Université de Paris-Sud

Institut National des Télécommunications

Rapport de Stage
Master de Recherche en Informatique

# PARTITIONS IN WIRELESS GROUP COMMUNICATION SYSTEM

Muhammad Usman BHATTI
**Responsable du Master Recherche** : Olivier TEMAM
**Responsable de stage** : Denis CONAN

Septembre 2004

Ce stage de DEA a été réalisé au sein du laboratoire GET/INT CNRS Samovar, Équipe MARGE du département **Informatique** de l'**Institut National des Télécommunications**

# Contents

**1 Introduction** **1**

**2 Disconnection, Failure Detectors and Group Communication Systems** **3**

2.1 Disconnection and Failure Detection . . . 4

2.1.1 Failure Detection . . . 4

2.1.2 Connectivity and Disconnection Detection . . . 6

2.1.3 Failure, Disconnection and Consensus . . . 9

2.2 Group Communication System . . . 9

2.2.1 GCS Safety Properties . . . 9

2.2.2 GCS Liveness Properties . . . 12

2.3 Conclusion . . . 13

**3 Disconnection, Failure and Partition** **15**

3.1 Partition and Failure . . . 15

3.2 Partition and Disconnection . . . 17

3.3 Partition Detection . . . 18

3.3.1 Model . . . 18

3.3.2 Disconnection Detection with Partitions . . . 19

3.3.3 Heartbeat Failure Detector with Partitions . . . 22

3.3.4 Partition Detection with Neighborhood Topology . . . 23

3.3.5 Partition Detection with Global Topology . . . 25

3.3.6 Topology and Application-based Decisions . . . 28

3.4 Wireless Group Membership Service . . . 31

3.4.1 Membership Properties . . . 31

3.4.2 Group Membership Algorithm . . . 32

3.4.3 Group Agreement Algorithm . . . 33

3.5 Conclusion . . . 34

**4 Conclusions and Perspectives** **37**

**A Disconnection, Failure and Partition** **i**
A.1 Disconnection Detector $\mathcal{DDP}$ . . . . . . . . . . . . . . . . . . . . . . . i
A.2 Heartbeat Failure Detector with Partitions $\mathcal{HBDP}$ . . . . . . . . . . . . ii
A.3 Partition Detection Using Neighborhood Topology $\mathcal{PDN}$ . . . . . . . . iv
A.4 Partition Detection with Global Topology $\mathcal{PDG}$ . . . . . . . . . . . . . viii
A.5 Reconnections without Preserving Neighborhood $\mathcal{PDN}*$ . . . . . . . . ix
A.5.1 Modified Properties for $\mathcal{DDP}$ . . . . . . . . . . . . . . . . . . . . x
A.5.2 Proof for $\mathcal{DDP}$ . . . . . . . . . . . . . . . . . . . . . . . . x
A.5.3 $\mathcal{PD} - \mathcal{I}*$ . . . . . . . . . . . . . . . . . . . . . . . . . . . x
A.6 Optimizing Failure Detection . . . . . . . . . . . . . . . . . . . . . . . xiii

# List of Figures

2.1 Heartbeat Failure Detector for Partitionable Networks $\mathcal{HB}$ . . . . . . . 5
2.2 Hierarchical Failure Detection . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 The hysteresis of logical connectivity management . . . . . . . . . . . 7
2.4 Disconnection Detector$\mathcal{DD}$ . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5 Architecture of a Group Communication System . . . . . . . . . . . . 10
2.6 Merging views: $V_1$,$V_2$,$V_3$ and $V_4$ are the views. . . . . . . . . . . . . . . . 12

3.1 Failure and Partition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Disconnection and Partition . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Disconnection, Failure and Partition Detector . . . . . . . . . . . . . . . 18
3.4 Disconnection Detector for Partitionable Networks $\mathcal{DDP}$ . . . . . . . . 21
3.5 Heartbeat Failure Detector for Partitionable Networks $\mathcal{HBDP}$ . . . . . 23
3.6 Reachability Pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.7 Partition Detector with Neighborhood Topology $\mathcal{PDN}$ . . . . . . . . . 26
3.8 Partition Detector with Neighborhood Topology $\mathcal{PDN}$ (Ctn'd) . . . . . 27
3.9 Partition Detector with Global Topology $\mathcal{PDG}$ . . . . . . . . . . . . . . 29
3.10 Partition Detector with Global Topology $\mathcal{PDG}$ (Ctn'd) . . . . . . . . . 30

A.1 Partition Detector with Neighborhood Topology $\mathcal{PDN}$* . . . . . . . . . xi
A.2 Partition Detector with Neighborhood Topology $\mathcal{PDN}$* (Ctn'd) . . . . xii

# Chapter 1

# Introduction

Recent advancements in wireless data networking and portable information appliances have given the concept of mobile computing. Users can access information and services irrespective of their movement and physical location. Wireless communication, data processing and information services are becoming more and more important. With the increasing use of mobile terminals and mobile applications, mobility is an extra-functional feature which has become very significant. Mobility, being a vital aspect of today's mobile applications, must be built-in, not added as an afterthought, to all communication infrastructures and computing devices.

Mobile terminals are exposed to vast environments. The mobility of a mobile terminal gives rise to frequent disconnections, which are undesirable. These disconnections can be of two types: voluntary disconnections and involuntary disconnections; the former ones are decided by the users and the latter ones, a result of absence of wireless network signals. Disconnections can be very frequent and mobile terminals should continue working even while they are disconnected from the network. Hence, a mechanism is needed for measuring signal strength in order to anticipate for the forthcoming disruption in the network connectivity. We can call it connectivity detector. In addition, disconnection detector is necessary for the voluntary disconnection and involuntary disconnections in order to send an "alert" message to the other nodes declaring its disconnection. [Temal and Conan, 2004] presents the idea of disconnection and failure management in mobile applications.

Fault-tolerance is essential for distributed applications. Fault-tolerance comes in two phases: fault detection and fault correction [Gärtner, 1999]. Fault detection helps in maintaining application's safety and fault correction aides in maintaining application's liveness. In pure asynchronous distributed systems, consensus is insolvable in the presence of even one faulty process [Fischer et al., 1985]. This problem arises from the fact that we cannot differentiate amongst the faulty processes and the processes which are too slow. Nevertheless, unreliable failure detectors have been proposed, which help us solve the problem of consensus [Chandra and Toueg, 1996]. Consensus allows processes to reach a common decision, which depends on their initial inputs, despite failures. Thus, consensus assists in both, fault detection as well as fault correction.

Group communication systems (GCSs) are widely recognized as powerful building blocks for supporting consistency and fault-tolerance in distributed applications. The basic idea supported by GCSs is the notion of *multicast group*. Multicast groups are created on the fly by a *group membership service*. Traditionally, GCSs were generally employed in replicated objects and database consistency applications. Distributed systems, such as GCSs, are built-up by connecting various machines or components together through communication networks. They are prone to process crashes as well as link failures. Failures may cause a component or several sets of components to detach from the system, thus making a separate group disparting from the main network, making partition of the system. Partitions may result in service reduction or degradation but need not necessarily render the application completely unavailable. Partitions are a fact of life in most distributed systems and they tend to become more frequent as the geographic extent of the system grows or its connectivity weakens due to the presence of mobile units and wireless links. Thus, the partitions should perform as autonomous distributed systems providing services to their clients. The notion of partitionable GCS is an example where all the partitions are allowed to proceed in their computations [Montresor et al., 1999].

In this work, we are trying to establish a wireless group communication system which can support collaborative work or data sharing among mobile hosts. Chapter 2 describes the existing work which we are going to reuse for our purposes. We describe the notion of failure and disconnection detectors as well as group communication system. In Chapter 3, we develop algorithms to differentiate among disconnection, failure and partition and we specify the wireless group communication system. Finally, Chapter 4 concludes the report and gives perspectives. Proofs of the algorithms can be found in appendices.

# Chapter 2

# Disconnection, Failure Detectors and Group Communication Systems

We consider asynchronous distributed systems. They are distributed systems without bounds on message delay, clock drift or time necessary to execute a step. The system consists of processes and the processes communicate by message passing. As we have no bound on message delay, we cannot distinguish if a message is only taking too long to reach its destination or it is a failure [Fischer et al., 1985]. To circumvent this impossibility result, [Chandra and Toueg, 1996] proposed failure detectors, which monitor a subset of processes for failures and can make mistakes, thus the name *unreliable failure detectors*.

Apart from failures, mobile computing presents another challenge for wireless distributed system developers, which arises from the mobility of terminals. A mobile process, part of a distributed system, may not be slow or faulty but it may not find itself connected to the network because it has moved out of the communication range. We call them disconnections and they should be considered in the development of fault-tolerant distributed applications.

Group Communication Systems (GCSs) support consistency and fault-tolerance in distributed applications. Group communication systems provide multi-point to multi-point communication by organizing processes in groups [Vitenberg et al., 1999]. GCSs are powerful building blocks that facilitate the development of fault-tolerant distributed systems using the variant of the state-machine approach.

In this chapter, existing works relating to failure and disconnection detectors, and to group communication systems are presented. We precisely present disconnection and failure and how we can classify the two in Section 2.1. Then, we detail the group communication systems, types and properties in Section 2.2.

## 2.1 Disconnection and Failure Detection

In Section 2.1.1, we present failure detectors which are used for failure detection in asynchronous distributed settings. Mobility induced interruptions, called disconnections, are defined in the Section 2.1.2.

### 2.1.1 Failure Detection

Each process has an access to a failure detector module. Each module monitors a subset of processes in the system, and maintains a list of those that it currently suspects to have crashed. Failure detectors can be erroneous in their suspicions: they can suspect that process $p$ has crashed while it is still running. Later on, they will remove $p$ from the list of suspects if suspicion was erroneous. Mistakes made by failure detectors should not stop the correct processes behaving according to the specifications. Failure detectors introduce the concept of partial synchrony where the time to deliver a message or to execute a process step is bounded but these bounds are unknown. By introduction of such partial synchrony assumptions, it is possible to obtain practical solutions for various problems. There are various other techniques for evading the impossibility result like probabilistic solutions and initially dead processes but they are out of the scope of this work.

Failure detectors have two properties in terms of their functionality. Completeness states that there is a correct process which suspects every faulty process. Completeness is an important property as it satisfies the safety requirements for a failure detector. Completeness can be divided into two properties: *weak completeness* and *strong completeness*. Completeness in itself is not a useful property and has to be augmented with an accuracy property which restricts the mistakes that failure detector can make. Thus, accuracy states that no process is suspected before it crashes. Accuracy satisfies the liveness requirements of a system. Accuracy is divided into four properties, which are *strong accuracy*, *weak accuracy*, *eventual strong accuracy*, and *eventual weak accuracy*. Below we give the definitions of each of them.

1. **Completeness** :
   - *Strong Completeness* : There is an instant after which every faulty process is suspected by all the correct processes.
   - *Weak Completeness* : There is an instant after which every faulty process is suspected by at least one correct process.
2. **Accuracy** :
   - *Strong Accuracy* : No process is considered faulty before it crashes.

- *Weak Accuracy* : Some correct process is never suspected.
- *Eventual Strong Accuracy* : There is a time after which correct processes are not suspected by any correct process.
- *Eventual Weak Accuracy* : There is a time after which some correct process is never suspected by any correct process.

There are various forms of failure detectors that we find in the literature today. Some perform failure detection by sending a ping message and waiting for the response [Chandra and Toueg, 1996]. If they do not receive a response in a given time (timeout), they declare a process to be faulty. This may be the simplest form of a failure detector as they do not introduce various optimizations like exploiting network topology. Then, there are those which send "heartbeat" messages to their neighbors and count the heartbeats of their neighbors [Aguilera et al., 1997] (cf. Figure 2.1). This genre of failure detector only counts the heartbeats it receives from the neighboring processes and passes this information to the upper layers for deciding which processes are faulty. The version listed here is for partitionable networks. There are still those which randomly select a process and send heartbeat to that process. The receiving process merges the list of processes in the incoming processes with that of its own. Their authors call it gossip-style failure detection [Renesse et al., 1998].

```
for every process $p$ :
    initialization :
        for all $q \in \Pi$
            $D_p[q] \leftarrow 0$                          {$D_p$ is the output of $\mathcal{HB}$ at $p$}
    cobegin :
        || task 1 : repeat periodically
            $D_p[p] \leftarrow D_p[p] + 1$                 {Increment $p$'s own heartbeat}
            for all $q \in \Pi$ such that $q \in neighbor(p)$
                send$(\mathsf{HEARTBEAT}, p)$ to $q$
        || task 2 : upon receive$(\mathsf{HEARTBEAT}, path)$ from $q$
            for all $q \in \Pi$ such that $q$ appears after $p$ in $path$
                $D_p[q] \leftarrow D_p[q] + 1$
            $path \leftarrow path.p$
            for all $q \in \Pi$ : $q \in neighbors(p) \wedge q$ appears at most once in $path$
                send$(\mathsf{HEARTBEAT}, path)$ to $q$
    coend
```

Figure 2.1: Heartbeat Failure Detector for Partitionable Networks $\mathcal{HB}$

Failure detection algorithms are mainly defined for the LAN environments. That's why most of the failure detection services scale badly as the number of members to be monitored increases. With the advent of world-wide distributed systems, it is becoming clear that the failure detection systems that are being used today in their local settings (LAN), cannot simply be employed in their existing form for wide-area, large-scale operation. Hence, large-scale failure detection needs more attention than just trivially converting local-area failure detection to large-scale one. Hierarchical failure detection is an example of how to adapt failure detectors for large-scale environments [Bertier et al., 2003]. As shown in Figure 2.2, the system is composed of local groups, mapped upon a LAN, bounded together by a global group space, called WAN groups, where each group, either local or global, is a detection space.

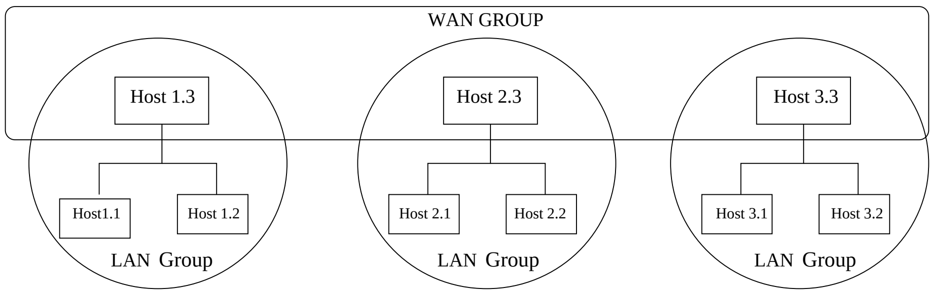

Figure 2.2: Hierarchical Failure Detection

[Burns et al., 1999] have given another concept of large-scale failure detection using network topology, instead of randomly choosing members for gossiping. They try to exploit Internet domains and sub-domains architecture for large-scale failure detection. Failure detectors for large-scale distributed and grid systems have been defined in [Hayashibara et al., 2002]. A grid system may change its configuration during its execution and the failure detection service should be aware of configuration changes, and should be able to alter itself according to these.

