application process Pi starting enquiring about connectivity time with Pj, it must first register the pair (Pi, Pj) in a local CTI database. From this point, the CTI database entry for (Pi,Pj) will be periodically updated with connectivity information about Pi and Pj *and* the local CTI will forward the pair (Pi, Pj) to the remote CTI where the process Pj reside.

5.0 Developing Failure Detectors from CTI's

5.1 System Model and Assumptions

It is assumed a distributed system of processes which are able to communicate with each other through reliable channels (i.e., a sent message will arrive at its destination as long as the sender process remains faulty-free). Processes fail only by crashing (halting execution) without producing any further actions. It is not assumed any bounds on communication or message processing delays (i.e., the system is asynchronous). It is also assumed the all the processes involved in a group communication or distributed consensus will be permanently exchanging “I am alive” or heartbeat messages. Due to space limitations, the description of the failure detectors, given below, covers only the main aspects of its implementations. Thus, aspects such as the updating of suspected lists are not presented in this paper. The following functions and parameters are assumed to be available:

1. ***Time-silence***$_i$***(j)*** denotes the time duration since process P_i last received a message (application related or heartbeat) from P_j;
2. ***justSuspected(i,j)*** is an array which denotes the fact that process P_i suspected process P_j and this suspicion has not been dropped;
3. λ is the amount of time necessary to produce “I am alive” or heartbeat messages by the group communication or consensus mechanism; and,
4. ***CTI***$_i$(j) denotes the connectivity time between processes Pi and Pj. The value “∞” is assigned to ***CTI***$_i$(j) when P_j becomes disconnected from P_i.

5.2 Failure Detector for Group Membership

What is required for failure detection in group membership is the use of timeouts values which reflect the load variations so that the occurrence of virtual partitioning of a process group can be minimised. The implementation of the Failure Detector for Group Membership based on timeout

[4] Notice that a failure detector which suspects all processes already satisfy the condition

values delivered by the CTI (therefore, dynamically adapted to loads variations), is shown in figure 1. Observe that the Failure Detector will only suspect a failure (i.e., FD will return the value true) if the remote process has been inactive for longer then the time necessary for producing an “I am alive” or heartbeat message (λ) plus the transmission time for that message indicated by $CTI_i(j)$ (line 3), or if the processes are considered as disconnected by the CTI module (line 2)[5].

5.3 Failure Detector <>S of Chandra-Toueg

The implementation of the Failure Detector <>S is given in figure 2. Besides minimising false suspicions (achieved by using the CTI) which could delay the convergence of a final value on the rotating coordinator consensus protocol, the Failure Detector presented also satisfies the properties required by <>S : *strong completeness* and eventual *weak accuracy* (see section 3). *Strong completeness* is trivially achieved since processes which actually crash will eventually have its time-silence set to a value larger than $CTI_i(j) + \lambda$ + recoveryTime (line 5). For the *eventual weak accuracy*, it must be guaranteed that after sometime at least a process will be fault-free and it will not be suspected[6]. For implementing such a property, first of all, it is introduced the concept of recoverable processes denoting special processes which have the ability of crashing and recovering without loosing any state or threat of execution. That is, recoverable processes can be relied on to eventually be operational long enough for deciding a decision value (when it becomes a round coordinator). Let the set RecoverableSet be the set of all recoverable processes. It is assumed that the content of RecoverableSet is supplied by the application related upper-layer software.

In order to guarantee that a recoverable process will not be (falsely) suspected, the timeout value, recoveryTime, is increased by a constant k, $k > 0$, every time consecutive suspicions occur for that process (line 2). Observe that, for avoiding the situation where crash detection will be indefinitely delayed for all processes, the parameter k is applied only for the recoverable ones[7] (line 1).

```
Funcion FD_i(j)
(1) When
(2)   CTI_i(j) = "∞"  or
(3)    time-silence_i(j) > CTI_i(j) + λ
(4)  holds return true;
```

Figure 1

```
Function FD_i,(j)
(1)  If justSuspected(i,j) and P_j ∈ recoverableSet
(2)          Then recoveryTime := recoveryTime +  k
(3)          Else recoveryTime := 0;
(4)  When  CTI_i(j) = "∞" or
(5)             time-silence_i(j) >  CTI_i(j) + λ + recoveryTime
(6)   holds  return true;
```

Figure 2

6 Concluding Remarks

Being able to detect failures is a fundamental issue in designing fault-tolerant distributed systems. However, the actual behaviour of a distributed system limits the ability of providing such a mechanism. Whereas synchronous systems allow for the construction of perfect failure detection based simply on local timeouts – which is required by Byzantine Agreement, Processor Membership, and System Diagnosis -, accurate failure detection cannot be developed for fully-asynchronous systems, unless some extra properties can be guaranteed – as it is the case for the failure detectors of Chandra e Toueg [1,2]. In all cases, the availability of a failure detection mechanism (being it reliable or not) is of crucial importance if one wants to built a fault-tolerant distributed system (i.e. a system which can provide continued service despite partial failures).

