Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Failure Detection and Recovery using Consensus algorithms in a Distributed Resource Management Framework

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΔΗΜΟΣΘΕΝΗ Γ. ΜΑΣΟΥΡΟΥ

Επιβλέπων : Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εργαστηριο Μικροϋπολογιστων & Ψηφιακων Συστηματων

Αθήνα, Ιούλιος 2016

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστηρίο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Failure Detection and Recovery using Consensus algorithms in a Distributed Resource Management Framework

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΔΗΜΟΣΘΕΝΗ Γ. ΜΑΣΟΥΡΟΥ

Επιβλέπων : Δημήτριος Σούντρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Ιουλίου 2016.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Δημήτριος Σούντρης	Κιαμάλ Πεκμεστζή	Γιώργος Οικονομάκος
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2016

(Υπογραφή)

.....

ΔΗΜΟΣΘΕΝΗΣ ΜΑΣΟΥΡΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστηρίο Μικροϋπολογιστών & Ψηφιακών Συστημάτων

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον κ. Δημήτριο Σούντρη για την υπομονή του και την εμπιστοσύνη που μου έδειξε καθ΄ όλη τη διάρκεια εκπόνησης της διπλωματικής μου εργασίας, καθώς επίσης και τον επιβλέποντα και φίλο Βασίλη Τσούτσουρα για την επίβλεψη της διπλωματικής μου εργασίας, την παραχώρηση του κώδικα του για την υλοποίηση πάνω σε αυτόν της εργασίας μου και για την αμέριστη βοήθεια του. Επιπλέον, θα ήθελα να ευχαριστήσω τους φίλους μου Γιώργο και Αθηνά για τη στήριξη αλλά και τη βοήθεια τους σε κρίσιμα σημεία. Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου για την στήριξη που μου προσέφερε όλα αυτά τα χρόνια.

Περίληψη

Η παρούσα διπλωματική εργασία επικεντρώνεται στην ανίχνευση αδιεξόδων και σφαλμάτων καθώς και στην ανάνηψη σε περίπτωση που αυτά συμβούν σε κάποιο Πολυ-Πύρηνο Συστήμα σε Ψηφίδα. Συγκεκριμένα, εξετάζουμε συστήματα τα χρησιμοποιούν αρχιτεκτονική Δικτύου-σε-Ψηφίδα. Η τεχνολογία αυτή παρουσιάζει πολλές ομοιότητες με την ιδέα των κατανεμημένων συστημάτων κυρίως στον τρόπο επικοινωνίας και της ιδέας κατανομής πόρων. Για το λόγο αυτό, στόχος της παρούσας διπλωματικής είναι η υλοποίηση γνωστών αλγορίθμων στον τομέα των κατανεμημένων συστημάτων σε ένα πλαίσιο κατανομής πόρων το οποίο διαχειρίζεται εφαρμογές σε ένα σύστημα σε ψηφίδα.

Στο κεφάλαιο 1, κάνουμε μία εισαγωγή πάνω στα κατανεμημένα συστήματα, στα συστήματα με αρχιτεκτονική δικτύου ψηφίδας καθώς και τις έννοιες της αξιοπιστίας, ανοχής σε σφάλματα και ομοφωνίας.

Στο κεφάλαιο 2, παρουσιάζουμε εργασίες και υλοποιήσεις τεχνολογιών οι οποίες επικεντρώνονται στην ανάνηψη συστημάτων από σφάλματα.

Στο κεφάλαιο 3, 4, 5 και 6 αναλύουμε τους τρόπους επικοινωνίας και τα είδη σφαλμάτων σε κατανεμημένα συστήματα καθώς τους τρόπους με τους οποίους μπορούμε να εντοπίσουμε αδιέξοδα και σφάλματα. Επίσης, παρουσιάζουμε ένα πλαίσιο κατανομής πόρων, γνωστό ως DRTRM, πάνω στο οποίο θα ενσωματώσουμε τους αλγορίθμους για εντοπισμό σφαλμάτων και αδιεξόδων καθώς και το πρωτόκολλο ανάνηψης σε περίπτωση σφάλματος PAXOS.

Στο κεφάλαιο 7, αναλύουμε τη διαδικασία την οποία ακολουθήσαμε προκειμένου να ενσωματώσουμε τον PAXO καθώς και τους ανιχνευτές σφαλμάτων στο πλαίσιο κατανομής πόρων.

Στο κεφάλαιο 8, εξετάζουμε διαφορετικά σενάρια σφαλμάτων και παρουσιάζουμε τα πειραματικά αποτελέσματα.

το κεφάλαιο 9 συνοψίζουμε τα συμπεράσματά μας και προτείνουμε ιδέες για μελλοντική έρευνα.

Λέξεις Κλειδιά

Σύστημα-σε-Ψηφίδα, Πολυ-Πύρηνο Δίκτυο σε Ψηφίδα, Εντοπισμός Σφαλμάτων, Εντοπισμός Αδιεξόδων, Ομοφωνία, PAXOS, DRTRM

Abstract

This diploma thesis focuses on deadlock and failure detection as well as recovery in case of failure on a Multi-Processor System-on-Chip (MPSoC). More precisely, we examine systems which utilize the Network-on-Chip (NoC) architecture. These types of systems share many similarities with a distributed system, specifically in the communication scheme and the allocation of resources. Thus, we implemented some popular algorithms which appear in distributed systems, on top of a resource management framework that manages applications on a MPSoC.

In chapter 1, we introduce the class of distributed systems and systems with a NoC architecture and proceed with the basic concepts of reliability and consensus.

In chapter 2, we present published works and real-life implementations which focus on recovery after failure.

In chapters 3, 4, 5 and 6 we analyze the different communication methods and the types of failures that occur in distributed systems, as well as the ways in which we can detect failures and deadlocks. In addition, we present the DRTRM resource management framework, which was used to implement the deadlock and failure detection algorithms and the PAXOS protocol, which is used to recover in case of failure.

In chapter 7, we give detailed information on how we merged PAXOS and detectors with the DRTRM framework.

In chapter 8, we examine different failure scenarios and we present our theoretical and experimental results.

Lastly, in chapter 9 we summarize our conclusions and propose ways and ideas for future research.

Keywords

System-on-Chip, Multi-Processor System-on-Chip, Failure Detection, Deadlock Detection, Consensus, PAXOS, DRTRM

Εκτεταμένη Περίληψη

Εισαγωγή

Σύμφωνα με το νόμο του Moore στην πιο απλοϊκή του μορφή γνωρίζουμε πως η υπολογιστική ισχύς ενός επεξεργαστή διπλασιάζεται κάθε περίπου δύο χρόνια. Κανείς θα περίμενε πως ένας τέτοιος ρυθμός σχεδιαστικής ανάπτυξης θα αρκούσε για τις υπολογιστικές ανάγκες που έχουμε σήμερα. Παρόλα αυτά, δε συμβαίνει κάτι τέτοιο τόσο διότι ο νόμος του Moore έχει φτάσει σε ένα τέλμα καθώς και επειδή οι απαιτήσεις των χρηστών για υπολογιστική ισχύ αλλά και αποθήκευση δεδομένων αυξάνεται ολοένα και περισσότερο. Μία πρώτη λύση στο παραπάνω πρόβλημα ήταν η εμφάνιση πολυπύρηνων συστημάτων (Multicore Systems). Ένα πρόβλημα που παρουσιάζουν τα εν λόγω συστήματα είναι πως ο ρυθμός ανάπτυξης των δεδομένων προς επεξεργασία καθώς και ο αριθμός των χρηστών που ζητούν δεδομένα είναι κατά πολύ μεγαλύτερος από αυτά που μπορεί να επεξεργαστεί ένα πολυπύρηνο σύστημα. Η λύση στο πρόβλημα αυτό ήταν η εμφάνιση της έννοιας του κατανεμημένου συστήματος (Distributed System). Η κύρια ιδέα πίσω από τα συστήματα αυτά είναι πως αφού η υπολογιστική δύναμη ενός μονάχα μηχανήματος δεν αρκεί για τον υπολογισμό του τεράστιου αριθμού δεδομένων που χρειαζόμαστε σήμερα, μπορούμε να διαμοιράσουμε την επεξεργασία τους σε διαφορετικές επεξεργαστικές μονάδες, κάθε μία από τις οποίες θα εκτελεί το δικό της κομμάτι και στη συνέχεια να επιστρέφουμε το αποτέλεσμα ολοκληρωμένο στο χρήστη. Ένα επιπλέον πρόβλημα των πολυπύρηνων συστημάτων είναι πως λόγω του τεράστιου όγκου δεδομένων οι ανάγκες επικοινωνίας μεταξύ των πυρήνων δεν μπορούν να διενεργηθούν από τις κλασσικές τεχνικές επικοινωνίας με διαύλους δεδομένων (buses) και έτσι οδηγούμαστε σε ένα βοττλενεχ. Για το λόγο αυτό, οι σχεδιαστές ψηφίδων οδηγήθηκαν στη δημιουργία συστημάτων σε ψηφίδα (Network on Chip - NoC), τα οποία αποτελούν πολυπύρηνα συστήματα με αρχές επικοινωνίας που βασίζονται σε δίκτυα υπολογιστών. Τέλος, αρκετές φορές οι διαφορετικές διεργασίες πρέπει να συμφωνήσουν σε κάποιες τιμές δεδομένων τα οποία χρειάζονται κατά τη διάρκεια της εκτέλεσης, όπως για παράδειγμα εκλογή αρχηγού (leader election) ή αντιγραφή κάποιου μηχανήματος καταστάσεων (state machine replication). Επιτακτική λοιπόν ανάγκη για τη σωστή λειτουργία και την αξιοπιστία (reliability) των κατανεμημένων συστημάτων είναι

αυτά να βρίσκονται σε ομοφωνία (consensus).