### 2.1.2 Connectivity and Disconnection Detection

With the advent of mobility in distributed applications, we have a new kind of problem that may appear: the problem of disconnection while a distributed application is running on the mobile terminal and the mobile terminal moves out of communication range. We cannot classify it as a link or process failure because it has different properties. For example, we can prevue the disconnection before it happens by monitoring the signal strength. This requires to invent a new disconnection detector with a new set of properties. [Temal and Conan, 2004] has already defined a disconnection detector analogous to unreliable failure detector and its properties. In this section, we find out the details of a connectivity detector based on the principles of signal strength and hysteresis mechanism. Then, we examine how to disseminate this information to other nodes about the forthcoming disconnection, therefore, called a disconnection detector.

Connectivity detectors are based on the idea of physical connectivity managers and logical connectivity managers, first presented in [Conan et al., 2002]. The idea is to monitor the network resources to foresee the network disconnection. Physical connectivity manager monitors different types of networks available to the mobile terminal in order to enable the seamless switching. When a network disconnection occurs on the link, which is used by the application, physical connectivity manager detects the disconnection event and notifies either the application or some other service. Physical connectivity manager keeps monitoring the network activity and periodically attempts to reconnect. In order to insulate the application from the insignificant variations in resource level, the logical connectivity managers rely on hysteresis mechanism for smoothing variations in resource availability (see Figure 2.3). In Figure 2.3-1, one can remark that as long as the resource level remains lower than lowUp (*resp.* highUp ), the mobile terminal remains disconnected (*resp.* partially connected). When the resource level decreases but remains higher than highDown (*resp.* lowDown ), the mobile terminal remains connected (*resp.* partially connected). Note that without Figure 2.3-2, there is a risk of ping-pong around the values highDown and lowUp . So, when the resource level decreases to the state E from the state F (*resp.* decreases to the state B from the state C) and the resource level again increases to highDown (*resp.* decreases to lowUp ), the mobile terminal remains in the partially connected mode until it reaches the value highUp (*resp.* lowDown ). In the later work [Temal and Conan, 2004], connectivity detector algorithm and proofs were presented based on the hysteresis mechanism.

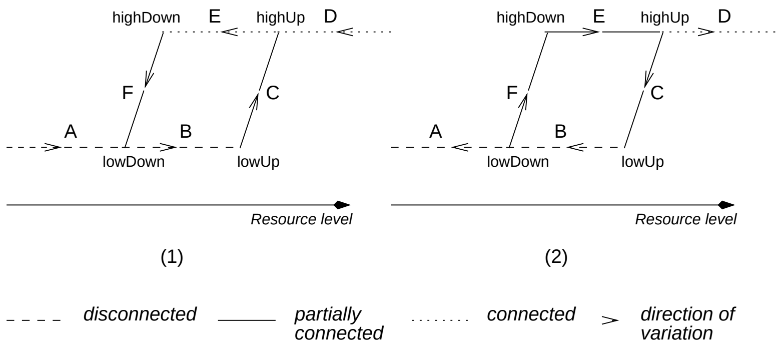

Figure 2.3: The hysteresis of logical connectivity management

The connectivity information is local to each node. Thus, there is a need for exchanging connectivity information such that a process could declare its disconnection or reconnection to the other processes. For this reason, disconnection detectors (cf. Figure 2.4) were introduced which could transmit this local connectivity information to all the connected processes [Temal and Conan, 2004]. Disconnection detector, like failure detector, is described in abstract terms to generalize the model and not to give any implementation-specific details. For this purpose, abstract properties like *disconnection completeness* and *disconnection accuracy* have been defined**:**

- *Strong disconnection completeness*: There is a time after which all the disconnected processes are seen as disconnected by all the connected processes.

- *Weak disconnection completeness*: There is a time after which all the disconnected processes are seen as disconnected by at least one connected process.

- *Strong disconnection accuracy*: No process is seen as disconnected until it disconnects.

```
for every process $p$ :
   initialization :
      $disc_p \leftarrow \emptyset$                {set of processes seen disconnected}
      for all $q \in \Pi$
         $N_p[q] \leftarrow 0$                     {vector of received disc./rec. numbers}
   cobegin :
      || task 1 : upon change mode notification
         if $getMode() =$ ‘d’
            for all $q \in \Pi \setminus disc_p$
               send($\mathsf{DISCONNECT}, N_p[p]$) to $q$
            $N_p[p] \leftarrow N_p[p] + 1$
            $disc_p \leftarrow \emptyset$
         else if $getMode() =$ ‘c’
            for all $q \in \Pi$
               send($\mathsf{RECONNECT}, N_p[p]$) to $q$
            $N_p[p] \leftarrow N_p[p] + 1$
      || task 2 : upon receive($\mathsf{DISCONNECT}, n_q$) from $q$
         if $q \notin disc_p \wedge N_p[q] < n_q$ then $disc_p \leftarrow disc_p \cup \{q\}$
         $N_p[q] \leftarrow max(N_p[q] + 1,\, n_q)$
      || task 3 : upon receive($\mathsf{RECONNECT}, n_q$) from $q$
         if $q \in disc_p \wedge N_p[q] < n_q$ then $disc_p \leftarrow disc_p \setminus \{q\}$
         $N_p[q] \leftarrow max(N_p[q] + 1,\, n_q)$
   coend
```

Figure 2.4: Disconnection Detector$\mathcal{DD}$

### 2.1.3 Failure, Disconnection and Consensus

Three approaches have been suggested in [Temal and Conan, 2004] for unifying the failure detectors and disconnection detectors. The first one proposes the use of connectivity detection for failure detection. It is reasoned that the heartbeats can be configured according to the connectivity. If the connectivity is good, it can send heartbeats to other processes more frequently and vice versa. The processes can also, according to their logical connectivity, negotiate the number of heartbeats they exchange. The second approach neglects the disconnected processes in consensus algorithm thus no message is expected from them. Also, the processes, which disconnect while running the consensus algorithm, never wait for a decision or termination. In the third approach, disconnection detectors and failure detectors are used together in order to optimize the heartbeat algorithm so that heartbeats are sent only to the processes which are connected. Mobile terminals have low battery capacity and network resources. Thus, these optimizations can help saving battery power.

## 2.2 Group Communication System

The first known GCS was developed as a part of the Isis toolkit [Birman, 1986]. Group communication systems enable processes, located at different nodes of a distributed network, to operate collectively as a group, which is facilitated by a *membership service* and the communication service that delivers the messages to the group members is known as *multicast service*. An architecture for typical GCS is presented in Figure 2.5-1, where membership and multicast services are built on top of an unreliable network.

Section 2.2.1 describes the safety properties of the membership and the multicast service for the partitionable group communication systems. Liveness properties for membership and multicast services are presented together in Section 2.2.2.

### 2.2.1 GCS Safety Properties

GCSs are prone to process crash and link failures. Thus, it would not be surprising if some machine or process becomes faulty due to one of them. There might be cases where only one component breaks off from the main group or there might be a group of components which detach from the main group. Different group membership services handle such situations in different ways. The main group continues its processing and the segregated groups block while waiting for restoration of links. This type of GCS is called *primary component* because only the primary group (with a majority of processes) proceeds. *Partitionable* membership service allows all the components to operate autonomously, that is, they do not block. Partitionable GCS allows multiple disjoint views of the same group to exist concurrently in different network components.

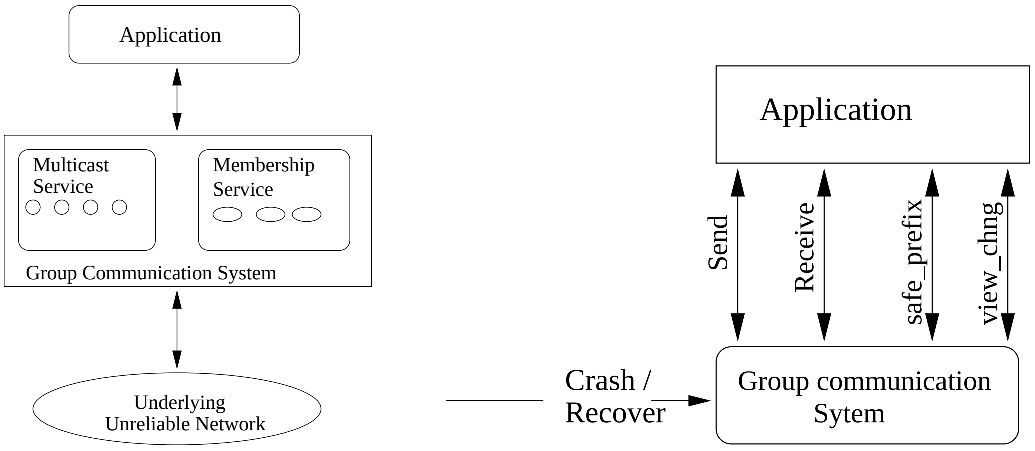

Figure 2.5: Architecture of a Group Communication System

Partitionable GCS was introduced as part of Transis [Amir et al., 1992]. Since then, they have found their uses in resource allocation, system management, highly available server, and collaborative computing applications.

Group communication systems organize processes in groups and each group is associated with a logical name. So, this is this logical name which is addressed while sending a message to all the members of a group. The task of a membership service is to maintain a list of currently active and connected processes in the group and a unique identifier. When this list changes (with new members joining and old ones departing or failing), the group membership service reports the change to the group members. The membership service strives to deliver the same view (consisting of the same member list and the same identifier) to mutually connected members. Events occur at processes within the context of views that is why they are widely known as view-oriented group communication systems. Partitionable membership service, defined in [Dolev et al., 1996, Babaoglu et al., 2001, Fekete et al., 2001], has to know which processes are currently part of the given partition. In particular, network partitions may split the group into several clusters that may later merge when partitions are repaired. There may well be many autonomous partitions existing in the system at the very same time. The safety properties are necessary as they keep the overall system in a consistent state. Below, we list safety properties for a partitionable membership service:

- Integrity: A local view on a process always includes itself. All the members in a view should be mutually reachable from each other.
- Same Order: View installations at overlapping members occur in the same order. In partitionable systems, it is unreasonable to require that all the correct processes install views according to some total order due to the possibility of concurrent

partitions. Yet, for a partitionable group membership service to be useful, the set of views must be consistently ordered by those processes that do install them. If two views are installed by a process in a given order, the same two views cannot be installed in the opposite order by some other process.

- View Coherency: If a process in a view $v'$ installs a new view $v$ then all other processes in that view $v'$ also install a new view $v$. It states that processes should not stick to a view and should install new views while the network configuration changes.

- Safety: Processes are removed only if they do not respond to the messages sent to them. The idea is to avoid removing any processes from the views which are not reachable from all the processes. This property prevents capricious view splitting [Anceaume et al., 1995], a phenomenon where processes install views excluding those which might be permanently reachable.

Multicast service provides its services in association with membership service to deliver message to the members of the current view. Figure 2.5-2 shows that there are various messages that are exchanged between a GCS and an application (cf. Figure 2.5): namely *send* and *receive* are the primitives to send and to receive messages; *Safe_prefix* message confirms the delivery of the received message that is received currently and all the messages that were received before current message; *View_chng* informs the application of a view change after the change in underlying configuration of processes. If the two consecutive views are delivered to several processes, then exactly the same multicast messages are delivered to these processes between these two views. This is called *virtual synchrony*. Virtual synchrony provides a convenient framework for the state machine replication approach: Since messages and views are delivered in the same order to all non-faulty replicas, consistency is preserved. Below we list the safety properties for a partitionable multicast service:

- Message Agreement: Two processes install the same view $v$ in the same view $v'$ then both of them receive the same set of messages in $v$.

- Same View Delivery: Two processes deliver the same messages in the same view. The two properties are necessary for view synchrony. View synchrony helps in consistency as there is no need for a state transfer between two processes after a new view installation if both of them were members of the same view earlier. This is especially useful for applications that implement data replication. But additional information is required by processes to assess which processes satisfy virtual synchrony, hence the next property helps.

- Merging Views: Two merging views must have disjoint composition. Consider the scenario depicted in Figure 2.6 where three processes $p$, $q$,and $r$ have all installed view $V_1$. At some point, process $r$ crashes and process $p$ becomes temporarily unreachable from process $q$. Process $p$ reacts to both events by installing view $V_2$ containing only itself before merging back with $q$ and installing view $V_3$. Process $q$, on other hand, reacts only to the crash of $r$ and installs view $V_3$ excluding $r$. Suppose that $p$ and $q$ share the same state in view $V_1$ and that $p$ modifies its state during $V_2$. When $p$ and $q$ install $V_3$, $p$ knows immediately that their states may have diverged, while $q$ cannot infer this fact based on local information alone. Therefore, $q$ could behave inconsistently with respect to $p$. In an effort to avoid this situation, $p$ could collaborate by sending to $q$ a warning message as soon as it installs view $V_3$, but $q$ could perform inconsistent operations before receiving such a message. The problem stems from the fact that views $V_1$ and $V_2$ that merge to form view $V_3$ have at least one common member ($p$).

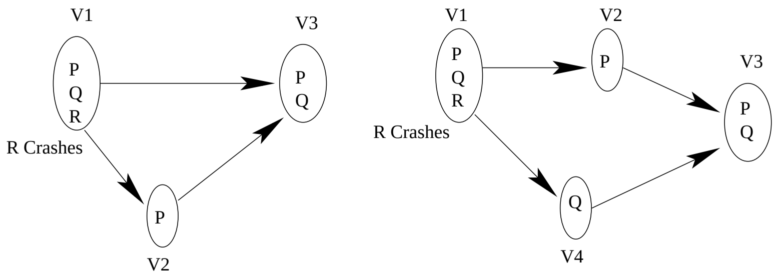

Figure 2.6: Merging views: $V_1$,$V_2$,$V_3$ and $V_4$ are the views.

- Delivery Integrity: For every *receive*, there is an associated *send* event and each process delivers a message at most once.
- Self Delivery: A correct process always delivers its own multicast messages.