This paper first discussed the failure detection requirements for two widely used fault-tolerant mechanisms meant to asynchronous systems : process group membership and distributed consensus. It

[5] notice that the expression $CTI_i(j)$ = “∞” will hold true immediately after CTI has detected a crash or disconnection (e.g., if the P_j no longer belongs to the operating system lists or tables).

[6] In fact, it is only required that the process eventually remains fault-free and it is not suspected for the time duration necessary to decide a value.

[7] actually, *eventual weak accuracy* requires only one recoverable process.

then introduced a mechanism, called the Connectivity Time Indicator (CTI), which takes into account load variations in both communication channels and CPUs and dynamically delivers time connectivity information about pair of processes. Finally, CTI based implementations for failure detectors for group membership and <>S based distributed consensus have been presented, as well as a brief discussion on their correctness.

References

[1] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 43(2):225-267, March 1996.
[2] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. Journal of the ACM, 43(4):685--722, July 1996.
[3] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, 32(2):374--382, April 1985.
[4] Amir, Y., Dolev, D., Kramer, S., Malki, D. Transis: A Communication Subsystem for High Availability. In Proc. of the 22nd Int. Symp. on Fault-Tolerant Comp. pp. 76-84, Boston, July, 1992.
[5] K. Birman. The Process Group Approach to Reliable Distributed Computing. Communications of the ACM, Vol. 9, No. 12. pp. 36-53, December 1993.
[6] P. Ezhilchelvan, R. Macêdo, S. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In Proc. of the IEEE 15th Int. Conf. on Dist. Comp. Syst. Vancouver, pp. 296-306, 1995.
[7] M. Kashoek, A. Tanenbaum. Group Communication in the Amoeba Dist. Op. System. In Proc. of the Int. Workshop on Parallel and Distributed Systems, Vol.5, No.5, pp. 459-473, May, 1994.
[8] M. Shivakant, L. Peterson, R. Schlichting. A Membership Protocol based on Partial Order. In Proc. of the IEEE Int. Working Conf. on Dep. Comp. for Critical Applications, pp 137-145, February, 1991.
[9] M. P. Melliar-Smith, L. E. Moser, V. Agarwala. Processor Membership in Asynchronous Distributed Systems. IEEE Trans. on Parallel and Distributed Systems, 5(5):459-473, May1994.
[10] R. Renesse, K. Birman, R. Cooper, B. Glade, P. Stephenson. The Horus System. In K. Birman e R. Renesse, editores, Reliable Distributed Computing with the Isis Toolkit, pp. 133-147. IEEE Computer Society Press, Los Alamitos, CA, 1993.
[11] Hurfin, M., Macêdo, R., Raynal, M., Tronel, F. A General Framework to Solve Agreement Problems. Proc. of the IEEE Int. Symp. on Reliable Distributed Systems, SRDS'99, Lausanne. 1999.
[12] Badache, N., Hurfin, M., Macêdo, R. Solving The Consensus Problem In A Mobile Environment. Proc. of the IEEE International Performance, Computing, and Communications Conference – IPCCC'99, Phoenix/Scottsdale, USA: IEEE Press, 1999. p.29-35.
[13] Greve, F., Hurfin, M., Macêdo, R., Raynal, M. Consensus Based on Strong Failure Detectors : A Time and Message Efficient Protocol. Lecture Notes in Computer Science, v.1800, p.1258-1267, May/2000.
[14] Rampath, S., Dahbura, A. A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies. IEEE Trans. on Computers, vol. 44, No 4, Feb 1995.
[15] Duarte Jr, L., Nanya, T. A Hierarchical Adaptative Dist. System-level DiagnosisAlgorithm. IEEE Trans. on Computers, vol. 47, No 1, Jan/1998
[16] Lamport, L., Shostak, R., Pease, M. The Byzantine Generals Problem. ACM Trans Program. Lang. Syst. 4, 3 (July/1982), pp. 382-401.
[17] Cristian, F. Reaching Agreement on Processor-group Membership in Synchronous Distributed Systems. Distributed Comp. 4, 175-187, 1991.
[18] Schiper, A., Early Consensus in an Asynchronous System with a Weak Failure Detector. Distributed Computing, 10:149-157. 1997.
[19] Batalha, M., Macêdo, R. Arquitetura Orientada a Objetos para um Serviço Distribuído de Diagnóstico de Falhas sobre CORBA. Technical Report RT002/2000, Laboratório de Sistemas Distribuídos – LaSiD, UFBA, May/2000.
[20] Macêdo, Raimundo. "Implementing Failure Detection through the use of a self-tuned Time Connectivity Indicator". Relatório Técnico RT008/98, Laboratorio de Sistemas Distribuidos, UFBA, Agosto/98.