Κατανεμημένα Συστήματα

Ένα κατανεμημένο σύστημα ουσιαστικά είναι ένα σύνολο υπολογιστών οι οποίοι συνδέονται μεταξύ τους σε δίκτυο, οι οποίοι όμως εμφανίζονται στον τελικό χρήστη σαν ένα σύστημα. Όπως εύκολα καταλαβαίνουμε ένα κατανεμημένο σύστημα μπορεί να προσφέρει πολύ μεγάλη υπολογιστική δύναμη καθώς μία εφαρμογή μπορεί να διαχωριστεί σε πολλές επιμέρους, να διαμοιραστεί σε διαφορετικούς κόμβους, οι οποίοι να παράξουν παράλληλα πολύ γρήγορα το αποτέλεσμα.

Στα συστήματα αυτά οι επιμέρους κόμβοι επικοινωνούν μεταξύ τους ανταλλάσοντας μηνύματα μέσω του δικτύου. Τα βασικά χαρακτηριστικά των κατανεμημένων συστημάτων είναι τα εξής:

- **Συγχρονισμός κόμβων:** Όλοι οι επιμέρους υπολογιστές του κατανεμημένου συστήματος πρέπει να είναι θεωρητικά συγχρονισμένοι μεταξύ τους, εννοώντας ότι κάθε στιγμή θα πρέπει να συντονίζονται οι ενέργειες των επιμέρους κόμβων προκειμένου να παραχθεί κάποιο αποτέλεσμα.

- Το παραπάνω θα πρέπει να επιτευχθεί **χωρίς την παρουσία κάποιου καθολικού ρολογιού**. Δηλαδή, ο συγχρονισμός των κόμβων επιτυγχάνεται πλήρως μέσω της ανταλλαγής μηνυμάτων μεταξύ τους.

- Τέλος, κάθε επιμέρους κόμβος του συστήματος μπορεί να **αποτύχει ανεξάρτητα** από τι συμβαίνει με τους υπόλοιπους.

Όπως καταλαβαίνουμε, όταν συμβεί κάποια αποτυχία (είτε υλικού, είτε αποτυχία επικοινωνίας) σε κάποιον επιμέρους κόμβο, θα πρέπει το συνολικό κατανεμημένο σύστημα να συνεχίζει να λειτουργεί ομαλά και ο τελικός χρήστης να μην αντιλαμβάνεται την αποτυχία αυτή. Είναι λοιπόν επιτακτική ανάγκη να εφαρμόσουμε τεχνικές προκειμένου να επιτύχουμε ανοχή σε σφάλματα.

Ανοχή σε Σφάλματα

Όπως είπαμε και προηγουμένως, ένα σύστημα (τόσο κατανεμημένο αλλά και γενικότερα) θα πρέπει να συνεχίσει να λειτουργεί σε περίπτωση που συμβεί κάποια αποτυχία. Για να καταφέρουμε όμως να επιτύχουμε ανοχή σε κάποιο σφάλμα, θα πρέπει να είμαστε αρχικά σε θέση να εντοπίσουμε πως κάποιο επιμέρους στοιχείο απέτυχε. Το ερώτημα είναι πως μπορούμε να εντοπίσουμε κάποιο τέτοιο σφάλμα· Την απάντηση στο ερώτημα αυτό δίνουν οι Chandra και Toueg το 1996, με τους ανιχνευτές σφαλμάτων. Η πρόβλεψη τέτοιων ανιχνευτών δεν είναι

απαραίτητο να είναι πάντα σωστή, αλλά και σε περίπτωση λανθασμένης πρόβλεψης η ανίχνευση πιθανού σφάλματος είναι πολύ σημαντική. Γενικά, οι ανιχνευτές σφάλματος χωρίζονται σε 8 κατηγορίες οι οποίες προκύπτουν με βάση τα παρακάτω:

- **Πληρότητα:** Υποψία των αποτυχημένων κόμβων.
- **Ακρίβεια:** Υποψία των μη αποτυχημένων κόμβων.

Τα δύο παραπάνω χαρακτηριστικά, χωρίζονται σε δύο επιπλέον κατηγορίες ανάλογα με τη διάδοση της πληροφορίας στους κόμβους του συστήματος. Όσον αφορά λοιπόν την *πληρότητα*, έχουμε τον επιπλέον διαχωρισμό της σε:

◇ **Ισχυρή Πληρότητα**: *Κάθε* αποτυχημένος κόμβος ανιχνεύεται από *κάθε* μη αποτυχημένο.

◇ **Ασθενής Πληρότητα**: *Κάθε* αποτυχημένος κόμβος ανιχνεύεται από *μερικούς* μη αποτυχημένους.

Αντίστοιχα, όσον αφορά την *Ακρίβεια*, έχουμε:

◇ **Ισχυρή Ακρίβεια**: *Κανένας* κόμβος δεν ανιχνέυεται σαν αποτυχημένος, πριν όντως να έχει αποτύχει.

◇ **Ασθενής Ακρίβεια**: *Μερικοί* μη αποτυχημένοι κόμβοι δεν ανιχνεύονται ποτέ.

◇ **Ενδεχομένως Ισχυρή Ακρίβεια**: Έπειτα από κάποια αρχική *περίοδο σύγχησης*, *κανένας* κόμβος δεν ανιχνέυεται σαν αποτυχημένος, πριν όντως να έχει αποτύχει.

◇ **Ενδεχομένως Ισχυρή Ακρίβεια**: Έπειτα από κάποια αρχική *περίοδο σύγχησης*, *μερικοί* μη αποτυχημένοι κόμβοι δεν ανιχνεύονται ποτέ.

Με βάση τα παραπάνω λοιπόν, οι ανιχνευτές σφαλμάτων χωρίζονται σε 8 κατηγορίες όπως αναφέραμε και παραπάνω και οι οποίες φαίνονται στον παρακάτω πίνακα:

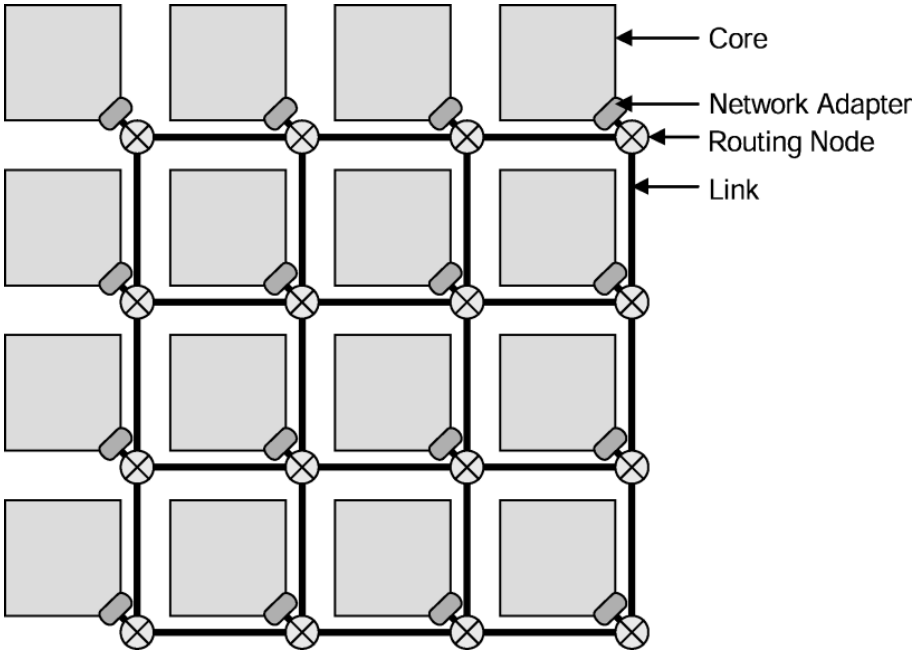
Κατηγορίες ανιχνευτών σφαλμάτων

Πληρότητα	**Ακρίβεια**			
	Ισχυρή	Ασθενής	Ενδεχόμενη Ισχυρή	Ενδεχόμενη Ασθενής
Ισχυρή	*Τέλειος* $\mathcal{P}$	*Ισχυρός* $\mathcal{S}$	*Ενδεχομένως Τέλειος* $\Diamond\mathcal{P}$	*Ενδεχομένως Ισχυρός* $\Diamond\mathcal{S}$
Ασθενής	$\mathcal{Q}$	*Ασθενής* $\mathcal{W}$	$\Diamond\mathcal{Q}$	*Ενδεχομένως Ασθενής* $\Diamond\mathcal{W}$