### 2.2.2 GCS Liveness Properties

Liveness is an important complement to safety, since without requiring liveness, safety properties can be satisfied by trivial implementation that do nothing. Liveness of GCSs depends on the network conditions. We briefly list down the liveness properties for membership as well as multicast service. Liveness properties are seen as a whole for group communication systems because they, generally, depend on the underlying network conditions and it is not important to separately define the two:

- Membership Precision: $p$ installs a view $V$ as its last view.

- Multicast Liveness: Every message sent by a correct process is received by all the correct processes.
- Self Delivery: $p$ delivers every message it sent unless it crashed after sending it.
- Safe Indication Liveness: Every message sent in $V$ is indicated safe by all other correct processes.
- Membership Accuracy: If the system stabilizes the group membership is consistent.
- Termination of Delivery: Every message sent is received or the sender installs a new view.

## 2.3 Conclusion

In this chapter, we have considered asynchronous distributed systems. They are systems without bounds on message delay, clock drift or time necessary to execute a step. The system consists of processes and the processes communicate by message passing. Impossibility result hinders the detection of failures in these systems. For this, unreliable failure detectors have been introduced, which render partial synchrony in pure asynchronous systems but this partial synchrony is necessary for fault-tolerance. Thus, bounds are placed on execution of process steps or message delivery. Each process has an access to a failure detector module. Each module monitors a subset of processes in the system, and maintains a list of those that it currently suspects to have crashed. Failure detectors were, primarily, introduced for LAN environments. With the increase in the geographical extent of distributed systems, failure detectors should also be adapted with the spread of these systems. For that reason, wide-area failure detection is getting more importance and a small survey of large-scale failure detection has been presented in this chapter.

Advances in the mobile technology and increase in application mobility expose the problem of disconnections, where an entity of an application may not find itself connected to the other entities of the application. Disconnection detectors can be designed by anticipating the disconnections while monitoring the signal strength of the network. In this chapter, we have seen the abstract properties of disconnection detectors. Connectivity detectors are local to each process. They continuously monitor the network resource level according to various threshold levels. When the resource level decreases below a threshold value, they declare the terminal to be disconnected. Information provided by connectivity detectors is local to each entity. Therefore, disconnection detectors are designed and used to spread this information to other processes. Disconnection detectors have abstract properties like completeness and accuracy.

Group communication systems (GCSs) are widely recognized as powerful building blocks for supporting consistency and fault-tolerance in distributed applications. GCS provide strong fault-tolerance semantics as the group of processes make an agreement over the correct processes and secondly, whenever processes exchange messages they send and receive them without the loss of messages and sometimes, in their total order. Group communication is a means for providing multi-point to multi-point communication by organizing processes in groups. Consequently, processes move in locksteps while exchanging the same messages in the same order which is a variant of state-machine approach. Thus, system developers can implement these systems according to their requirements. GCSs have a membership service, which keeps a list of correct processes and binds them into one view. In addition, Multicast service provides the services for message exchange among the processes within a view. Group communication systems provides two kinds of membership and multicast services. Primary component GCSs ensure that there is only one view in the system and the processes which are not part of that view are considered faulty, and henceforth, they are blocked. Partitionable GCSs consider the fact that detached processes from the main view might be able to provide services, may be in a degraded form. Partitionable GCSs support installation of concurrent views to exist on processes which have detached from the main network. They do not get blocked and do not stop providing their services. But, we need a modified set of properties for partitionable GCSs since they allow concurrent views to exist, as a result needing reconciliation when they merge.

# Chapter 3

# Disconnection, Failure and Partition

In Chapter 2, we have described in detail the failure and disconnection detectors. With the increase in mobile computing, the weak connectivity model of wireless networks is becoming more conspicuous, that is, there are more chances of disconnections due to the constraints like communication range and battery power. While we move from LAN environments to WAN and wireless environments, all-to-all connection has become more of a myth than a reality. Mobile network models range from cellular network, where only mobile hosts can roam around and the communication infra-structure remains static and operational, to ad hoc networks, where the networks might not have a basic infra-structure, therefore all nodes have to collectively make decisions.

In the absence of all-to-all communications, certain failures and disconnections might divide the network, called *partitions*. These are possible within wireless as well as wired networks. Within wired networks, there can be a router in the Internet which disrupts traffic between two regions. Within wireless networks, there can be a host that routes the traffic of other nodes, and upon its disconnection, the two set of processes make two partitions. For this reason, partition detection is an important aspect of todays distributed systems. In this chapter, we try to differentiate disconnections, failures, and partitions. In Section 3.1, we try to distinguish between failures and partitions. The same exercise is repeated for disconnections and partitions in Section 3.2. Section 3.3 details the algorithms which distinguish among the three: disconnection, failure, and partition. In Section 3.4, we present wireless group membership properties and algorithm.

## 3.1 Partition and Failure

We consider only crash failures in our work where a process halts prematurely and never recovers. A failure occurs when a process crashes due to an internal failure and does not make any progress. A partition on the other hand, is a problem of the network or a connecting node, while the processes do not crash. In the following paragraphs, we explore these issues.

The general model of failure detection works as follows: the sender sends a "ping" message and the receiver replies with an "ACK" message. Thus, both sender and receiver know that both of them are "alive". When there is an absence of a reply for a certain amount of time, called "timeout", the sender declares the receiver as faulty, or more generally "problematic". Figure 3.1-a shows that the message $m$ is lost due to a link failure even if both processes, $p$ and $q$, are alive. Figure 3.1 depicts this situation where a process $q$ sends a message $m$ to another process $p$. Figure 3.1-b shows that the receiver crashes and does not reply to the message $m$. In both situations, the process on the other side of the network is declared faulty, while that process is not responding or is not receiving the message due to link problems. This deficiency comes from the inherent mechanism of failure detection using the "ping" message. Thus, we have an impossibility result here that we cannot distinguish between link failure and process failure, which we do not prove but only reason for its existence. This impossibility result comes from the fact that we cannot distinguish between link failure and process failure by the existing failure detection techniques, defined in Section 2.1.1.

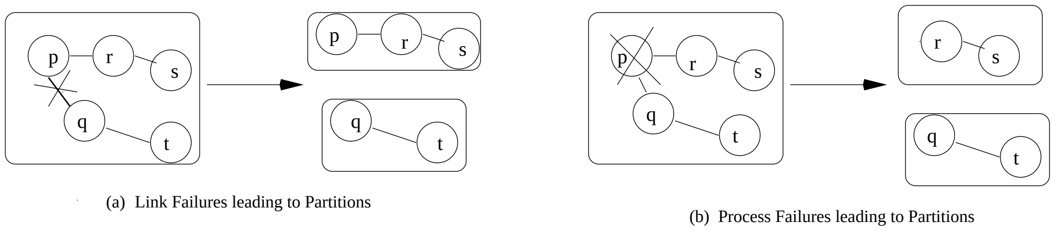
p
r
s
q
t
(a) Link Failures leading to Partitions
(b) Process Failures leading to Partitions

Figure 3.1: Failure and Partition

A partition occurs when two processes detach themselves such that every process within the partition considers the other processes within the partition to be alive. In the literature, processes outside a partition are considered to be faulty. Therefore, more information needs to be collected for the distinction of failure and partition. One more thing to note is the fact that for the reason of partitions detection we can call process failure as equivalent to link failures. This is because a process failure may eventually cause a link failure between two set of processes.

Two processes are called *reachable* if they can communicate with each other directly, or through some other process, which routes the messages to correct processes. A number of events can cause the reachability of the processes to be changed into *unreachability*, namely link crashes, buffer overflows, incorrect and inconsistent routing tables. A process crash, may as well, render the two processes as unreachable. Reachability is formally defined in Section 3.3.4.

## 3.2 Partition and Disconnection

While failure detectors make it impossible to distinguish failures from partitions, disconnection detectors does not have this shortcoming because when a process disconnects, it sends a disconnection message to all the processes it is connected to. In this section, we mainly discuss the peculiarities associated with disconnection detector. In the following paragraphs, we elaborate the disconnection detector in terms of partition detection and differentiating the two. Afterwards, we explain the reachability issues for a disconnection detector.

As defined earlier, processes send an "alert" message before they disconnect. This way processes can know of a disconnection leading to a partition. But, still we need additional information to know if the disconnecting process is creating a situation depicted in Figure 3.2. In the figure, process $p$ disconnects by sending message 'd' to detach two sets of processes. One thing to note here is that with the disconnection message, process $q$ only knows about the disconnection of process $p$. Had it known it was connected to process $r$ through $p$, then it would have declared that process $p$ disconnects to form two partitions of the network. Hence, we need some network topology information, that we develop in Section 3.3 for detecting any disconnection leading to partitions. One more thing to consider is that processes might be disconnected and reconnect afterwards restoring the original topology as in Figure 3.2.

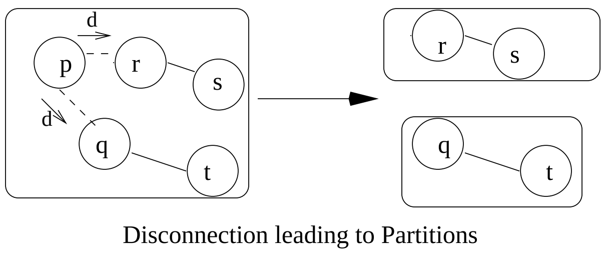

Figure 3.2: Disconnection and Partition

We reuse the very same notion of reachability as introduced in Section 3.1, that is, two processes are reachable if they can communicate with each other directly, or using a third process as a router. Disconnections may change reachability as well. Looking at Figure 3.2, we can infer that reachability of processes $p$ and $q$ changes completely thus changing the reachability of other processes, making $q$ unreachable to processes $r$ and $s$. Reachability is formally defined in Section 3.3.

## 3.3 Partition Detection

The architecture for partition detection and membership management is presented in Figure 3.3. Partition detection is performed using the information collected from the disconnection detector and the failure detector. This information is used by the membership service for view formation. In the two preceding sections, we have seen that we cannot distinguish amongst disconnection, failure and partition using the existing works. Therefore, additional information has to be collected. We develop two algorithms for partition detection: The neighborhood topology and the global topology. In this section, we detail these two algorithms that we develop in the framework of this work. First, we present our model in Section 3.3.1. In Section 3.3.3, the heartbeat algorithm for partitionable networks is listed. In Section 3.3.2, we present the modified version of disconnection detectors adapted for partitionable networks. In Section 3.3.4, we present the partition detector based on neighborhood information. In section 3.3.5, we present the partition detector based on global topology information.

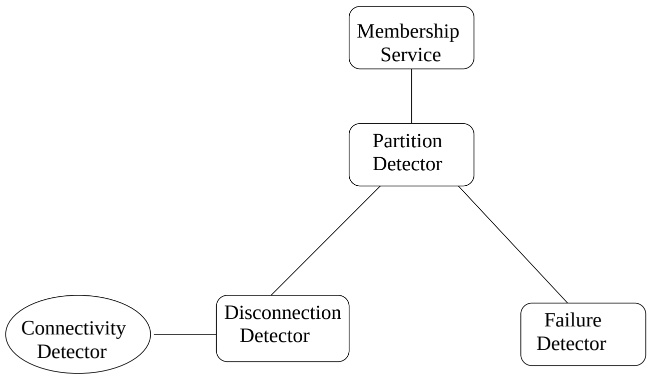

Figure 3.3: Disconnection, Failure and Partition Detector

### 3.3.1 Model

A network is a directed graph $G = (\Pi, \Lambda)$ where $\Lambda \subset \Pi \times \Pi$. The system consists of a set of $n$ processes $\Pi = \{p_1, p_2 ..., p_n\}$. Every pair of processes is connected by fair communication path. We take the range $\mathcal{T}$ of the clock's tick to be the set of natural numbers. Processes do not have access to $\mathcal{T}$: it is introduced for the convenience of presentation.

Processes can fail by crashing, that is, by prematurely halting. A *process failure pattern* $F_p$ is a function from $\mathcal{T}$ to $2^{\Pi}$, where $F_p(t)$ denotes the set of processes that have crashed through time $t$. Once a process crashes, it does not "recover", that is, $\forall t \in \mathcal{T} \; : \; F_p(t) \subseteq F_p(t+1)$. We define $crashed(F_p) = \bigcup_{t \in \mathcal{T}} F_p(t)$ and

$correct(F_P) = \Pi \setminus crashed(F_P)$, the set of correct processes. If $p \in crashed(F_P)$, we say that $p$ crashes in $F_P$ and if $p \in correct(F_P)$, we say that $p$ is correct in $F_P$.

A *link failure pattern* $F_L$ is a function from $\mathcal{T}$ to $2^{\Lambda}$. $F_L$ denotes the set of links that have crashed through time $t$. Crashed links do not recover, that is, $\forall t \in \mathcal{T} : F_L(t) \subseteq F_L(t+1)$. This is defined as $crashed(F_L) = \bigcup_{t \in \mathcal{T}} F_L(t)$. If $p \rightarrow q \in crashed(F_L)$, we say link between $p$ and $q$ crashes. If $p \rightarrow q \notin crashed(F_L)$, we say $p \rightarrow q$ is fair in $F_L$.

A *failure pattern* $F = (F_P, F_L)$ combines a process failure pattern and a link failure pattern.

Formally, a *disconnection detector history* $H_{DC}$ is a function from $\Pi \times \mathcal{T}$ to $2^{\Pi}$ where $H_{DC}(p, t)$ is the value of disconnection detector of process $p$ at time $t$ in $H_{DC}$. If $q \in H_{DC}(p, t)$ then we say that $p$ considers $q$ as disconnected at time $t$. Disconnection detector $\mathcal{DD}$ is a function which maps each disconnection pattern $DC$ to a set of disconnection history $\mathcal{DD}(DC)$. $DC$ is a function from $\mathcal{T}$ to $2^{\Pi}$, where $DC(t)$ is the set of processes disconnected at time $t$. $disconnected(DC)$ represents the set of processes disconnected in $DC$, and $connected(DC) = \Pi \setminus disconnected(DC)$ represents the set of processes that are connected.

We consider a network with two types of links: links that are fair and links that crash. A link can intermittently drop messages, but do so infinitely and if $p$ repeatedly sends a message to $q$, then $q$ eventually receives that message. We consider *fair links*, that is if $p$ sends a message $m$ to $q$ an infinite number of times and $q$ is correct, then $q$ receives $m$ from $p$ an infinite number of times.

Process $q$ is reachable from process $p$ if there is a fair path from $p$ to $q$. If processes $p$ and $q$ are mutually reachable (noted $p \leftrightarrows q$) then they are considered to be in the same partition, that is, $q \in partition(p)$. The partition of process $p$ with respect to $F$ is denoted $partition(p)$. If $p$ is faulty or disconnected, we define $partition(p) = \emptyset$. As defined earlier, reachability can be affected by disconnections or failures. A *partition detector history* $H_{PD}$ is a function from $\Pi \times \mathcal{T}$ to $2^{\Pi}$ where $H_{PD}(p, t)$ is the value of partition detector of process $p$ at time $t$ in $H_{PD}$. If $q \in H_{PD}(p, t)$ then we say that $p$ considers $q$ as partitioned at time $t$.

## 3.3.2 Disconnection Detection with Partitions

In this section, the disconnection detector for partitionable networks $\mathcal{DDP}$ is presented. The algorithm is a modified form of the algorithm presented in Section 2.1.2. We define a new primitive called $qr\_neighbor\_send$. Using this primitive, a process can send a disconnection message with reliability to all its neighbors. A process $p$ tries to send a disconnection message repeatedly to all of its neighbors and waits for acknowledgment from the receiving process. Since the link is fair, sending a message infinitely means that the message will be received. In case $p$ receives the acknowledgment from the receiving process, it stops sending the message. But if it disconnects before having any acknowledgment from any neighboring process then it will have to send a disconnection message whenever it reconnects. This property is used in the proof of this algorithm. In the following paragraphs, the algorithm is explained and its properties are listed. Proofs are given in the appendices in Section A.1.

The disconnection detector for partitionable networks $\mathcal{DDP}$ is presented in Figure 3.4. The methods $getMode()$ and $getVoluntary()$ are getters of the variables $m$ (for the mode) and $voluntary$ (that states whether there is a voluntary disconnection) of the connectivity detector $\mathcal{CD}$ (cf. Section 2.1.2). The algorithm executes three parallel tasks:

1. The first task executes if there is a change in the connectivity mode, which is detected by the connectivity detector using the hysteresis algorithm (cf. Section 2.1.2). If the terminal goes into a disconnected mode, the process sends a DISCONNECT message to all of its neighbors. The disconnection message is enumerated and paired with $N_p[p]$ in order to avoid the duplication of messages. The counter $N_p[p]$ is incremented and the list of processes regarded as disconnected is emptied in order to avoid a conflict with the configuration when the process reconnects itself. If the terminal, earlier disconnected, reconnects itself, a RECONNECT message is sent to all the neighbors along with its counter $N_p[p]$. It then increments the counter.

2. The second (*resp.* third) task handles the reception of DISCONNECT (*resp.* RECONNECT) message sent by processes that disconnect (*resp.* reconnect). The receiving process verifies if it has received this message earlier. If it has not received this message earlier, it sends a DISCONNECT (*resp.* RECONNECT) message to all its neighbors using the primitive $qr\_neighbor\_send$. The primitive is used in order to make the message dissemination reliable over fair links that may drop messages. Then, it appends the sender to (*resp.* removes the sender from) the list of disconnected processes. We reuse the algorithm defined by [Aguilera et al., 1999], therefore it need not to be proved again.

The intended model and properties are the same as described in Section 2.1.2. We keep the same assumption that if a process disconnects then it reconnects to the same set of process (in Section A.1, we drop this assumption). The abstract properties defined for disconnection detector are:

- *Strong disconnection completeness*: There is a time after which all the disconnected processes are seen as disconnected by all the connected processes in the same partition. Formally,

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \exists t \in \mathcal{T},\ \forall p \in disconnected(DC),\ \forall q \in connected(DC), \forall q \in partition(p),\ \forall t' \geq t : p \in H_{DC}(q,\ t')$$

```
for every process $p$ :
    initialization :
        $disc_p \leftarrow \emptyset$                {set of processes seen disconnected}
        for all $q \in \Pi$
            $N_p[q] \leftarrow 0$                {vector of received disc./rec. numbers}
    cobegin :
        || task 1 : upon change mode notification with ($mode_p$, $voluntary_p$)
            if $mode_p =$ ‘d’ $\vee\ voluntary_p$
                for all $q \in neighbor(p)$
                    qr_neighbor_send($\mathsf{DISCONNECT}, p, N_p[p], p$) to $q$
                $N_p[p] \leftarrow N_p[p] + 1$
                $disc_p \leftarrow \emptyset$
            else if $mode_p =$ ‘c’ $\wedge \neg voluntary_p$
                for all $q \in neighbor(p)$
                    qr_neighbor_send($\mathsf{RECONNECT}, p, N_p[p], p$) to $q$
                $N_p[p] \leftarrow N_p[p] + 1$
        || task 2 : upon receive($\mathsf{DISCONNECT}, q, n_q, path$)
            if $q \notin disc_p \wedge N_p[q] < n_q$
                for all $r$ such that $r \in neighbor(p)$ and $r$ appears at most once in
                $path$
                    $send(\mathsf{DISCONNECT}, q, N_p[q], path)$
                $disc_p \leftarrow disc_p \cup \{q\}$
                notify disconnection of $q$
            $N_p[q] \leftarrow max(N_p[q] + 1,\ n_q)$
            $path \leftarrow path.p$
        || task 3 : upon receive($\mathsf{RECONNECT}, q, n_q, path$)
            if $q \in disc_p \wedge N_p[q] < n_q$
                for all $r$ such that $r \in neighbor(p)$ and $r$ appears at most once in
                $path$
                    $send(\mathsf{RECONNECT}, q, N_p[q], path)$
                $disc_p \leftarrow disc_p \setminus \{q\}$
                notify reconnection of $q$
            $N_p[q] \leftarrow max(N_p[q] + 1,\ n_q)$
    coend
```

Figure 3.4: Disconnection Detector for Partitionable Networks $\mathcal{DDP}$

- *Weak disconnection completeness*: There is a time after which all the disconnected processes are seen as disconnected by at least one connected process in the same partition. Formally,

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \exists t \in \mathcal{T},\ \forall p \in disconnect(DC),\ \exists q \in connected(DC), \exists q \in partition(p),\ \forall t' \geq t : p \in H_{DC}(q,\ t')$$

- *Strong disconnection accuracy*: No process is seen as disconnected by a process in the partition until it disconnects. Formally,

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \forall t \in \mathcal{T},$$
$$\forall p \in partition(q),\ q \in \Pi - DC(t) : p \notin H_{DC}(q,\ t)$$

### 3.3.3 Heartbeat Failure Detector with Partitions

Heartbeat failure detector for partitions is based on the algorithm for partitionable networks defined by [Aguilera et al., 1999] (cf. Figure 4). Below, we list the algorithm and its properties. Section A.2 in the appendices provide the proof of the algorithm.

The failure detector $\mathcal{HBDP}$, taking into account partitions, is presented in Figure 3.5. The algorithm is divided into two parallel tasks:

1. In the first task, each process $p$ periodically increments its own heartbeat. The process sends a heartbeat message $(\mathsf{HEARTBEAT}, path)$ to all the neighbors.

2. The second task handles the receipt of messages of the form $(\mathsf{HEARTBEAT}, path)$. Upon the receipt of such a message from process $q$, $p$ adds all the processes that appear after $p$ in $reachable_p[q]$, avoiding duplicated entries in the set. We have not explicitly defined that we are avoiding duplicate entries but it is not very difficult to add a module that does so. Therefore, $reachable_p[q]$ contains a list of processes that are not directly connected to $p$ but they can be communicated through process $q$. Henceforth, the set reachable tries to collect information about the processes which can be communicated through some neighbor. Process $p$ increases the heartbeats of $r$ for all the processes that appear after $p$ in $path$. Process $p$ appends itself to $path$ and forwards this message $(\mathsf{HEARTBEAT}, path)$ to all the neighbors that appear at most once in $path$.

There are few changes that have been introduced in original $\mathcal{HB}$ for partitionable networks, consequently we do not need to prove the properties that have already been proved by the authors. The only changes are:

1. Lines 5, 6 and 7 introduce a new set called $reachable$, for each neighbor.

2. Line 16 adds the processes appearing in $path$ after $p$ to $reachable$.

With all the changes, $\mathcal{HBDP}$ satisfies $\mathcal{HB}$-**Completeness** and $\mathcal{HB}$ **-Accuracy** (cf. Section A.2 for proof of the algorithm ).

```
for every process $p$ :
    initialization :
        for all $q \in \Pi$
            $D_p[q] \leftarrow 0$                          $\{D_p$ is the output of $\mathcal{HBDP}$ at $p\}$
            $reachable_p[q] \leftarrow \emptyset$
        for all $q \in neighbor(p)$ do    $\{neighbor(p)$ is given at configuration time$\}$
            $reachable_p[q] \leftarrow \{q\}$
    cobegin :
        || task 1 : repeat periodically
            $D_p[p] \leftarrow D_p[p] + 1$                     {Increment $p$'s own heartbeat}
            for all $q \in \Pi$ such that $q \in neighbor(p)$
                send(HEARTBEAT, $p$) to $q$
        || task 2 : upon receive(HEARTBEAT, $path$) from $s$
            for all $r \in \Pi$ such that $r$ appears after $p$ in $path$ and $q$ appears right next
            to $p$ in $path$
                $reachable_p[q] \leftarrow reachable_p[q] \cup \{r\}$   {Avoiding duplicate entries}
                $D_p[r] \leftarrow D_p[r] + 1$
            $path \leftarrow path.p$
            for all $r \in \Pi$ such that $neighbors(p) \wedge r$ appears at most once in $path$
                send(HEARTBEAT, $path$) to $r$
    coend
```

Figure 3.5: Heartbeat Failure Detector for Partitionable Networks $\mathcal{HBDP}$

### 3.3.4 Partition Detection with Neighborhood Topology

Neighborhood topology tries to discover partition formation using the network topology of the neighbors. The idea of neighborhood connectivity is to find the neighbors of the neighbors. In Figure 3.6, process $p$ tries to discover the neighbors of process $w$ and $q$. We construct our neighborhood topology in two layers, as shown in Figure 3.3. The lower layer, consisting of $\mathcal{HBDP}$ and $\mathcal{DDP}$, collects the reachability information and passes it to the $\mathcal{PDN}$, which uses this information for finding partitions in the network. Partition detection starts as soon as a disconnection or failure is detected in the system. In this section, the partition detector algorithm that uses the two previously defined algorithms, $\mathcal{DDP}$ and $\mathcal{HBDP}$ for partition detection, is explained. The algorithm is listed in below along with its properties. The proofs are given in appendices (cf. Section A.3).

The partition detector algorithm in Figures 3.7 and 3.8 is divided into four parallel tasks:

1. The first task periodically examines the heartbeat counters and reachability sets. First, the process gets a list of processes seen as reachable by $\mathcal{HBDP}$. The set

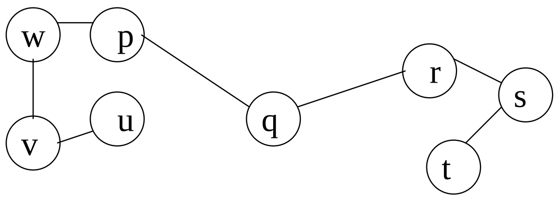

Figure 3.6: Reachability Pattern

$reachable_p$, acquired from $\mathcal{HBDP}$ is made coherent with respect to local sets in order to avoid any conflict with the local set of reachable processes. Thus, all the processes seen as reachable by $\mathcal{HBDP}$ but disconnected, faulty or partition locally are not added to the set $reachable_p$. The difference between the values of the new heartbeat and the old heartbeat is compared against a failure detection threshold $k$. If the difference is less than the failure detection threshold then the process is suspected to have crashed. The process certifies its mode to be connected. The suspected process is added to the set $faulty_p$, which contains the process suspected to have crashed. Process $q$ is then eliminated from $reachable_p[q]$. All the other processes in $reachable_p[q]$, which are not found in the set $reachable_p[i]\ \forall i \in neighbor(p)$, are added to the set $partition_p[q]$. Thus, in $partition_p[q]$, we have all the processes that are no more reachable due to the failure of $p$. A partition message $send(pid, q, r, part_q[r])$ is sent to all the neighbors. $pid$ is used to discard any old messages still roaming in the network. It is generated by a global entity and whenever a process want to send a new partition message, it obtains a new $pid$ and sends the message. $fchg$ tracks the changes in the set $reachable_p$, is set to true. The set $reachable_p[q]$ is emptied. The other possibility is that of a false suspicion where a process is wrongly suspected to have crashed. The process which was falsely suspected is removed from the list of faulty processes. The processes believed to have partitioned due to the false suspicion are restored and $fchg$ is set to true. If there is a change in the $reachable_p$ ($fchg$ set to true), the upper layers are reported of this change.

2. The second task monitors the disconnection notifications from process $q$ and of its own . If a process $q$ sends a disconnection message, the process is added to the set $discon_p$. The disconnected process is then eliminated from $reachable_p$. The processes seen as reachable through $q$ are then added to the set $partition_p[q]$. A partition message, $(pid, q, r, part_q[r])$, is sent to all the neighbors. $pid$ is used to discard any old messages still roaming in the network. It is generated by a global entity and whenever a process want to send a new partition message, it obtains a new $pid$ and sends the message. Afterwards, the set $reachable_p[q]$ is emptied and a notification is sent that there is a change in the $reachable$ set. If a process $q$,

earlier suspected to have crashed, sends a disconnection message, it is removed from the $faulty_p$ set and it is appended to $discon_p$ set. If a process receives its own disconnection message, the sets $partition_p$ is emptied for all the processes and the set reachable is emptied for all the neighbors.Finally, a reachable change notification is sent.

3. The third task handles the reconnection messages. If a reconnection message is received for a process $q$ other than p and if $q$ exists in the set $neighbor(p)$, then $reachable_p[q]$ is initialized to contain $q$. If a process receives its own reconnection message, the configuration file containing the list of neighbors is reread to initialize the network topology and all the neighbors are added to the list of reachable processes.

4. The fourth task handles the reception of the messages of type $send(pid, q, r, part_q[r])$. Every process that receives such a message, after verifying that it has not received message with this $pid$, appends the processes mentioned in the message to the set $partition_p$ to indicate the set of partitioned processes. The processes seen as partitioned are removed from the list of reachable processes. Then, it forwards this message to all its neighbors in order to disseminate the partition information to all the processes within $partition(p)$.