Πολυεπεξεργαστικά Συστήματα σε Ψηφίδα

Τα πολυεπεξεργαστικά συστήματα σε ψηφίδα προέκυψαν από την ανάγκη για όλο και περισσότερους πυρήνες στο ίδιο ολοκληρωμένο σύστημα. Στα συστήματα αυτά, ένας μεγάλος αριθμός πυρήνων τοποθετούνται στον ίδιο σύστημα προκειμένου να προσφέρουν υψηλότερη επίδοση. Κατά τη διάρκεια του σχεδιασμού αυτών των συστημάτων όμως, προέκυψε το ακόλουθο πρόβλημα: η κλασσική αρχιτεκτονική διαύλου δεν κάλυπτε τις ανάγκες για τον τεράστιο αριθμό πληροφορίας που χρειαζόταν να ανταλλαχθεί μεταξύ των πυρήνων. Το πρόβλημα αυτό ξεπεράστηκε με την επινόηση του Δικτύου σε Ψηφίδα. Στην αρχιτεκτονική Δικτύου σε Ψηφίδα ουσιαστικά οι πυρήνες επικοινωνούν μεταξύ ανταλλάσοντας μηνύματα, χρησιμοποιώντας τεχνικές και πρωτόκολλα εμπνευσμένα από τα Δίκτυα Υπολογιστών. Πιο συγκεκριμένα:

- Οι *πυρήνες* είναι Intellectual Property (IP) οντότητες οποιουδήποτε τύπου με κάποια τοπική μνήμη σε κάθε έναν από αυτούς.
- *Προσαρμογείς Δικτύου* χρησιμοποιούνται προκειμένου να συνδέσουν τους πυρήνες στο Δίκτυο σε Ψηφίδα.
- Οι *κόμβοι δρομολόγησης* είναι παρόμοιοι με τους δρομολογητές στα δίκτυα υπολογιστών. Είναι υπεύθυνοι για την εφαρμογή των σωστών πρωτοκόλλων δρομολόγησης στην πλατφόρμα.



Παράδειγμα ενός 4 επί 4 συστήματος αρχιτεκτονικής Δικτύου σε Ψηφίδα[3].

- Τέλος, *σύνδεσμοι* χρησιμοποιούνται για τη σύνδεση των πυρήνων μεταξύ τους, οι οποίοι παρεχουν ένα μέσο επικοινωνίας μεταξύ τους.

Συνεπώς, όπως είναι αναμενόμενο τα συστήματα αυτά παρουσιάζουν πολλές ομοιότητες με τα κατανεμημένα συστήματα, κυρίως στον τρόπο με τον οποίο οι διάφοροι πυρήνες επικοινωνούν μεταξύ τους.

Για τον διαμοιρασμό εργασιών σε τέτοια συστήματα, έχουν προταθεί κατά καιρούς διάφορα πλαίσια κατανομής πόρων, στα οποία οι πυρήνες λαμβάνουν διαφορετικούς ρόλους για την επίτευξη της σωστής διαχείρησης των διεργασιών. Άρα, θα πρέπει σε αντιστοιχία με τα κατανεμημένα συστήματα, έτσι και εδώ, να είμαστε σε θέση σε τέτοια πλαίσια να έχουμε ανοχή σε περίπτωση που κάποιος από τους πυρήνες αυτούς αποτύχει.

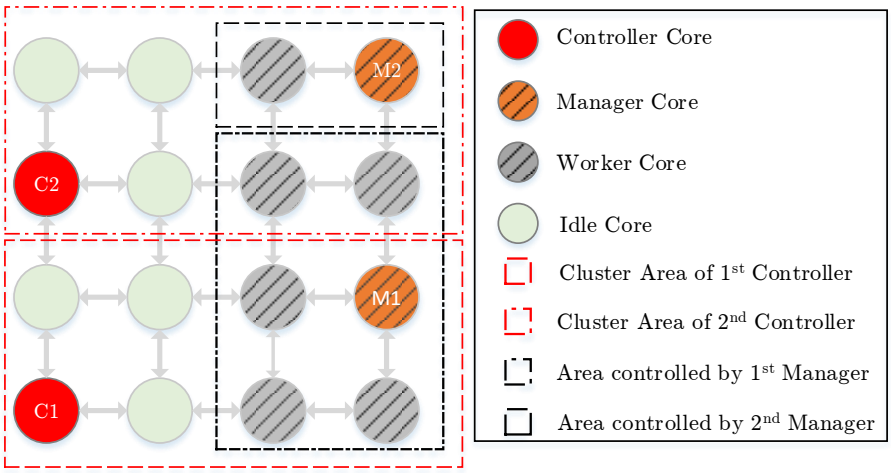
Το DRTRM πλαίσιο κατανομής πόρων

Το πλαίσιο DRTRM προτάθηκε το 2013 [16] και χρησιμοποιείται για τη διαχείρηση εύπλαστων εφαρμογών σε πολυ-επεξεργαστικά συστήματα σε ψηφίδα. Το πλαίσιο είναι υπεύθυνο για την ανάθεση αρχικών ρόλων σε διαφορετικούς πυρήνες αλλά και την αρχική κατανομή των διεργασιών στους πυρήνες καθώς και για την διαχείριση των εφαρμογών σε ολόκληρο τον κύκλο ζωής τους. Στόχος του πλαισίου αυτού είναι η ελαχιστοποίηση των μηνυμάτων που ανταλλάσονται μεταξύ των πυρήνων. Για να επιτευχθεί αυτό οι πυρήνες λαμβάνουν έναν ή περισσότερους από τους παρακάτω ρόλους:

- *Initial*: Ο συγκεκριμένος ρόλος πυρήνα είναι υπεύθυνος για την εύρεση κάποιων πυρήνων στους οποίους θα τρέξει αρχικά η εφαρμογή.
- *Controller*: Αυτός ο τύπος πυρήνα ουσιαστικά διαχειρίζεται όλους τους idle πυρήνες μέσα σε μία προκαθορισμένη περιοχή η οποία ονομάζεται *σύμπλεγμα*.
- *Manager*: Διαχειρίζεται μία εφαρμογή σε όλο τον κύκλο ζωής της. Η σχέση μεταξύ ενός manager και μιας εφαρμογής είναι 1 προς 1, δηλαδή ένας manager διαχειρίζεται μόνο μία εφαρμογή κάθε χρονική στιγμή, καθώς και μία εφαρμογή μπορεί να ανήκει σε έναν και μόνο manager.
- *Worker*: Ο τύπος αυτός πυρήνα εκτελεί οποιοδήποτε φόρτο εργασίας του σταλθεί από τον manager του.
- *Idle*: Τέλος, ο idle παραμένει άεργος έως ότου του ανατεθεί κάποιος ρόλος από τους παραπάνω.

Όπως αναφέραμε και παραπάνω το *σύμπλεγμα* είναι μία προκαθορισμένη περιοχή η οποία δε μπορεί να αλλάξει κατά τη διάρκεια εκτέλεσης. Επίσης, κάθε manager πυρήνας λέμε πως διαχειρίζεται μία περιοχή της πλατφόρμας, η οποία ουσιαστικά συνίσταται από τον εαυτό του

και τους workers του. Στο παρακάτω σχήμα βλέπουμε ένα παράδειγμα με δύο controllers και δύο managers μαζί με τις περιοχές που διαχειρίζονται



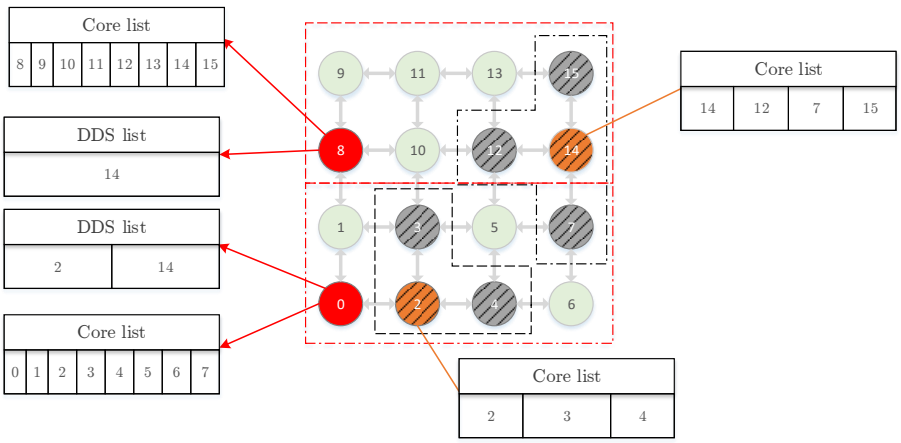
Παράδειγμα συμπλέγματος και περιοχής διαχειριζόμενης από manager

Μεγίστης σημασίας επίσης, για τη σωστή λειτουργία του DRTRM είναι οι λίστες που κρατάει κάθε είδος πυρήνα. Πιο συγκεκριμένα, κάθε controller πυρήνας κρατάει τις δύο παρακάτω λίστες:

- *DDS List*: Λίστα με όλους τους managers οι οποίοι έχουν workers μέσα στο σύμπλεγμα του controller.
- *Core List*: Λίστα με όλους τους πυρήνες μέσα στο σύμπλεγμα μου

Αντίστοιχα ο manager κρατάει:

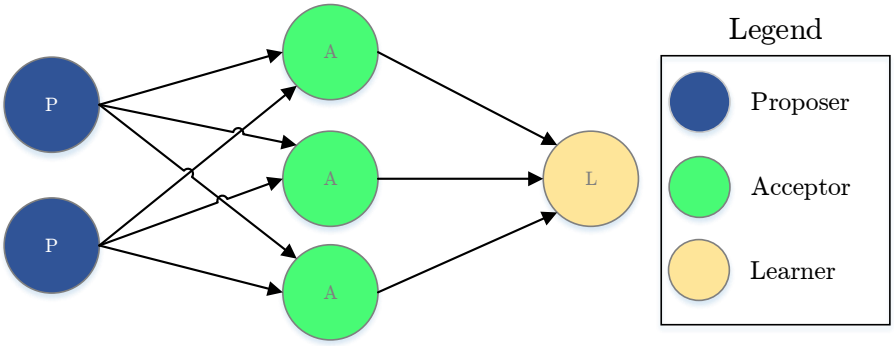
- *Core List*: Λίστα με όλους τους workers μου.



Παράδειγμα core list και DDS List για manager και controller.

Το πρωτόκολλο PAXOS

Το πρωτόκολλο ΠΑΞΟΣ προτάθηκε τη δεκαετία του 80 από τον Leslie Lamport [19][6] και χρησιμοποιείται για επίτευξη ομοφωνίας μεταξύ διαφορετικών κόμβων. Πιο συγκεκριμένα, στόχος του πρωτοκόλλου αυτού είναι σε περίπτωση που διαφορετικοί κόμβοι προτείνουν διαφορετικές τιμές, εν τέλει όλοι μαζί να καταλήξουν και να συμφωνήσουν σε μία από αυτές τις τιμές. Όπως είναι λογικό, όταν κάποιος πυρήνας αποτύχει σε ένα πολυεπεξεργαστικό σύστημα σε ψηφίδα, ο οποίος κατείχε κάποιον σημαντικό ρόλο του πλαισίου DRTRM (controller, manager, worker) τότε θα πρέπει να επιλέξουμε κάποιον καινούριο πυρήνα προκειμένου να εξυπηρετήσει το ρόλο αυτό. Στη διαδικασία αυτή μας βοηθάει το πρωτόκολλο PAXOS, δηλαδή στο πως θα συμφωνήσουν όλοι οι πυρήνες ποιος θα είναι ο καινούριος controller ή manager.



Αφηρημένη λειτουργία του πρωτοκόλλου PAXOS

Επομένως, στο συγκεκριμένο πρωτόκολλο υπάρχουν τριών ειδών ρόλοι (ανεξάρτητοι των controller, manager κτλ):

- *Proposer*: Ο συγκεκριμένος ρόλος προτείνει τιμές τις οποίες καλούντε να επιλέξουν οι acceptors.
- *Acceptor*: Αυτός ο τύπος πυρήνα ουσιαστικά αποδέχεται ή απορρίπτει τις τιμές που του προτείνει ο proposer.
- *Learner*: Μαθαίνει την επιλεχθείσα τιμή όταν αυτή επιλεγεί μέσω του PAXOS.

Επίσης, προκειμένου όλοι οι πυρήνες να συμφωνήσουν στην καινούρια τιμή χρειαζόμαστε 3 φάσεις:

- *Prepare*
- *Accept*
- *Learn*

Στην πρώτη φάση του πρωτοκόλλου, κάθε proposer επιλέγει έναν μοναδικό αριθμό n και στέλνει ένα σήμα Prepare_Request(n) σε όλους τους acceptors.

Φάση Prepare - Proposer

```
1: Pick unique proposal number pn
2: for i ← 1, k in A(j, k) do
3:     Send: prepare_request(i,pn)
4: end for
```

Ένας acceptor από την άλλη, όταν λάβει κάποιο μήνυμα Prepare_Request εξετάζει αν έχει λάβει παλιότερα κάποιο μήνυμα με μεγαλύτερο αριθμό n. Εάν ναι, τότε απορρίπτει το σήμα που του ήρθε. Αλλιώς, απαντάει στον proposer με την τιμή την οποία έχει δεχτεί καθώς και τον αντίστοιχο αριθμό n αυτής, εάν υπάρχει κάποια, στέλνοντας ένα σήμα Prepare_Accept.

Φάση Prepare - Acceptor

```
1: procedure PREPARE_REQUEST_HANDLER(n)
2:     max_pn ← highest proposed proposal number seen
3:     max_n ← highest accepted proposal number
4:     max_v ← Corresponding value of max_n
5:     if n > max_pn then
6:         max_pn ← n
7:         Reply: prepare_accept(max_n,max_v)
8:     else
9:         Reply: prepare_reject()
10:    end if
11: end procedure
```

Στη δεύτερη φάση του πρωτοκόλλου ο proposer εξετάζει εάν έλαβε απάντηση Prepare_Accept από την πλειοψηφία των acceptors καθώς και αποθηκεύει την τιμή με το μεγαλύτερο n που του έστειλε κάποιος acceptor (εάν υπήρχε κάποια τέτοια). Εάν έλαβε απάντηση από την πλειοψηφία, τότε στέλνει ένα σήμα Accept_Request(n,v) όπου n ο αριθμός που επέλεξε στην πρώτη φάση του πρωτοκόλλου και v η τιμή που του επιστράφηκε από κάποιον acceptor (εάν υπήρχε), αλλιώς κάποια τιμή που αυτός προτείνει.

Αντίστοιχα ο acceptor εξετάζει και πάλι αν έχει λάβει κάποιο σήμα με μεγαλύτερο n από αυτό του σήματος Accept_Request που έλαβε, και αν ναι απορρίπτει το αίτημα. Σε αντίθετη περίπτωση, αποδέχεται την τιμή που του προτείνει ο proposer, την αποθηκεύει και απαντάει με ένα σήμα Accepted.

Φάση Accept - Proposer

```
1: max_n ← 0
2: max_v ← 0
3: cnt ← 0
4: pn ← proposal number from prepare phase
5: procedure PREPARE_ACCEPT_HANDLER(n,v)
6:     cnt ← cnt + 1
7:     if n > max_n then
8:         max_n ← n
9:         max_v ← v
10:    end if
11:    if cnt ≥ k then
12:        if max_v = 0 then
13:            max_v ← proposer's proposing value
14:        end if
15:        for i ← 1, k in A(j, k) do
16:            Send: accept_request(i,pn,max_v)
17:        end for
18:    end if
19: end procedure
```

Φάση Accept - Acceptor Side

```
1: max_pn ← highest proposed proposal number seen
2: max_n ← highest accepted proposal number
3: max_v ← Corresponding value of max_n
4: procedure ACCEPT_REQUEST_HANDLER(n,v)
5:     if n > max_pn then
6:         max_pn ← n
7:         max_n ← n
8:         max_v ← v
9:         Reply: accepted()
10:    else
11:        Reply: rejected()
12:    end if
13: end procedure
```

Στην τελευταία φάση του πρωτοκόλλου, ο proposer πάλι εξετάζει αν έλαβε απάντηση Accepted από την πλειοψηφία των acceptors. Εάν ναι, τότε αντιλαμβάνεται πως η τιμή που

πρότεινε έχει επιλεχθεί και την ανακοινώνει σε όλους τους Learners.