The abstract properties defined for the partition detector are:

- *Strong partition completeness*: There is a time after which all the partitioned processes $q$ are seen as partitioned by all the processes $p \notin partition(p)$ . Formally,

$$\forall q \in \Pi \setminus \{partition(p) \,\cup\, faulty_p \,\cup\, discon_p\},\, \forall H_{PD},\, \exists t \in \mathcal{T},\, \forall t' \geq t :\\ q \in H_{PD}(p,\, t')$$

- *Strong partition accuracy*: No correct and connected process is seen as partitioned until it partitions. Formally,

$$\forall q \in partition(p),\, \forall H_{PD},\, \forall q,\, \forall t \in \mathcal{T} :\, q \notin H_{PD}(p,\, t)$$

### 3.3.5 Partition Detection with Global Topology

In this section, the second partition detector algorithm that uses the two previously defined algorithms, $\mathcal{DDP}$ and $\mathcal{HBDP}$ for partition detection, is explained. In the following paragraphs, the algorithm is listed along with its properties. The proof is given in Section A.4.

The algorithm $\mathcal{PDG}$ is presented in Figure 3.9 and Figure 3.10. This algorithm constructs a knowledge of global topology instead of neighborhood topology. In the

```
for every process $p$ :
    initialization :
        $faulty_p \leftarrow \emptyset$                    {Processes suspected to be faulty}
        $discon_p \leftarrow \emptyset$                    {Processes suspected to have disconnected }
        $fchg \leftarrow false$                    {boolean stating whether $faulty_p$ changes}
        $reachable_p \leftarrow \emptyset$                    {$reachable_p[q], \forall q \in neighbor(p)$}
        $pid \leftarrow 0$                    {Identifi er for partition messages}
        $recv\_pid \leftarrow 0$                    {Tracks the received $pid$}
        for all $q \in \Pi$
            $oldD_p[q] \leftarrow 0$                    {$oldD_p$ is the previous output of $\mathcal{HBDP}$ at $p$}
            $D_p[q] \leftarrow 0$                    {$D_p$ is the output of $\mathcal{HBDP}$ at $p$}
            $partitioned_p[q] \leftarrow \emptyset$                    {Processes suspected to be partitioned}
    cobegin :
        || task 1 : repeat periodically
            for all $q \in \Pi \setminus discon_p$
                $reachable_p \leftarrow getReachable() \setminus \{discon_p \cup faulty_p \cup partition_p\}$ {Get
                $reachable_p$ from $\mathcal{HBDP}$}
                if $(D_p[q] - oldD_p[q]) \leq k$                    {$k$: failure detection threshold}
                   $\wedge q \in reachable_p$                    {$reachable_p[q], \forall q \in neighbor(p)$}
                   $\wedge getMode() =$ ‘c’ then
                    $faulty_p \leftarrow faulty_p \cup \{q\}$
                    $reachable_p[q] \leftarrow reachable_p[q] \setminus \{q\}$
                    if $q \in neighbor(p)$
                        for all $r \in reachable_p[q]$ such that $reachable_p[r] = \emptyset \wedge \forall s \in \Pi \setminus$
                        $\{q\}$ : $r \notin reachable_p[s]$
                            $partition_p[q] \leftarrow partition_p[q] \cup \{r\}$
                        $fchg \leftarrow true$
                        $reachable_p[q] \leftarrow \emptyset$                    {Empty the corresponding entry}
                        $pid \leftarrow getPid()$                    {Get an identifi er for partition message}
                        for all $r \in neighbor(p)$
                            qr_neighbor_send$(pid, p, q, partition_p[q])$ to $r$
                else if $(D_p[q] - oldD_p[q]) > k \wedge q \in faulty_p$                    {False suspicion}
                    $faulty_p \leftarrow faulty_p \setminus q$
                    for all $r \in partitioned_p[q] \wedge q \in neighbor(p)$
                        $reachable_p[q] \leftarrow partition_p[q]$
                    $fchg \leftarrow true$
                    $partition_p[q] \leftarrow \emptyset$
                if $fchg$ then notify new $reachable_p$                    {Possible new view}
                $oldD_p \leftarrow D_p$                    {prepare next task’s execution}
```

Figure 3.7: Partition Detector with Neighborhood Topology $\mathcal{PDN}$

```
|| task 2 : upon disconnection notification of $q$
    if $p \neq q \wedge q \in reachable_p$
        if $q \notin faulty_p$
            $discon_p \leftarrow discon_p \cup \{q\}$
            $reachable_p[q] \leftarrow reachable_p[q] \setminus \{q\}$
            if $q \in neighbor(p)$
                for all $r \in reachable_p[q]$ such that $reachable_p[r] = \emptyset \wedge \forall s \in \Pi \setminus \{q\} : r \notin reachable_p[s]$
                    $partition_p[q] \leftarrow partition_p[q] \cup \{r\}$
                $reachable_p[q] \leftarrow \emptyset$            {Empty the corresponding entry}
                $pid \leftarrow getPid()$            {Get an identifier for partition message}
                for all $r \in neighbor(p)$
                    qr_neighbor_send($pid, p, q, partition_p[q]$) to $r$
        else if $q \in faulty_p$
            $faulty_p \leftarrow faulty_p \setminus \{q\}$
            $discon_p \leftarrow discon_p \cup \{q\}$      {Do not recalculate $partitioned_p[q]$}
    else
        $partition_p \leftarrow \emptyset$            {Empty the partitioned set for all $q$}
        for all $q \in neighbor(p)$
            $reachable_p[q] \leftarrow \emptyset$ {Empty the reachable set on self-disconnection}
        notify new view $reachable_p$            {Possible new view with $p$ only}
|| task 3 : upon reconnection notification of $q$
    if $q \neq p$
        $partition_p[q] \leftarrow \emptyset$            {Empty the partitioned set for $q$}
        $discon_p \leftarrow discon_p \setminus \{q\}$            {Remove $q$ from $discon_p$}
        if $q \in neighbor(p)$
            $reachable_p[q] \leftarrow \{q\}$            {Initialize if $p \in neighbor$}
    else
        for all $q \in neighbor(p)$            {given at configuration time}
            $reachable_p[q] \leftarrow \{q\}$
            $partitioned_p[q] \leftarrow \emptyset$
    notify new view $reachable(p)$
|| task 4 : upon $receive(pid, q, r, part)$ from $s$
    if $recv\_pid \neq pid$
        for all $t \in part$ such that $t \notin faulty_p$
            $partition_p[r] \leftarrow partition_p[r] \cup \{t\}$   {$faulty_p \cap partition_p = \emptyset$}
            $reachable_p[s] \leftarrow reachable_p[s] \setminus \{t\}$     {Remove from $reachable_p$ of sending process}
        for all $t$ such that $t \in neighbor(p)$
            qr_neighbor_send($pid, q, r, part$)
        $recv\_pid \leftarrow pid$
coend
```

Figure 3.8: Partition Detector with Neighborhood Topology $\mathcal{PDN}$ (Ctn'd)

algorithm, $p \leftrightarrows q$ means that there is a fair path from $p$ to $q$ ($p \rightarrow q$) and there is a fair path from $q$ to $p$ ($q \rightarrow p$). We assume that the topology (processes plus links) is known at the starting time of the distributed application —*i.e.*, there exist configuration data describing the initial configuration of the distributed application and these data are known to every process of the distributed application. Then, during the execution, new processes and new links are explicitly added and committed by all the processes of a partition. Each process updates the graph by tagging nodes due to disconnections, failure, or partitions. The basic functionality of the algorithm is the same except for some minor changes such as the set reachable is kept implicit for each process and equals to the relation $\Pi \setminus (d_p \cup f_p \cup p_p)$. One major change is that partition message is not needed to be sent to any neighbors. This is because the knowledge of partitions is global in this algorithm and all the processes in the $partition(p)$ can detect the process that have partitioned. Boolean variable $fchg$, in the first task, tracks the changes in reachability and in case of a change, sends a notification to the concerned layers. Whenever a process reconnects, it reconnects in the same neighborhood.

### 3.3.6 Topology and Application-based Decisions

Our work differentiates between physical topology and logical topology. The difference between the physical topology and the logical topology comes from the fact that in our work, we only consider two processes reachable if they have the ability of communicate with each other. For us, this is a higher level of abstraction than actually looking at physical links that link two or more processes. We combine the reachability information with the physical topology by discovering the links connecting two neighbors in $\mathcal{PDN}$, meaning that we gather the reachability information with the help of the individual neighbors. Consequently, every process constructs its local view by discovering the processes reachable through a neighbor and collecting them in one set. Thus, a neighborhood logical topology is built, instead of actual link in order to estimate the connectivity patterns. The second algorithm, $\mathcal{PDG}$ keeps a global information of each link and process. In case of link and process failures, the global topology is affected. This change is seen by all the processes within the current partition and they tag the links and processes.

Failure detectors and neighborhood information can only provide hints for applications. Thus, the sets of processes in the two partitions can only "speculate" what has gone wrong on the other side, since they cannot communicate directly. It is the application which will decide, considering its functionality, whether to drop the users or open new connections to them. Thus, there can be an optimistic approach and a pessimistic approach.

Processes can follow optimistic heuristics for the processes in the other partition. The underlying idea is to consider the processes behind a faulty process to be correct. The processes in the other partition can be waited for before dropping them off from

```
for every process $p$ :
  initialization :
    for all $q \in \Pi$
      $oldD_p[q] \leftarrow 0$          {$oldD_p$ is the previous output of $\mathcal{HBDP}$ at $p$}
      $D_p[q] \leftarrow 0$          {$D_p$ is the current output of $\mathcal{HBDP}$ at $p$}
    $p_p \leftarrow \emptyset$          {Processes suspected to be partitioned}
    $f_p \leftarrow \emptyset$          {Processes suspected to be faulty}
    $d_p \leftarrow \emptyset$          {Processes suspected to have disconnected }
    $fchg \leftarrow false$          {boolean stating whether $f_p$ changes}
  cobegin :
    || task 1 : repeat periodically
      $fchg \leftarrow false$
      if $p \notin d_p$          {current process not disconnected}
        $D_p \leftarrow getD_p()$
        for all $q \in \Pi \setminus (d_p \cup p_p \cup \{p\})$
          if $|D_p[q] - oldD_p[q]| \leq k$          {$k$: failure detection threshold}
            if $q \notin f_p$          {$q$ not already suspected}
              $f_p \leftarrow f_p \cup \{q\}$
              $fchg \leftarrow true$
              for all $r \in \Pi \setminus (d_p \cup f_p \cup p_p) : \neg p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p), E)$
                $p_p \leftarrow p_p \cup \{r\}$
        for all $q \in f_p$
          if $|D_p[q] - oldD_p[q]| > k$          {false suspicion}
            $f_p \leftarrow f_p \setminus \{q\}$
            for all $r \in p_p$ : $p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p) \cup \{r\}, E)$
              $p_p \leftarrow p_p \setminus \{r\}$
            $fchg \leftarrow true$
      if $fchg$ then notify new view $(\Pi \setminus (d_p \cup f_p \cup p_p)$
      $oldD_p \leftarrow D_p$          {prepare next task's execution}
```

Figure 3.9: Partition Detector with Global Topology $\mathcal{PDG}$

```
        || **task 2** : **upon** disconnection notification of $q$
            $d_p[q] \leftarrow q$
            **if** $q \in faulty_q$                {$q$ suspected before seen disconnected}
                $faulty_q \leftarrow faulty_q \setminus \{q\}$
            **if** $q \in partition_q$             {fair path suspected before receiving the
            disconnection message}
                $partition_q \leftarrow partition_q \setminus \{q\}$
            **for all** $r \in \Pi \setminus (d_p \cup f_p \cup p_p)\ :\ \neg p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p), E)$
                $p_p \leftarrow p_p \cup \{r\}$
            **notify** new view $(\Pi \setminus (d_p \cup f_p \cup p_p)$
        || **task 3** : **upon** reconnection notification of $q$
            **if** $p = q$                 {$p$ reconnects, so no knowledge of failures...}
                $d_p \leftarrow \emptyset$
                $f_p \leftarrow \emptyset$                   {begin assuming no faulty process}
                **for all** $r \in p_p\ :\ p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p) \cup \{r\}, E)$
                    $p_p \leftarrow p_p \setminus \{r\}$
            **else**
                $d_p \leftarrow d_p \setminus \{q\}$
                **for all** $r \in p_p\ :\ p \leftrightarrows q$ in $(\Pi \setminus (d_p \cup f_p \cup p_p) \cup \{r\}, E)$
                    $p_p \leftarrow p_p \setminus \{r\}$
            **notify** new view $(\Pi \setminus (d_p \cup f_p \cup p_p)$
    **coend**
```

Figure 3.10: Partition Detector with Global Topology $\mathcal{PDG}$ (Ctn'd)

the application. There might be a disconnection such that the routing process moved out of the communication range without being able to send a disconnection alert. The process may, sometime later, move in again. Another approach is to open new connections through alternative network resources to the processes in the other partition. This can work where there is a possibility of various communication channels, for example, WIFI or GPRS. Thus, a new route may be opened connecting all the processes in the other partition and consequently, merging the two partitions. Henceforth, it will be the application that, according to its performance needs, reposes its decision.

Depending on its requirements, an application may not want to delay its processing, and may decide to drop the processes that are in the other partitions. The application can do this as soon as the processes are partitioned. The two partitions may proceed in their computations with degraded performance, but, as a consequence, the partitions do not merge.