Φάση Learn - Proposer Side

```
1: v ← Value proposed in accept_request (3).
2: cnt ← 0
3: procedure ACCEPTED_HANDLER
4:     cnt ← cnt + 1
5:     if cnt ≥ k then
6:         for each learner i in learners do
7:             Send: learn(i,v)
8:         end for
9:     end if
10: end procedure
```

Επέκταση DRTRM για ανίχνευση σφαλμάτων

Όπως είπαμε και προηγουμένως, θα πρέπει κάθε στιγμή να είμαστε σε θέση να εντοπίζουμε εάν κάποιος πυρήνας έχει αποτύχει στην πλατφόρμα και να επιλέγουμε νέους πυρήνες για να αντικαταστήσουν τον ρόλο του, εάν είναι απαραίτητο. Ίσως ο πιο απλός τρόπος για να εντοπίσουμε εάν κάποιος πυρήνας έχει αποτύχει είναι ο παρακάτω:

1. Θέτουμε ένα άνω όριο καθυστέρησης στην επικοινωνία μεταξύ δύο πυρήνων Δ.
2. Κάθε Δ δευτερόλεπτα στέλνουμε ένα σήμα Heartbeat_Request
3. Εάν δεν λάβουμε κάποια απάντηση Heartbeat_Reply στα επόμενα Δ δευτερόλεπτα, υποθέτουμε πως ο πυρήνας έχει αποτύχει

Perfect Failure Detector $\mathcal{P}$

```
1:  procedure PFD_INIT
2:      alive := Π
3:      detected := ∅
4:      starttimer(Δ)
5:  end procedure

6:  procedure TIMEOUT_HANDLER
7:      for each p in Π do
8:          if (p ∉ alive) ∧ (p ∉ detected) then
9:              detected := detected ∪ {p}
10:         end if
11:         Send: heartbeat_request(q,p)
12:     end for
13:     alive := ∅
14:     starttimer(Δ)
15: end procedure

16: procedure HEARTBEAT_REQUEST_HANDLER(q,p)
17:     Send: heartbeat_reply(p,q)
18: end procedure

19: procedure HEARTBEAT_REPLY_HANDLER(p,q)
20:     alive := alive ∪ {p}
21: end procedure
```

Με αυτόν τον τρόπο χρειαζόμαστε Δ δευτερόλεπτα για εντοπίσουμε εάν κάποιος πυρήνας έχει αποτύχει. Ονομάζουμε αυτόν τον τύπο ανιχνευτή Perfect Failure Detector (PFD).

Όπως παρατηρούμε ο παραπάνω ανιχνευτής απαιτεί μεγάλο κόστος ανίχνευσης, καθώς κάθε Δ δευτερόλεπτα χρειάζεται να σταλούν N^2 μηνύματα όπου N ο αριθμός των πυρήνων. Μία βελτιστοποίηση του παραπάνω είναι η εξής. Αφού, το πλαίσιο κατανομής πόρων ούτως

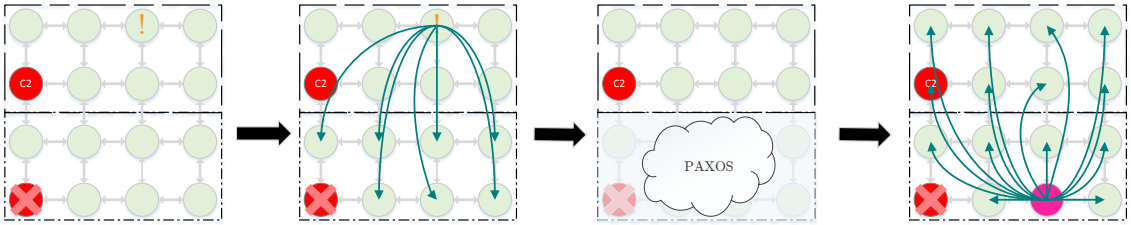
tweaked Perfect Failure Detector $\mathcal{P}_t$

```
1:  procedure TPFD_INIT
2:      alive := Π
3:      detected := ∅
4:      suspected := ∅
5:      starttimer(Δ)
6:  end procedure

7:  procedure TIMEOUT_HANDLER
8:      for each p in Π do
9:          if (p ∈ suspected then
10:             detected := detected ∪ {p}
11:         end if
12:         if (p ∉ alive) ∧ (p ∉ detected) ∧ (p ∉ suspected) then
13:             suspected := suspected ∪ {p}
14:             Send: heartbeat_request(q,p)
15:         end if
16:     end for
17:     alive := ∅
18:     starttimer(Δ)
19: end procedure

20: procedure HEARTBEAT_REQUEST_HANDLER(q,p)
21:     Send: heartbeat_reply(p,q)
22: end procedure

23: procedure ANY_SIGNAL_RECEIVED_HANDLER(p,q)
24:     alive := alive ∪ {p}
25:     if (p ∈ suspected) then
26:         suspected := suspected \ {p}
27:     end if
28: end procedure
```

ή άλλως ανταλλάσει μηνύματα για τη σωστή λειτουργία του, τότε δεν είναι απαραίτητο να στέλνουμε σήματα Heartbeat_Request, παρά μόνο εάν δε λάβαμε κάποιο σήμα του πλαισίου τα τελευταία Δ δευτερόλεπτα. Ενώ το κόστος ανίχνευσης με αυτήν την προσέγγιση μειώνεται κατα πολύ, ο χρόνος ο οποίος απαιτείται για την ανίχνευση ενός σφάλματος είναι $2 * \Delta$. Ονομάζουμε αυτόν τον τύπο ανιχνευτή tweaked Perfect Failure Detector (tPFD). Συνεπώς, πρέπει να αποφασίσουμε κατά τη διάρκεια υλοποίησης κάθε φορά και ανάλογα με το σκοπό της εφαρμογής που θα διαχειριστεί το πλαίσιο αν χρειαζόμαστε γρηγορότερη ανίχνευση σφαλμάτων ή μικρότερο αριθμό ανταλλαγής μηνυμάτων.

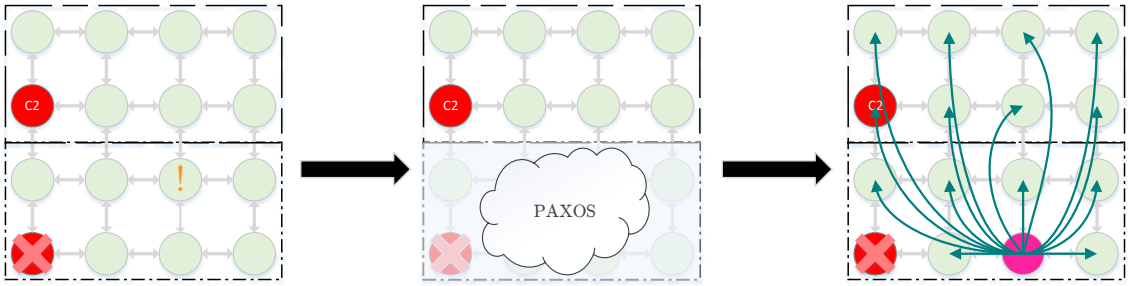
Ανάνηψη σε περίπτωση σφάλματος στο πλαίσιο DRTRM

Για την διαδικασία ανάνηψης έπειτα από κάποιο σφάλμα διακρίνουμε τρεις περιπτώσεις. Στην πρώτη περίπτωση ο πυρήνας που απέτυχε είχε τον ρόλο του controller. Σε αυτήν την περίπτωση, εάν κάποιος πυρήνας εκτός του συμπλέγματος του controller ανιχνεύσει την αποτυχία στέλνει ένα σήμα SIG_CONTR_TO σε όλους τους πυρήνες μέσα στο σύμπλεγμα, και στη συνέχεια εκκινείται μία διαδικασία του PAXOS όπως περιγράφθηκε προηγουμένως.



Πυρήνας εκτός συμπλέγματος ανιχνεύει την αποτυχία ενός controller

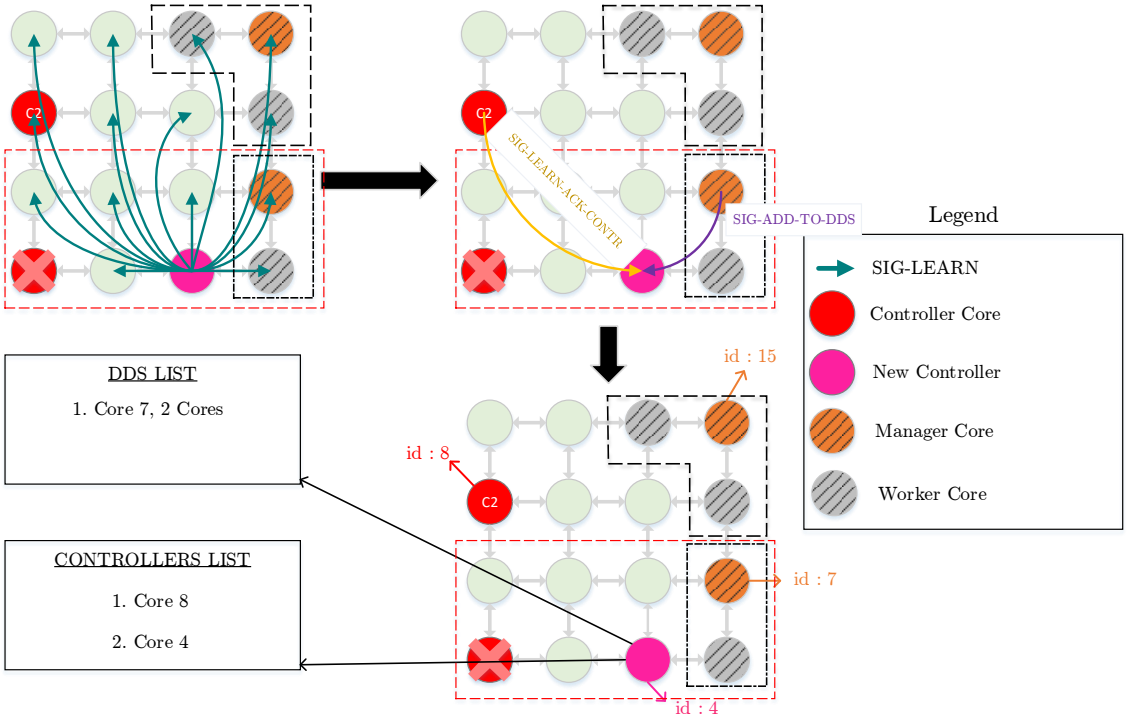
Αντίστοιχα, όταν κάποιος πυρήνας εντός του συμπλέγματος καταλάβει πως ο controller του έχει αποτύχει τότε ξεκινάει αυτός ο ίδιος μία διαδικασία PAXOS.