## 3.4 Wireless Group Membership Service

Group communication systems provide consistency and fault-tolerance in distributed applications. With the rapid growth of mobile networks it is necessary to devise GCS for wireless environments. In this section, we propose a membership service based on the disconnection, failure and partition detection services (cf. Figure 3.3). The membership service strives to build the views based on the information of the three detectors. We try to build two solutions for each of the partition detector $\mathcal{PDN}$ and $\mathcal{PDG}$. We use the same underlying network model defined in Section 3.3.1. Section 3.4.1 defines the general set of properties desired from the membership service. The outline of the membership algorithm is presented in Section 3.4.2. The idea of agreeing on the composition of the sets of disconnected, faulty and partitioned processes is presented in Section 3.4.3

### 3.4.1 Membership Properties

Our set of properties have been carefully designed keeping in view the mobility of the terminal. The set of specifications for the membership algorithm are:

- **Self Inclusion**: If process $p$ installs a view $V$, then $p$ is a member of $V$. Formally,

$$installs(p, V){:}p \in V.members$$

- **Local Monotonicity**: If a process $p$ installs a view $V$ after installing view $V'$ then the identifier of $V$ is greater than that of $V'$. Formally,

$$e_i = view\_chng(p, V, T) \wedge e_j = view\_chng(p, V', T') \wedge i > j{:}V.id > V'.id$$

- **Initial View Event**: Every $send$, $receive$, and $safe\_prefix$ event occurs within some view. Formally,

$$t_i = send(p, m) \vee t_i = recv(p, m) \vee t_i = safe\_prefix(p, m){:}viewof(t_i) \neq \emptyset$$

- **Eventual Strong View Accuracy**: There is a time after which mutually reachable processes belong to the same view. Formally,

$$\exists t_0, \forall t \geq t_0,\ q \in partition(p){:}\exists t_1, \forall t \geq t_1 : q \in view(p, t)$$

- **Strong View Completeness**:There is a time after which all the processes which are not mutually reachable from $p$ do not belong to the view including $p$. Formally,

$$\exists t_0,\ \forall t \geq t_0,\ \forall q,\ q \notin partition(p){:}\exists t_1,\ \forall t \geq t_1 : q \notin view(p, t)$$

### 3.4.2 Group Membership Algorithm

We propose only the sketch of the algorithm that we propose for the membership service. This is a very simple membership service that operates on top of the two partition detectors and tries to fulfill the properties defined in Section 3.4.1. The partition detector provides the set of reachable processes and the membership service tries to build a view consisting of all the mutually reachable processes in $reachable_p$ ($reachable_p$ corresponds the variable in membership service). Every process $p$ that detects a change in its set $reachable_p$ sends a message $(\mathsf{JOIN\_VIEW}, new\_vid, reachable_p)$ to all the processes in the new set $reachable_p$. We assume that there is an entity in the systems that generates the view identifiers ($vid$). Thus, it generates view identifiers ($vid$) that are monotonically increasing with time. Every process that receives the view change message may be in two states: it may be in a view having a $vid$ lower than that of the sending process or the process is already in the process of changing a view. For the first case, the process $q$ accepts the message, changes its set $reachable_q$ according to the information sent by $reachable_p$ and sends an ACK message. The process that initiates the view change process, adds all the processes which sent an ACK to the new view. A new view message is sent to all the process in the new view of the form $(new\_vid, vcomp)$ corresponding to new view identifier and the view composition respectively. If it receives a NACK from one or more processes, and that may occur when a process is already in a view having a higher $vid$, restarts the algorithm with the higher $vid$. The process may also ABORT, depending on the requirements. Everyone joins a new view on receiving a new view message. If a process is already in the view change process and it receives a message with higher $vid$, it relays that message to all the processes so that everyone knows of that new view message. Consequently, old view session ends and the new one becomes the active membership proposal. A disconnected process may install a new view consisting of its own so, it can carry on its computation even in the disconnected mode. If the view initiator fails during the algorithm then the network reachability changes and the processes are forced to start a instance of the membership algorithm.

It can be shown that the above algorithm follows the membership properties defined in Section 3.4.1. The first three properties can be shown to be satisfied in each run. Eventual strong view accuracy and strong view completeness are satisfied by the partition and disconnection detector because they satisfy these properties for every disconnection, failure and partition. If a process disconnects, fails or partitions, then, eventually, every correct and connected process satisfies these properties and suspects the process as disconnected, failed or partitioned. Thus, this algorithm is simple enough to be implemented in this form. But there are some applications which require more stronger requirements for consistency and coherency. We consider that the above mentioned group membership algorithm can be adapted according to the needs.

In general, we can say that we have proposed a disconnection, failure and partition detection service that can be a foundation for a group communication systems. The set $reachable_p$ always proposes a set of $reachable$ processes ($reachable = \Pi \setminus (d_p \cup f_p \cup p_p$ is in case of $\mathcal{PDG}$), and all the processes in $partition(p)$ at one moment can become a part of the current view. One such membership algorithm has been proposed by [Babaoglu et al., 2001]. They build their membership service on top of the set $reachable$. In the original algorithm, they use only the failure detectors to build the set $reachable$. They do not perform partition detection. We plan to reuse their membership algorithm on top of the partition detectors defined above. A view is formed for all the processes in the set $reachable$. The membership algorithm presented in their work is a comprehensive one. But the only difference lies in the underlying model. In our model, we consider links that do not recover after crashing. They consider links that can recover even after they crash.

### 3.4.3 Group Agreement Algorithm

In this section, we present a sketch of an algorithm that operates on top of the two partition detectors $\mathcal{PDN}$ and $\mathcal{PDG}$. The distributed computation carried out by the agreement service tries to build a consensus over the three sets of processes: disconnected, faulty and partitioned. Thus, where a group membership algorithm tries to build a set of mutually reachable processes, group agreement algorithm proposes an approach which has not been used before, that is, agreeing on the composition of non-reachable processes. We propose this algorithm because of the fact that the composition of these sets may diverge for individual process, hence requiring an agreement. The consensus algorithm can be based on the rotating coordinator paradigm used in [Chandra and Toueg, 1996]. They exchange an estimate until all the processes decide. Using the same form, the three sets are exchanged by all the processes in a $partition(p)$ to agree on their composition. They mutually agree on the invariant $discon_p \cap faulty_p \cap partition_p = \emptyset$ for $\mathcal{PDN}$. For the algorithm $\mathcal{PDN}$, they agree on the invariant $d_p \cap f_p \cap p_p = \emptyset$. Eventually, they mutually agree on the processes seen as disconnected, faulty or partitioned and avoiding any wrong suspicion in terms of disconnection, failure and partition. Consequently, deciding which processes are not reachable by agreeing on the three sets and the reason, why they are not reachable by agreeing on the individual composition of the three sets. This algorithm can be used by applications that rely on the composition of the three sets for making critical decisions. For example, an application does not drop all the processes suspected to have partitioned and waits for them while there is a process appears in the set $partition_p$ but actually it has failed. Thus, making an agreement on the set of faulty processes can stop application waiting indefinitely.

## 3.5 Conclusion

In this chapter, we have defined two partition detectors based on the information provided by disconnection and failure detectors.

The first partition detection algorithm uses the information provided by $\mathcal{HBDP}$ to calculate the neighborhood topology. A set is created for every neighbor and this is appended with the processes that are reachable through that neighbor. Consequently, partitions are detected if a connecting process disconnects or fails and the processes reachable through that neighbor are tested for reachability. If they are not reachable through other processes, a partition is declared. This information is sent to other hosts through reliable primitive $qr\_neighbor\_send$. Two invariants are maintained during the execution of the algorithm: $discon_p \cap faulty_p \cap partition_p = \emptyset$ and $reachable_p = \Pi \setminus (discon_p \cup faulty_p \cup partition_p$.

The second partition detector creates a global view of the network topology. Initially, every process and every link is known and is added to the graph G. Then every process that connects and new link is created, this change is committed in the graph is added to this graph. In this way, a global view of the network topology is maintained by every process. In case of a disconnection or a failure, every process, knowing the global topology, calculates any processes that might have partitioned. The invariant $d_p \cap f_p \cap p_p = \emptyset$ is maintained during the execution of the algorithm. If there are network partitions, every partition keeps track of its the changes visible to the partition and this information is made global on merges.

There are two categories of heuristics for the processes detected to be partitioned. The optimistic approach, considers every process in the partition to be correct. The application may not drop such processes but waits for them until the link disconnecting such processes restores, in case of a disconnection. Another work around may be to open new connection to the processes on the other side of the partition. The pessimistic approach considers the processes in the other partition to be faulty and they are dropped from the application as soon as they partition.

We have defined only sketches of the group membership service because we could not complete their proofs. The partition detector defined in Section 3.3.4 and Section 3.3.5 can support any membership service that builds it views consisting of the members of the reachable set. We define a new paradigm for agreeing on the processes not in the partition and reason for their absence from the partition, as seen by processes in the partition. These processes make an agreement over the composition of the three sets disconnected, faulty and partitioned.

# Chapter 4

# Conclusions and Perspectives

Mobile Computing is gaining more and more importance with time. Need of the day is to target the problems which are emerging due to mobility of the terminals. Disconnection is one of such problems, which springs up when a mobile terminal moves out of the communication range. Disconnection can be very frequent and may depend on the mobility pattern of the user. Another problem that may occur in both fixed and wireless network is the problem of process and link crashes. These may hamper the progress of a distributed application or may render it completely unavailable. Thus, there is a need to detect and correct such unexpected behavior. There can be a scenario where disconnection or failure of a process can render two set of components completely unreachable, called partitions.

Group communication systems define a powerful paradigm for distributed systems where processes take the very same steps and exchange the same set of messages in order to preserve the overall consistency of the application. Processes are organized as multicast groups, called views. Every process within a view shares the same set of messages, which is called view synchrony. Primary group communication system allows only one view to exist at one time. Partitionable group communication services allow multiple views to progress in their computations.

In this work, we have tried to distinguish the three: disconnection, failure and partition. Disconnection and failure may lead to the formation of autonomous network components which can only communicate within their groups. We have modified the already existing disconnection detector to work for partitionable networks with fair links. The already existing heartbeat failure detector for partitionable networks has been modified to discover the reachability patterns of the underlying network. In the first partition detector, we try to discover the neighborhood topology. The idea of neighborhood topology is to discover the processes that are reachable from some neighbor. If that neighbor disconnects and fails, all the processes that were reachable only through that neighbor are declared to have partitioned. The second partition detector builds an overall view of the system with the initial processes and links. Afterwards, every new process and link is committed by all the processes. In case of failure or disconnection, the reachability between two processes is verified and if the disconnection and failure has affected reachability, the unreachable processes are added to the list of partitioned processes. Since we cannot ascertain the status of partitioned processes, we try to develop the heuristics or opinions based on the application requirements. These speculations can be optimistic and pessimistic. Certain ideas for optimization of failure detection can be developed based on the collected topology information.

Wireless group communication systems are becoming need of the day with the increasing use of wireless applications. Users can share collaborative applications, they can play multi-player games and they keep data consistency while they use wireless group communication systems. The membership service defined can be used to assist the use of such applications. We think that our solution is weak enough to be implemented.

During this work, we tried to search for any existing work for partition detection and we found nothing. Thus, we are giving here a framework for partition detection which is a general one, that is the solution proposed does not depend on network type or network topology. This framework can be adapted for various application types. We have already provided the a draft version of algorithm which supports reconnections in different partitions, along with its propreties (cf. Section A.5).We foresee that the disconnection, failure and partition detection can be based on one generic service, which can minimize the number of messages exchanged between various processes. For mobile terminals, where the battery is already too small to support normal operation of few hours, the computations and message complexity can be a huge burden. Consequently, there is a need for optimizing algorithms for their effectiveness. In this work, we only consider the processes and the links that never recover from a failure. Efforts are required in order to adapt them for a general failure model where links can fail and recover . Another advancement can be the addition of a multicast layer to complete the view synchrony model. The message exchange that reveals the neighborhood topology is based on a crude model. It doesn't serve to reveal the global topology because of the nature of the messages. One way is to ameliorate the message passing in order to reveal the global connectivity patterns. Thus, everyone detects the formation of a partition.

# Bibliography

[Aguilera et al., 1997] Aguilera, M., Chen, W., and Toueg, S. (1997). Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In Mavronicolas, M. and Tsigas, P., editors, *Proc. 11th*, volume 1320 of *Lecture Notes in Computer Science*, pages 126–140, Saarbrücken, Germany. Springer-Verlag.

[Aguilera et al., 1999] Aguilera, M. K., Chen, W., and Toueg, S. (1999). Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.*, 220(1):3–30.

[Amir et al., 1992] Amir, Y., Dolev, D., Kramer, S., and Malki, D. (1992). Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 292–312. Springer-Verlag.

[Anceaume et al., 1995] Anceaume, E., Charron-Bost, B., Minet, P., and Toueg, S. (1995). On the formal specification of group membership services. Technical Report TR95-1534.

[Babaoglu et al., 2001] Babaoglu, Z., Davoli, R., and Montresor, A. (2001). Group communication in partitionable systems: Specification and algorithms. *IEEE Trans. Softw. Eng.*, 27(4):308–336.

[Bertier et al., 2003] Bertier, M., Marin, O., and Sens, P. (2003). Performance analysis of hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, San-Francisco (USA). IEEE Society Press.

[Birman, 1986] Birman, K. P. (1986). Isis: A system for fault-tolerant distributed computing. Technical report.

[Burns et al., 1999] Burns, M. W., George, A. D., and Wallace, B. A. (1999). Simulative performance analysis of gossip failure detection for scalable distributed systems. *Cluster Computing*, 2(3):207–217.

[Chandra and Toueg, 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2).

[Conan et al., 2002] Conan, D., Chabridon, S., Villin, O., Bernard, G., Kotchanov, A., and Saridakis, T. (2002). Handling Network Roaming and Long Disconnections at Middleware Level. In *Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices (in conjunction with EDOC'2002)*, Lausanne (Switzerland).

[Dolev et al., 1996] Dolev, D., Malki, D., and Strong, R. (1996). A framework for partitionable membership service. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, page 343. ACM Press.

[Fekete et al., 2001] Fekete, A., Lynch, N., and Shvartsman, A. (2001). Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216.

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[Gärtner, 1999] Gärtner, F. (1999). Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1):1–26.

[Hayashibara et al., 2002] Hayashibara, N., Cherif, A., and Katayama, T. (2002). Failure detectors for large-scale distributed systems. In *the 21st IEEE Symposium on Reliable Distributed Systems (SRDS-21), the International Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS'2002)*, Osaka, Japan.

[Montresor et al., 1999] Montresor, A., Davoli, R., and Babaoglu, O. (1999). Middleware for dependable network services in partitionable distributed systems. Technical report.

[Renesse et al., 1998] Renesse, R. V., Minsky, Y., and Hayden, M. (1998). A gossip-style failure detection service. Technical Report TR98-1687.

[Temal and Conan, 2004] Temal, L. and Conan, D. (2004). Détections de défaillances, de connectivité et de déconnexions. In *Actes de la 1ère Conférénce Francophone Mobilité et Ubiquité*, pages 90–97, Nice, France. ACM Press.

[Vitenberg et al., 1999] Vitenberg, R., Keidar, I., Chockler, G., and Dolev, D. (1999). Group communication specifications: A comprehensive study. Technical Report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem. and MIT Technical Report MIT-LCS-TR-790.

# Appendix A

# Disconnection, Failure and Partition

## A.1 Disconnection Detector $\mathcal{DDP}$

In this section, we present the proof of the algorithm $\mathcal{DDP}$, presented in Section 3.3.2.

**Lemma 1** *If two processes $p$ and $q$ are in the same partition then every message sent by $p$ is received by $q$.*

**Proof.** This lemma has already been proved in [Aguilera et al., 1999], in the proof of Lemma 25 for their algorithm in Figure 4. We present it here precisely for our purposes. If $p$ and $q$ are in the same partition then there is a fair and simple path from $p$ to $q$ and every message sent infinitely often from $p$ to $q$ is received infinitely often by $q$. Thus, every message sent by $p$ to $q$ is eventually received. $\square$

**Lemma 2** ***Strong Disconnection Accuracy*** *No process $p$ is seen as disconnected by a process in $p$'s partition until $p$ actually disconnects.*

**Proof.** We can see in Figure 3.4 that a process adds another process to the set of disconnected processes, $disc_p$ only in line 21 of the algorithm. This instruction is executed only when a process receives a disconnection message from a process that has not been added to the set of disconnected processes and earlier no message has been received from that process. Another possibility is that the process may receive the disconnection message from some other process. If the message is found to be new, —*i.e.* not an "older" disconnection —, the process indicated in the message will be added to the list of disconnected processes. Otherwise, the message will be ignored. $\square$

**Lemma 3** ***Strong Disconnection Completeness*** *There is a time after which all the disconnected processes are seen as disconnected by all the connected processes in their partition.*

**Proof.** There can be four cases. In the first case, a process $p$ successfully sends the disconnection message to all its neighbors. Using the algorithm, this message is successfully disseminated to all the processes in the partition of $p$ by Lemma 1 and the definition of the primitive $qr\_neighbor\_send$. In the second case, $p$ sends the disconnection message to at least one correct and connected process $q$. If $q$ does not disconnect after receiving disconnection of $p$ then it successfully disseminates this message as done in the first case. But if the process $q$ disconnects right after receiving the DISCONNECTION message of $p$, or $p$ disconnects just before sending a DISCONNECTION message, the message is sent whenever process $p$ reconnects to the network. This is because of the properties of the primitive $qr\_neighbor\_send$; messages are saved in the mobile terminal's cache and are sent when the terminal reconnects. Hence, strong completeness is satisfied whenever the mobile terminal reconnects.

The fourth case is a subtle one. Since the links are fair, process $p$ sends a message just before its disconnection but the link may drop this message. No process will ever know of the disconnection and the process will be suspected faulty. For alleviating this problem and for making the disconnection message to be sent in a reliable manner, we use the primitive $qr\_neighbor\_send$ that sends a message relaibably on fair links. □

**Theorem 1** *Algorithm $\mathcal{DDP}$ implements a disconnection detector which satisfies strong disconnection completeness and strong disconnection accuracy.*

**Proof.** From Lemmata 2 and 3. □

## A.2 Heartbeat Failure Detector with Partitions $\mathcal{HBDP}$

In this section, the proof of the algorithm $\mathcal{HBDP}$, given in Section 3.3.3 is presented.

**Lemma 4** *At each correct and connected process $p$, for each neighbor $q$, the set $reachable_p[q]$ eventually contains every correct and connected process $r$, such that there is a fair path from $p$ to $r$ through $q$ and $r$ is in the partition of $p$.*

**Proof.** This lemma is trivially true for all neighbors of $p$ (lines 6 and 7 in the initialization phase).

In the proof of Lemma 25 in the original algorithm [Aguilera et al., 1999], it is stated that if there is a fair path between two processes $p$ and $r$, (we rename $q$ as $r$ for our convenience), such that $(p_1, p_2, .., p_i)$ is a simple fair path from $p$ to $r$ and $(p_i, p_{i+1}, ..., p_k)$ is a simple fair path from $r$ to $p$, so that $p_i = r$ and $p_1 = p_k = p$, then $p$ sends the message $(\mathsf{HEARTBEAT}, P_j)$ an infinite number of times to $r$ where $P_j = (p_1, ..., p_j)$ for $j = 1, ..., k$. Since there is a fair path from $p$ to $r$, $r$ receives messages of the form $(\mathsf{HEARTBEAT}, P_j)$ an infinite number of times. Thus, there is at least one process $p_2 \in neighbor(p)$ which forwards all the messages from $p$ to $r$ either directly or indirectly.

In the proof of lemma 25, it is proved that these messages reach process $p_{k-1}$, which then forwards them to $p$. As a consequence, $p$ receives messages $(\mathsf{HEARTBEAT}, P_{k-1})$, where $P_{k-1}$ is of the form $p, p_2.., r, .., p_{k-1}$. Thus, letting $p_2 = q$, $p$ successfully adds $r$ into the set $reachable_p[q]$, using line 15. □

**Lemma 5** *If a process $r$ is only reachable from process $p$ through $q$, where $p \in neighbor(q)$, then the process $r$ will only appear reachable through $q$ in all the processes in $p \in neighbor(q)$.*

**Proof** Figure 3.6 shows the arrangement of processes where $r$ is accessible to $p$ through $q$ only. We show that $r$ appears only in $reachable_p[q]$ at $p$. $\forall i \in neighbor(p)$, $p$ constructs the set $reachable_p[i]$ by adding all the processes to the set $reachable_p[i]$ in such a way that $i$ appears after $p$ in $path$. The processes that are appended to $reachable_p[i]$ appear after $p$ and $i$ in $path$. Consider a message $m$ initiated by process $p$ in the direction of $w$. Message $m$ traverses up to process $u$ and returns to $p$. Process $p$ appends processes $u$ and $v$ to the set $reachable_p[w]$. Thus, $u$ and $v$ only appear reachable through $w$. Now, this message $m$ is forwarded to $q$ since $q$ appears at most once in $path$ ($path = p, w, v, u, p$). Message $m$ travels up to process $t$ and arrives at $q$. But $q$ will not forward this message to $p$ since $p$ already exists twice in $path$. This way process $r$ never appears in $reachable_p[w]$. Message $m$ will disappear after it reaches every host on both sides of the network (avoiding the ping-pong effect). Thus, every process that is reachable through a process $q$ will only appear to be reachable through that process. Consequently, a process reachable from one and only one neighbor $q$ will appear in the set of $reachable_p[q]$ at process $p$. □

**Lemma 6** $\mathcal{HB}$***-Completeness*** *At each correct process p, the heartbeat sequence of every process not in the partition of p is bounded.*

**Proof.** We do not change the message passing of the original $\mathcal{HB}$ and the functionality added does not change the working of the original algorithm. Our additions try to extract the neighborhood reachability information from the heartbeat messages. Since messages are sent and received by all the correct processes in the same way, the completeness property is not modified. Thus, $\mathcal{HBDP}$ satisfies $\mathcal{HB}$-Completeness . □

**Lemma 7** $\mathcal{HB}$***-Accuracy*** *At each process p, the heartbeat sequence of every process is nondecreasing, and at each correct process p, the heartbeat sequence of every process in the partition of p is unbounded.*

**Proof.** This also follows from the proof of the original algorithm. And as said in Lemma 6, we do not interfere with the message exchanging functionality of the original $\mathcal{HB}$. Thus, the accuracy property holds for this algorithm as well. □

**Theorem 2** *$\mathcal{HBDP}$ satisfies $\mathcal{HB}$-**Completeness** and $\mathcal{HB}$ -**Accuracy**.*

**Proof.** From Lemmata 6 and 7. □

## A.3 Partition Detection Using Neighborhood Topology $\mathcal{PDN}$

**Lemma 8** *All the processes within $partition(p)$ are eventually added to $reachable_p$ ($reachable_p$ represents the set of all the $reachable_p[q]$).*

**Proof** We prove this lemma by contradiction. Consider a process $q$ that is in $partition(p)$ and is not found in $reachable_p$. In the first case, the process is wrongly suspected and and the eventual strong accuracy property of $\mathcal{HBDP}$ will ensure that $q$ will be added to $reachable_p$ because it is reachable from $p$. The second case may be that the process $q$ has just reconnected. If $p$ is the neighboring process and it receives a reconnection message, then process $p$ adds process $q$ to the set of reachable processes (line 26 cf. Figure 3.8). If this is not a neighboring process then the $\mathcal{HBDP}$ will add the process $q$ to $reachable_p$ using Lemma 4. In the final case, the system has just started and eventually the network topology will be discovered, thus adding the process to the set $reachable_p$. Thus, a process in $partition(p)$ is added to the set $reachable_p$, a contradiction. □

**Lemma 9** *All the processes not in $partition(p)$ are eventually added to the set $faulty_p$, $discon_p$, or $partition_p$ ($partition_p$ represents the set of all $partition_p[q]$).*

**Proof** We prove this lemma by contradiction. Consider a process $q$ that is not in $partition(p)$ but still it is found in the set $reachable_p$, that is $\mathcal{HBDP}$ considers the process reachable and it is not considered as faulty, disconnected or partitioned. So, the heartbeat of $q$ at $\mathcal{HBDP}$ of $p$ is unbounded. This suggests that there is a fair path from process $p$ to process $q$. Therefore $p$ and $q$ are in the same partition, a contradiction. □

**Lemma 10** *If a process $q$ fails then there is a time after which every process $p \in neighbor(q)$ considers $q$ faulty and adds all the processes connected to $p$ through $q$ to the set $partitioned_p[q]$ if the processe are no more reachable from $p$.*

**Proof** If a process $q$ fails then the heartbeat of $q$, which was earlier in the partition of $p$, will be bounded at $p$ according to the property of strong completeness of $\mathcal{HBDP}$. Eventually, this process will be considered as faulty and will be added to the set $faulty_p[q]$ by each neighbor according to the statement 20 in the algorithm in Figure 3.7. Then, process $p$ scans the set $reachable_p[q]$ for every process that is suspected to have partitioned by the failure of process $q$. If a process is found in $reachable_p[r]$, that is process $s$ is still reachable through $r$, then no action is taken. But, if process $s$ is not found in the set $reachable_p$ for all the processes, that is, it is not reachable any more according to Lemma 5, then $s$ is added to the set $partition_p[q]$, meaning that $s$ has partitioned due to the failure of $q$. □

**Lemma 11** *If a process $q$ disconnects then there is a time after which every process $p \in neighbor(q)$ considers $q$ as disconnected and adds all the processes connected to $p$ through $q$ to the set $partitioned_p[q]$, if the processes are no more reachable from $p$.*

**Proof** If a process $q$ disconnects then the process $p$ eventually receives a disconnection message from $q$ according to the property of strong disconnection completeness of $\mathcal{DDP}$. Eventually, this process will be considered as disconnected and will be added to the set $discon_p[q]$ by each neighbor according to the statement 4 in the algorithm in Figure 3.8. Then, process $p$ scans the set $reachable_p[q]$ for every process $s$ that was in $reachable_p[q]$ and that is suspected to have partitioned by the failure of process $q$. If the process is found in $reachable_p[r]$— i.e. process $s$ is still reachable through $r$—, no action is taken. But, if process $s$ is not found in $reachable_p$, that is, it is not reachable any more according to Lemma 5, then $s$ is added to the set $partition_p[q]$, implying that $s$ has partitioned due to the disconnection of $q$. □

- **Lemma 12** *There is a time after which every process $r$ that breaks up from a $partition(p)$ due to the disconnection of a connecting process $q$ is seen as partitioned by all the correct and connected processes in $partition(p)$.*

  **Proof** Every process $r$ that breaks off (that is, the process that is connecting this process to $partition(p)$) from $partition(p)$ does that because it was connected to the partition by a process $q$ that has disconnected. The situation is depicted in Figure 3.6 where $q$ disconnects making two distinct groups that are unreachable from each other. If a process $q$ disconnects, it is likely to send a disconnection message. We can have two cases. Either every process receives this message (according to strong disconnection completeness of $\mathcal{DDP}$) or this disconnection message does not reach any neighbors of $q$, which causes the processes to timeout on $q$, which is considered as failure. So, here we only treat the first case. The process $r$ successfully sends the disconnection message to all the processes. Lemma 12 infers that the neighbors that receives this disconnection message start executing task 2 (line 1 in Figure 3.8). While executing task 2, it is found out that $r$ is only reachable through $q$. This is done by scanning the set $reachable_p$ for all the neighbors. Since $r$ is reachable through $q$ only for process $p$, according to Lemma 5 process $p$ will not see $r$ reachable through any other process, that is, it will not appear in $reachable_p[w]$ in Figure 3.6. Process $p$ will add $r$ to $partition_p[q]$ (line 8 of Figure 3.8) and will be sent to other processes. The partition message is sent to every process $i$ in $partition(p)$, which adds process $r$ to its set $partition_i[q]$. □

- **Lemma 13** *There is a time after which every process $r$ that breaks up from a $partition(p)$ due to the failure of a connecting process $q$ is seen as partitioned by all the correct and connected processes in $partition(p)$.*

  **Proof** Every process $r$ that breaks off from $partition(p)$ does that because it was connected to the partition by a process $q$ that has failed. The situation is depicted in Figure 3.6. If the process $q$ fails, every process eventually times-out on $q$ (according to Lemma 18). The neighbors that time-out on $q$ start executing task 1 (line 14 of Figure 3.7). According to Lemma 10, every process considers $r$ to have partitioned due to the failure of $q$. Lemma 5 shows that since the process $r$ is reachable only through $p$, it will not appear reachable to process $p$ through some other neighbor. Process $p$ will add $r$ to $partition_p[q]$ (line 24 of Figure 3.7). A partition message will be sent to all the processes about the partition of $r$. Thus, every process $s$ that receives the partition message, executes task 4 (line 32 of Figure 3.8) and adds process $r$ to the set $partition_s[q]$. □

**Theorem 3** *$\mathcal{PDN}$ satisfies Strong Partition Completeness.*

**Proof** From Lemmata 12 and 13. □

**Lemma 14** ***Strong Partition Accuracy*** *There is a time after which every process that is reachable from $p$ appears in $partition(p)$.*

**Proof** A process is considered partitioned only if the connecting process fails or disconnects. In Figure 3.6, process $r$ becomes unreachable and thus, partitioned from $p$ in two cases: either $q$ disconnects or fails, or either any of the link between $p$ and $q$ fails. The second case is eventually considered as a failure because the heartbeat of $q$ times-out at $p$, the result is the same as the first case. If process $q$ remains correct and there are no link failures, no disconnection message will be generated or no process will time-out. Disconnection messages are generated only when the process disconnects (Section 3.3.2). Task 2 of the algorithm will not be executed and eventually, no process considers $r$ as partitioned or unreachable. If every process receives a timely heartbeat from $q$, it will not be suspected as a failure and thus task 1 will not consider $q$ faulty satisfying strong partition accuracy in both cases. □

**Lemma 15** *Every process that eventually appears in $partition_p$, $faulty_p$, or $discon_p$, eventually does not appear in $reachable_p$.*

**Proof** From Lemmata 8 and 9. □

**Lemma 16** *Every process that eventually appears in $reachable_p$, eventually does not appear in $partition_p$, $faulty_p$, or $discon_p$.*