Πυρήνας εντός συμπλέγματος ανιχνεύει την αποτυχία ενός controller

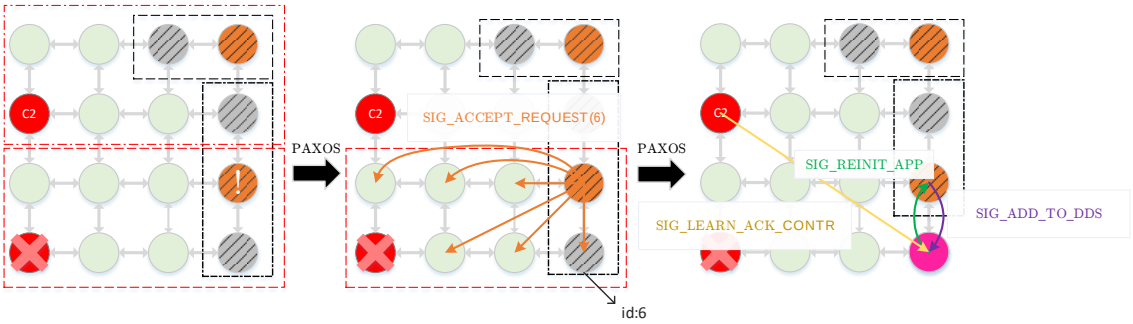
Στη συνέχεια αφού έχει εκλεγεί ο καινούριος controller πρέπει να πραγματοποιηθούν

κάποιες περαιτέρω ενέργειες προκειμένου να επιστρέψουμε σε σταθερότητα. Πιο συγκεκριμένα, ο καινούριος controller πρέπει να δημιουργήσει τις DDS-List και Core-List. Επίσης, όλοι οι υπόλοιποι controllers πρέπει να ειδοποιήσουν τον καινούριο προκειμένου να τους αποθηκεύσει και να είναι σε θέση να επικοινωνήσει μαζί τους μετέπειτα.



Ανάνηψη μετά τη διαδικασία PAXOS - Δημιουργία Λιστών

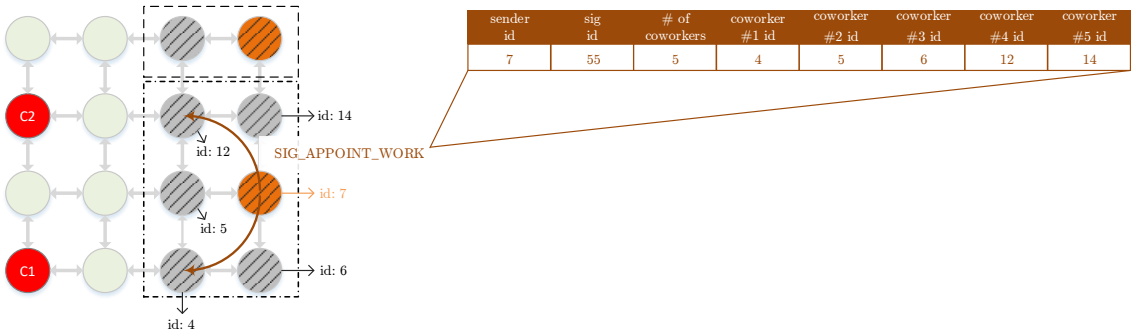
Εκτός από τη δημιουργία των λιστών, πρέπει επίσης να εξετάσουμε τι έργο επιτελούσε ο καινούριος controller προτού εκλεγεί. Ουσιαστικά, η μόνη περίπτωση που χρειάζεται να κάνουμε κάτι παραπάνω είναι όταν ο καινούριος controller ήταν worker. Σε αυτήν την περίπτωση ειδοποιούμε τον manager προκειμένου να στείλει το φόρτο εργασίας που εκτελούσαμε σε κάποιον άλλο πυρήνα. Στην περίπτωση που ο κάποιος manager καταλάβει ότι ο controller



Ανάνηψη μετά τη διαδικασία PAXOS - Επαναδιορισμός φόρτου εργασίας

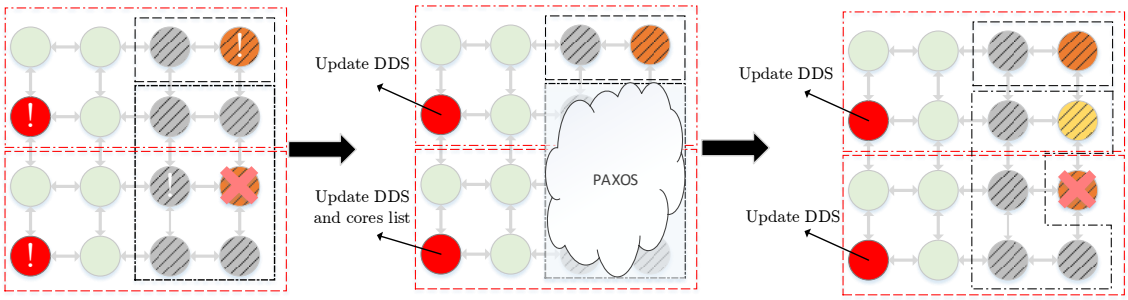
του έχει αποτύχει τότε στη δεύτερη φάση του πρωτοκόλλου PAXOS προτείνει κάποιον worker του για νέο controller.

Για την ανάνηψη έπειτα από αποτυχία κάποιου manager αντιμετώπιζαμε ένα πρόβλημα. Οι workers δεν γνωρίζαμε με ποιους πυρήνες δουλεύανε μαζί αφού είχε πέσει ο manager τους. Για το λόγο αυτό προσθέσαμε ένα νέο είδος λίστας την οποία ονομάζουμε coworkers-list. Στη λίστα αυτή ο κάθε worker αποθηκέυει τους συνεργάτες του, τους οποίους τους του στέλνει ο manager του κάθε φορά που του στέλνει κάποιο σήμα για να του αναθέσει κάποιο φόρτο εργασίας.



Αποστολή συνεργατών από manager σε worker μαζί με το φόρτο εργασίας

Όμοια με παραπάνω, η διαδικασία PAXOS εκτελείται μεταξύ των πυρήνων οι οποίοι ήταν workers του manager που απέτυχε. Επίσης, θα πρέπει και πάλι να δημιουργηθούν οι λίστες DDS και cores όπως και προηγουμένως.

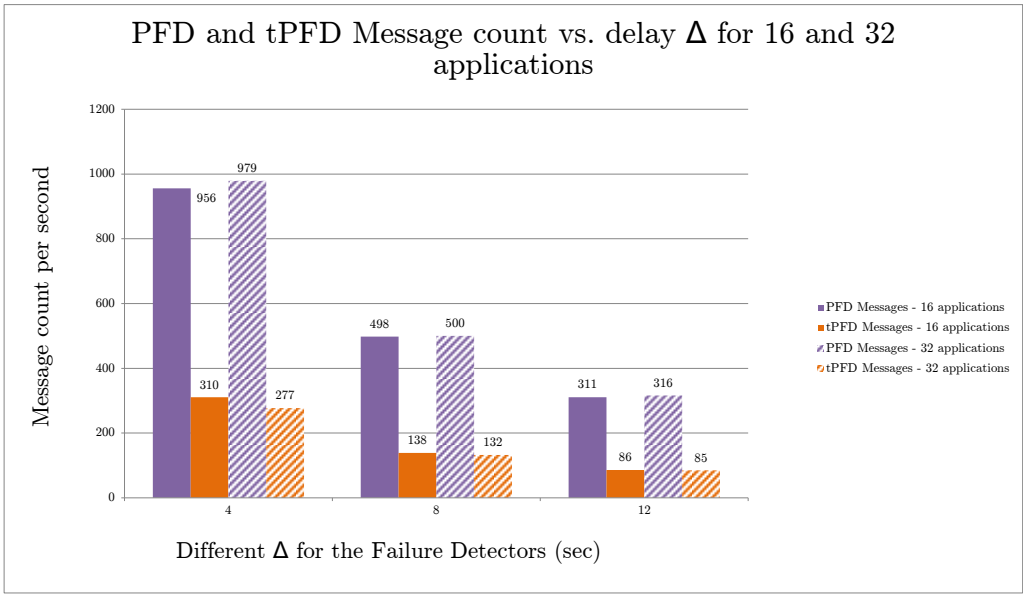


Ανάνηψη μετά τη διαδικασία PAXOS - Δημιουργία Λιστών

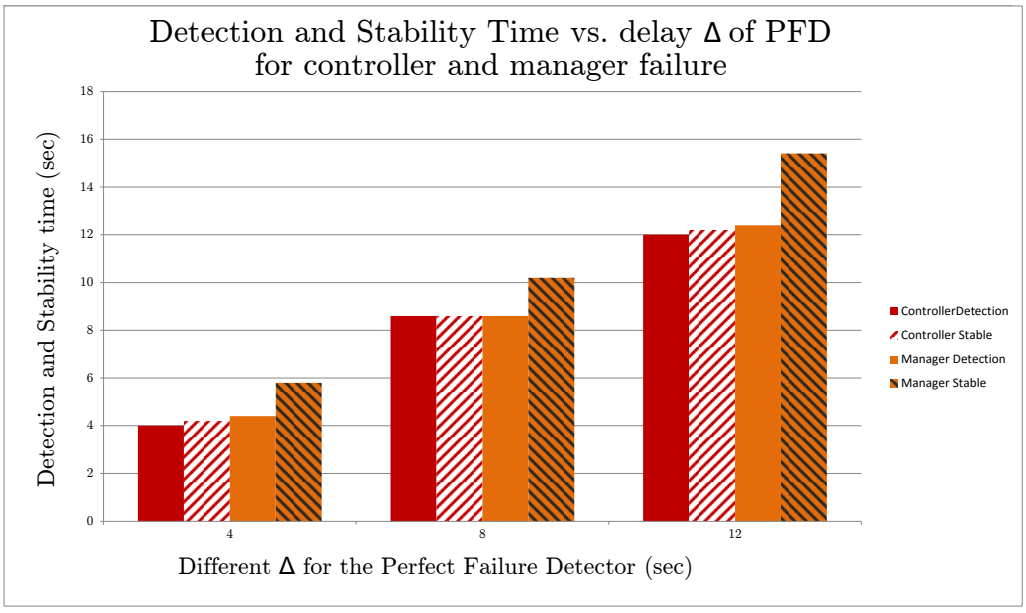
Τέλος, έαν αποτύχει κάποιος worker τότε ο αντίστοιχος manager κάποια στιγμή θα το ανιχνεύσει και το μόνο το οποίο πρέπει να γίνει είναι η ανακατανομή του φόρτου εργασίας σε κάποιον άλλο worker.

Πειραματικά Αποτελέσματα

Παρακάτω σε αυτή την ενότητα, παραθέτουμε μερικά πειραματικά αποτελέσματα τα οποία ουσιαστικά αποδεικύουν τις υποθέσεις μας. Πιο συγκεκριμένα, όπως είπαμε ο tPFD μειώνει κατά πολύ τον αριθμό μηνυμάτων, ο οποίος χρειάζεται προκειμένου να εντοπίσει κάποιο σφάλμα, αλλά χρειάζεται τον διπλάσιο χρόνο για την ανίχνευση.



Σύγκριση μεταξύ $\mathcal{P}$ και $\mathcal{P}_t$ - Αριθμός μηνυμάτων



Σύγκριση μεταξύ $\mathcal{P}$ και $\mathcal{P}_t$ - Καθυστέρηση Ανίχνευσης

Επιπλέον, στο παρακάτω διάγραμμα παρατηρούμε πως σε περίπτωση που αποτύχει κάποιος

ςοντρολλερ το μέγεθος του συμπλέγματος είναι αντιστρόφως ανάλογο του αριθμού των μηνυμάτων που ανταλλάσονται κατά τη διαδικασία PAXOS ενώ στην περίπτωση που αποτύχει ο μαναγερ ο αριθμός των μηνυμάτων παραμένει σχεδόν σταθερός.

Σύγκριση μεταξύ αποτυχίας controller και manager - Αριθμός μηνυμάτων

Contents

List of Figures

Chapter 1

Distributed Systems, Network on Chip and Consensus

According to Moore's Law in its simplest form we know that the processing power of CPU gains double for approximately every two years. One might expect that a design development rate such as this, would be more than enough to serve the computational needs we have nowadays. Nonetheless, that is not happening not only because Moore's law is approaching its close, but also because the demand of users in both processing power and data storage is getting extremely higher. A simple solution to the problem above could be to increase the CPU's frequency. However, the progressive increment of frequency would eventually cause overheating issues. A more general solution was the implementation of multi-core systems. Even though they served users' needs for data processing and data analysis at first sight, in the end the computational power was still not enough. A further solution to serve this huge amount of data, which is currently being used widely in the industry, was the development of distributed systems. The main idea behind these systems is that we have to distribute the data for computation to several computational units, since the power of a single one isn't enough to accomplish the task, collect the processed data and return them to the user. An additional issue of multi-core systems is that the communication between processors cannot be efficiently carried out by traditional buses without serious bottleneck issues, or point-to-point communication without serious space and energy waste. Thus, in order to overcome this obstacle, chip manufacturers led to the design of Network-On-Chip (NoC) architectures. Network on chip or network on a chip (NoC) is a communication subsystem on an integrated circuit, typically between intellectual property (IP) cores in a system on a chip (SoC). NoC technology applies networking theory and methods to on-chip communication and brings notable improvements over conventional bus and crossbar interconnections. NoC improves the scalability of SoCs, and the power efficiency of complex SoCs compared to other designs. In conclusion, a

fundamental problem in distributed computing is to achieve overall system reliability in the presence of faulty processes. This often requires processes to agree on some data value that is needed during computation. Such examples are leader election or state machine replication. So, an imperative need for the proper function and reliability of distributed systems is to achieve consensus between individual units.

1.1 Distributed Systems

A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages [1]. To a single user, these components appear as a single coherent system. Maybe the most known distributed system is the Internet itself. Except for that, many other contemporary applications make use of distributed systems varying from SOA-based-systems to MMO games and peer-to-peer applications. The components of a distributed system interact with each other in order to achieve a common goal.

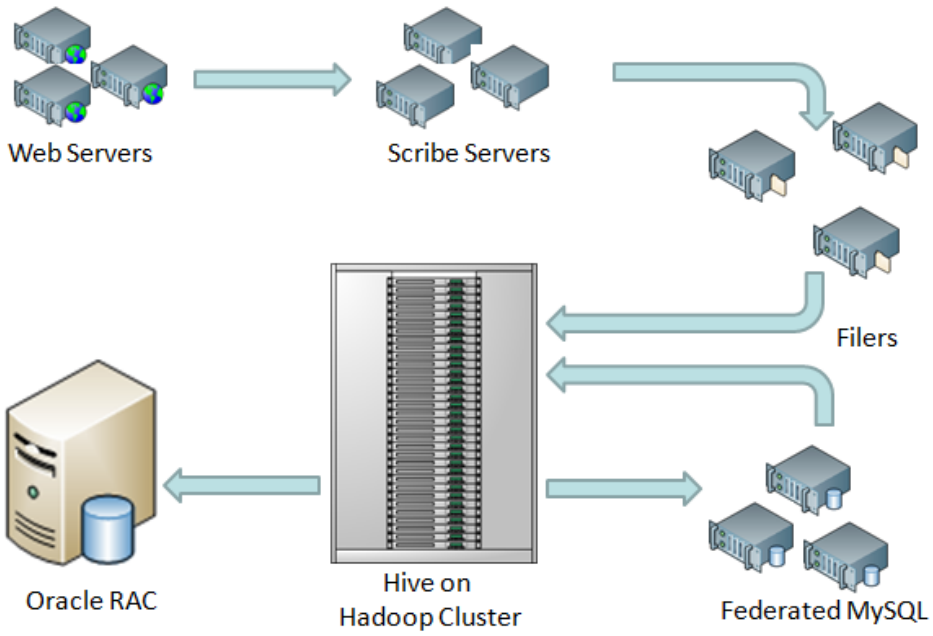


Figure 1.1: Facebook's system architecture [2]

Thus, when talking about distributed systems, we must keep in mind the following consequences:

- *Concurrency*: In distributed systems, components execute programs simultaneously. Therefore, we have to ensure that the coordination of concurrently executing programs is correct at any point.