**Proof** From Lemma 8, we know that every process in $partition(p)$ is added to the set $reachable_p$ at some process $p$. It is obvious from the algorithm that every time we add something to the set $reachable_p$, we remove it from one of the three sets. A process may leave the set $faulty_p$ and is added to the set $reachable_p$ after a false suspicion. Secondly, after a reconnection message a process is added to the set $reachable_p$ if a neighbor receives the message and the process is removed from the set $discon_p$ (line 23 and 26 in Figure 3.8). The third case occurs when a false suspicion is detected. The process is removed from the set $faulty_p$ and is added to the set $reachable_p$. Non-neighboring process will be eventually added to the set $reachable_p$ according to Lemma 8. □

**Lemma 17** *Every process $q$, $q \notin partition(p)$ appears either disconnected, partitioned or faulty, that is $partitioned_p \cap faulty_p \cap discon_p = \emptyset$.*

**Proof** There can be three cases where a process becomes unreachable from a process $p$. The heartbeat of a process $q$ may timeout at process $p$. In this case, the process is added to the set $faulty_p$ only if it doesn't exist in either of $discon_p$ or $partition_p$. The same is true for the disconnection as well, that is, no process is added to the set $discon_p$ if it already exists in $faulty_p$ or $partitioned_p$; whenever we receive a disconnection message from a process, we receive that because the process is reachable. This is impossible for a process to be partitioned if it is already disconnected or faulty since it will exist in $reachable_p[q]$ in that case. If the process has not disconnected or failed before and it is not suspected to have partitioned then it will appear in $reachable_p$ according to Lemma 8. Thus, it will be added to $partitioned_p[q]$ and will no more be reachable and will not appear in disconnected or faulty according to the test conditions on line 11 and 28. □

**Theorem 4** *$\mathcal{PDN}$ satifies the invariant $reachable_p = \Pi - \{faulty_p \cup discon_p \cup partition_p\}$.*

**Proof** From Lemmata 15 and 16. □

**Lemma 18** ***Strong Completeness*** *There is an instant after which every faulty process, not in $partition(p)$ is suspected by all the correct processes in $partition(p)$.*

**Proof** From Lemma 6, we know that the heartbeat of every process $q$, which is not in the partition of $p$, is bounded. Thus, all the processes in $partition(p)$ will time-out on the failure detection threshold $k$. Consequently, every process in $partition(p)$ will consider $q$ as faulty. □

**Lemma 19** ***Eventual Strong Accuracy*** *There is a time after which correct processes are not suspected by any correct process processes in $partition(p)$.*

**Proof** From Lemma 7, we know that the heartbeat counter of every process $q$, which is in the partition of $p$, is unbounded. Thus, no process in the $partition(p)$ will time-out on the failure detection threshold $k$ and consequently, no process will consider $q$ as faulty. □

**Theorem 5** *$\mathcal{PDN}$ implements a failure detector which satisfies strong completeness and eventual strong accuracy.*

**Proof** From Lemmata 18 and 19. □

## A.4 Partition Detection with Global Topology $\mathcal{PDG}$

**Lemma 20** *Every process that disconnects in $partition(p)$ is seen as disconnected and only as disconnected by all processes in $partition(p)$.*

**Proof** If a process disconnects then it sends a disconnection message to all the reachable processes. The process that receives the disconnection message from $\mathcal{DDP}$ executes task 2. The process is added to the set of disconnected processes. If the process was earlier wrongly suspected to have failed instead of being considered disconnected then it is removed from the set of the faulty processes. Since every process that disconnects is seen as disconnected according to Lemma 3. Thus, all the processes that receive disconnection message, see the sending process to have disconnected and process is seen to have disconnected. □

**Lemma 21** *Every process $q$ that fails in $partition(p)$ is seen as faulty and only as faulty by all the processes in $partition(p)$.*

**Proof** If a process $q$ fails then every process that considers $q$ reachable, will timeout on the heartbeat of $q$. Task 1 is executed and thus, the process is considered to be faulty and will be added to the set of failed processes. A failed process can never be seen as disconnected because disconnections are announced. A process that fails may be appended to the set $p_p$ because the connecting process had failed before. In this case, the process will only appear in the set $p_p$ and will be removed from the set of reachable processes, that is $\Pi \setminus (d_p \cup f_p \cup p_p)$. Thus, a faulty process may append itself into the set $f_p$ or $p_p$. But it will not appear in both of the sets at the same time. $\square$

**Lemma 22** *Every process that partitions from a $partition(p)$ is considered to be partitioned by all the processes in $partition(p)$.*

**Proof** A process or a set of processes can partition from the network in two ways. Either the connecting process fails or disconnects. At every disconnection or failure, the reachable processes are tested if they have partitioned. Every process that is reachable from the process is considered to be partitioned and added to the set $p_p$. In this way, every process that partitions, that is it is no more reachable due to failure or disconnection of another process is added to the set $p_p$ and thus, considered as partitioned and only partitioned. $\square$

**Theorem 6** *$d_p \cap f_p \cap p_p = \emptyset$ is maintained by $\mathcal{PDG}$.*

**Proof** From lemmata 21, 20, and 22.

## A.5 Reconnections without Preserving Neighborhood $\mathcal{PDN}$*

In this section, we present a modified version of the partition detector for the scenarios where reconnections do not occur in the same neighborhood. A process can use the broadcase mechanism for finding its new neighbors in the broadcast networks, e.g, wireless networks. A process can also obtain a list of its new neighbors by requesting the list from a predefined server where broadcast is not possible e.g, GPRS networks. This means whenever a process reconnects, it may reconnect to a totally new set of processes or to a new partition. In this case, the properties defined for disconnection detector in Section 3.3.2 are rendered useless. This is because disconnection completeness may not be fulfilled if the process reconnects in a new neighborhood after a disconnection. In Section A.5.1, we devise a new set of properties for disconnection detector which may not see processes connecting to the same neighborhood. The algorithm is retained because it satisfies these properties without any changes.

### A.5.1 Modified Properties for $\mathcal{DDP}$

- *Disconnection Reliability*: If one correct and connected process receives a DISCONNECT (*resp.* RECONNECT) message then every correct and connected process receive the disconnection message. Formally,

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \exists t \in \mathcal{T},\ \forall p \in disconnected(DC),\ \forall q \in connected(DC), \forall q \in partition(p),\ \forall t' \geq t : p \in H_{DC}(q,\ t')$$

- *Strong Disconnection Accuracy*: No process is seen as disconnected by a process in the partition until it disconnects. Formally,

$$\forall DC,\ \forall H_{DC} \in \mathcal{DD}(DC),\ \forall t \in \mathcal{T}, \\ \forall p \in parition(q),\ q \in \Pi - DC(t) : p \notin H_{DC}(q,\ t)$$

### A.5.2 Proof for $\mathcal{DDP}$

**Lemma 23** ***Strong Disconnection Accuracy*** *No process $p$ is seen as disconnected by a process in $p$'s partition until $p$ actually disconnects.*

**Proof.** This proof is same as that of Lemma 2.

**Lemma 24** ***Disconnection Reliability*** *If one correct and connected process receives a* DISCONNECT *(*resp. RECONNECT*) message then every correct and connected process receive the disconnection message.*

**Proof.** In Lemma 3, we proved that if a process successfully sends a message and the message is received by at least one correct process then the received disconnection message is received by all the processes in $partition(p)$. □

**Theorem 7** *DDP satisfies strong disconnection accuracy and disconnection reliability.*

**Proof** From lemmata 24 and A.5.2. □

### A.5.3 $\mathcal{PD} - \mathcal{I}*$

We present the modified version of the algorithm $\mathcal{PD} - \mathcal{I}$ called $\mathcal{PD} - \mathcal{I}*$. It inherits the same proof and properties from algorithm presented in Section 3.3.4. So, we only list the modified algorithm.

```
for every process $p$ :
  initialization :
    $faulty_p \leftarrow \emptyset$                    {Processes suspected to be faulty}
    $discon_p \leftarrow \emptyset$                    {Processes suspected to have disconnected }
    $fchg \leftarrow false$                    {boolean stating whether $faulty_p$ changes}
    $reachable_p \leftarrow \emptyset$                    $\{reachable_p[q], \forall q \in neighbor(p)\}$
    $pid \leftarrow 0$                    {Identifi er for partition messages}
    $recv\_pid \leftarrow 0$                    {Tracks the received $pid$}
    for all $q \in \Pi$
      $oldD_p[q] \leftarrow 0$                    {$oldD_p$ is the previous output of $\mathcal{HBDP}$ at $p$}
      $D_p[q] \leftarrow 0$                    {$D_p$ is the output of $\mathcal{HBDP}$ at $p$}
      $partitioned_p[q] \leftarrow \emptyset$                    {Processes suspected to be partitioned}
  cobegin :
    || task 1 : repeat periodically
      for all $q \in \Pi \setminus discon_p$
        $reachable_p \leftarrow getReachable() \setminus \{discon_p \cup faulty_p \cup partition_p\}$ {Get
        $reachable_p$ from $\mathcal{HBDP}$}
        if $(D_p[q] - oldD_p[q]) \leq k$                    {$k$: failure detection threshold}
          $\wedge q \in reachable_p$                    $\{reachable_p[q], \forall q \in neighbor(p)\}$
          $\wedge getMode() =$ ‘c’ then
          $faulty_p \leftarrow faulty_p \cup \{q\}$
          $reachable_p[q] \leftarrow reachable_p[q] \setminus \{q\}$
          if $q \in neighbor(p)$
            for all $r \in reachable_p[q]$ such that $reachable_p[r] = \emptyset \wedge \forall s \in \Pi \setminus$
            $\{q\} : r \notin reachable_p[s]$
              $partition_p[q] \leftarrow partition_p[q] \cup \{r\}$
            $fchg \leftarrow true$
            $reachable_p[q] \leftarrow \emptyset$                    {Empty the corresponding entry}
            $pid \leftarrow getPid()$                    {Get an identifi er for partition message}
            for all $r \in neighbor(p)$
              qr_neighbor_send$(pid, p, q, partition_p[q])$ to $r$
        else if $(D_p[q] - oldD_p[q]) > k \wedge q \in faulty_p$                    {False suspicion}
          $faulty_p \leftarrow faulty_p \setminus q$
          $fchg \leftarrow true$
          $partition_p[q] \leftarrow \emptyset$
        if $fchg$ then notify new $reachable_p$                    {Possible new view}
        $oldD_p \leftarrow D_p$                    {prepare next task’s execution}
```

Figure A.1: Partition Detector with Neighborhood Topology $\mathcal{PDN}$*

```
|| task 2 : upon disconnection notification of $q$
    if $p \neq q \land q \in reachable_p$
        if $q \notin faulty_p$
            $discon_p \leftarrow discon_p \cup \{q\}$
            $reachable_p[q] \leftarrow reachable_p[q] \setminus \{q\}$
            if $q \in neighbor(p)$
                for all $r \in reachable_p[q]$ such that $reachable_p[r] = \emptyset \land \forall s \in \Pi \setminus \{q\} : r \notin reachable_p[s]$
                    $partition_p[q] \leftarrow partition_p[q] \cup \{r\}$
                $reachable_p[q] \leftarrow \emptyset$          {Empty the corresponding entry}
                $pid \leftarrow getPid()$          {Get an identifier for partition message}
                for all $r \in neighbor(p)$
                    qr_neighbor_send$(pid, p, q, partition_p[q])$ to $r$
        else if $q \in faulty_p$
            $faulty_p \leftarrow faulty_p \setminus \{q\}$
            $discon_p \leftarrow discon_p \cup \{q\}$     {Do not recalculate $partitioned_p[q]$}
    else
        $partition_p \leftarrow \emptyset$          {Empty the partitioned set for all $q$}
        for all $q \in neighbor(p)$
            $reachable_p[q] \leftarrow \emptyset$ {Empty the reachable set on self-disconnection}
        notify new view $reachable_p$          {Possible new view with $p$ only}
|| task 3 : upon reconnection notification of $q$
    if $q \neq p$
        $partition_p[q] \leftarrow \emptyset$          {Empty the partitioned set for $q$}
        $discon_p \leftarrow discon_p \setminus \{q\}$          {Remove $q$ from $discon_p$}
        if $q \in neighbor(p)$
            $reachable_p[q] \leftarrow \{q\}$          {Initialize if $p \in neighbor$}
    else
        for all $q \in neighbor(p)$          {given at configuration time}
            $reachable_p[q] \leftarrow \{q\}$
            $partitioned_p[q] \leftarrow \emptyset$
    notify new view $reachable(p)$
|| task 4 : upon $receive(pid, q, r, part)$ from $s$
    if $recv\_pid \neq pid$
        for all $t \in part$ such that $t \notin faulty_p$
            $partition_p[r] \leftarrow partition_p[r] \cup \{t\}$   {$faulty_p \cap partition_p = \emptyset$}
            $reachable_p[s] \leftarrow reachable_p[s] \setminus \{t\}$    {Remove from $reachable_p$ of sending process}
        for all $t$ such that $t \in neighbor(p)$
            qr_neighbor_send$(pid, q, r, part)$
        $recv\_pid \leftarrow pid$
coend
```

Figure A.2: Partition Detector with Neighborhood Topology $\mathcal{PDN}$* (Ctn'd)

## A.6 Optimizing Failure Detection

Failure detection is a service which is necessary for a distributed system. But at the very same time its cost should not be too high, that is the failure detection should detect failures with as much efficiency as possible. There are various techniques, we think, can ameliorate the detection of failures in a distributed system. We present them in the following sections.

Reachability and neighborhood information can be exchanged as route information is exchanged by Internet routers in order to optimize the route information. One of the advantages that can be obtained from is by exchanging the neighbors information in order to avoid sending partition messages to all the processes. Every process $p$ may know the neighbors of each process. In case of a disconnection or a failure of a process, $p$ can infer the partitioned processes from reachability information combined with neighborhood information.

One way to rectify the problems of failure detection in large-scale environments is to divide the network in different groups based on the underlying topology. As seen in Section **??**, these groups are, however, built on the knowledge of network topology in a way that the topology information is already known. The idea we present here, although only theory, suggests the use of topology information at the runtime in order to adapt the failure detection. This can be a good idea in wireless networks where the topology is not known at the beginning and where reducing the number of messages in the system consumes lesser battery and network resources, to mention a few. Hierarchical failure detection should be designed in such a way that the processes with most number of neighbors should be designated as group leaders. For example in Figure 3.6, the failure detection can be mended, after the first topology discovering message, such that processes $u$, $v$ and $w$ should be within the local scope of $p$, and $r$, $s$ and $t$ in the local scope of $q$. Processes $p$ and $q$, which make the global scope, only exchange their heartbeats or the heartbeats of the all the members of the group in one message to decrease the message complexity. The local and global scopes may be inferred by analyzing the reachability information at each process.