- *No global clock*: As mentioned before, components communicate with each other only by exchanging messages. As a result, there is no single global notion of the correct time.
- *Independent failures*: Individual components might fail at any time or become extremely slow and be suspected as failed. System designers should look ahead for scenarios like these.

One might wonder, "Why do we need distributed systems?". The main answer is to cope with the extremely higher demand of users in both processing power and data storage. For example, according to *Data Center Knowledge*, Facebook has more than 500 millions users, 1 million photos are viewed every second and each month more than 3 billions photos are uploaded. With these extremely high demands, noone can believe that a single system could serve those. That's one reason why distributed systems comes in place. In figure 1.1 we see an example of Facebook's distributed system.

Distributed systems might appear as the perfect solution to many problems, however they also have some disadvantages. We mention some of both in the list below:

+ **Speed**: Obviously a system composed by several computers will be much faster than a single one in mean of execution time.
+ **Inherent Distribution**: Parallelization is the trend in new computing systems. Sharing workloads to several processes provides faster execution. Distributed systems provide this feature without any special need of hardware.
+ **Reliability**: A system composed by many components is more likely not only to output a correct result but also to actually output a result.
+ **Incremental growth**: Components can be added or removed from a distributed system. As a result, we can add more computing power any time we have to.
− **Software**: Most of the times, it is extremely difficult to design a software that not only ensures that all the components of the system work well together but also tolerates faults in the system.
− **Network**: Except for correctness of the software, we also have to ensure correct communication between components and overcome network issues.
− **More components to fail**: As the size of the system can infinitely grow, the probability that a component will fail rises.
− **Security**: Security is one of the most important parts to keep in mind while designing a distributed system. With the addition of more components in the system it more likely that a potential intruder might find a vulnerability in the system.

1.2 Networks On Chip

1.2.1 Overview

The idea of Networks on Chip arrived from the inability of traditional communication buses to carry out communication between modern processing elements. The first solution to this problem were distributed systems as explained in the previous section. As a consequence, designers decided to employ the same principles of distributed systems for on-chip communication. A NoC is generally a new approach to the System-On-Chip (SoC) model, in which computer networks' elements are used for on-chip communication and has the following characteristics:

- *Cores* are Intellectual Property (IP) blocks of any type (in practice they are usually different kinds of processors) with some local memory attached to them. They are also known as *tiles* of the NoC.
- *Network Adapters* are used to connect the cores to the NoC.
- *Routing Nodes* are similar to the routers in computer networks. They are responsible to apply the correct routing protocols in the platform.
- *Links* are used to connect the routing nodes and provide a communication mean between them.

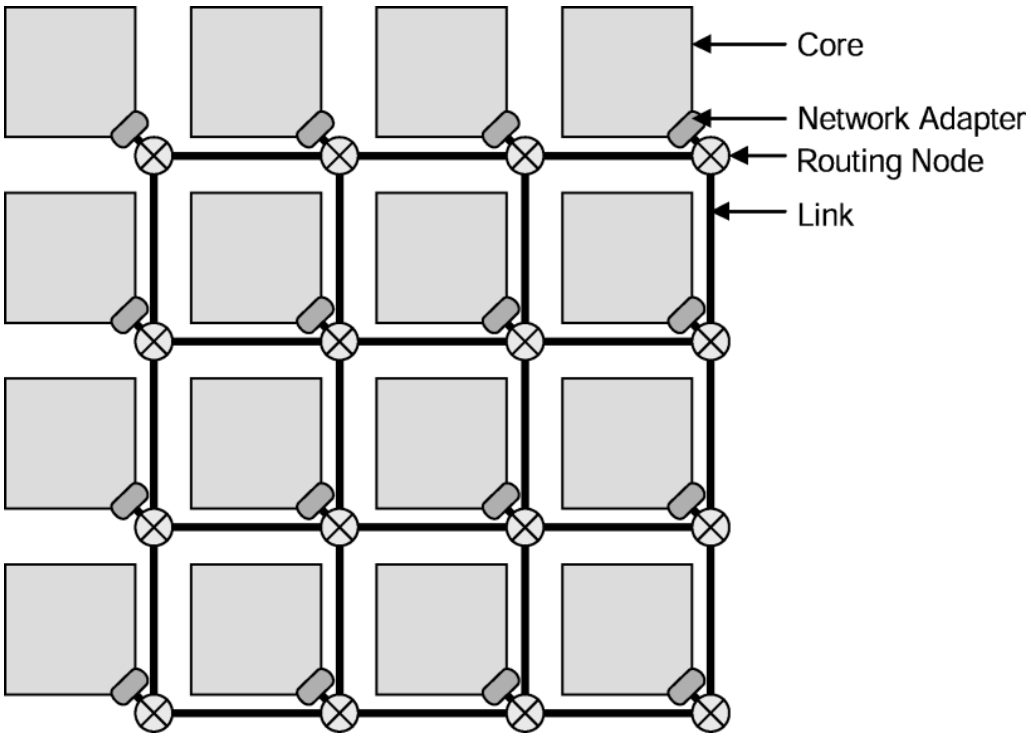


Figure 1.2: Example of a 4x4 Network on Chip topology[3].

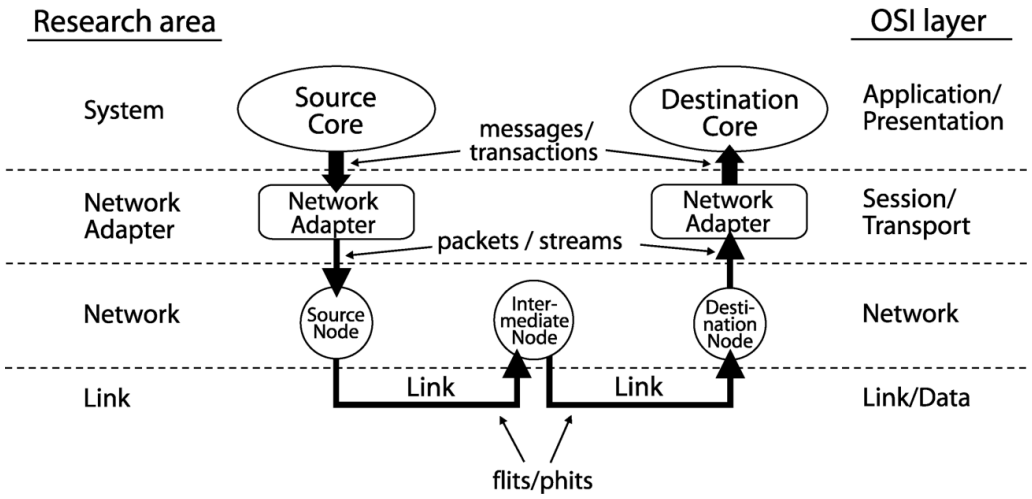


Figure 1.3: Communication between two cores in NoC[3].

1.2.2 Communication

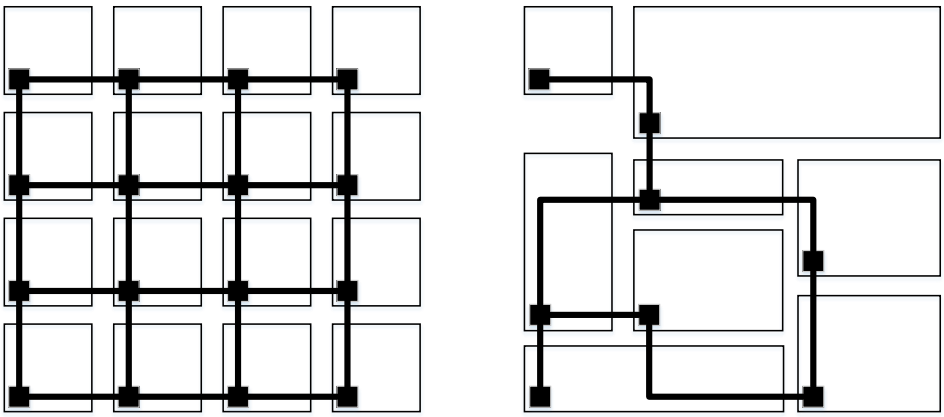
The communication of between cores is based on passing messages. Whenever a core wants to communicate with another, he sends the message to his network adapter. The latter decides the destination of the message because the core itself is not aware pf the platform. Subsequently, the network adapter forwards the message to the routing node, which is responsible to pass the message to the destination core, or an intermediate core if he has no available connection to the destination. Once the message is delivered to the destination core the routing node forwards it to the network adapter and the latter passes it to the destination core. As we can see, NoC design is inspired by the OSI model of computer networking with some modifications.

1.2.3 Homogeneity and Granularity

A NoC can be characterized by its *homogeneity* and *granularity*.

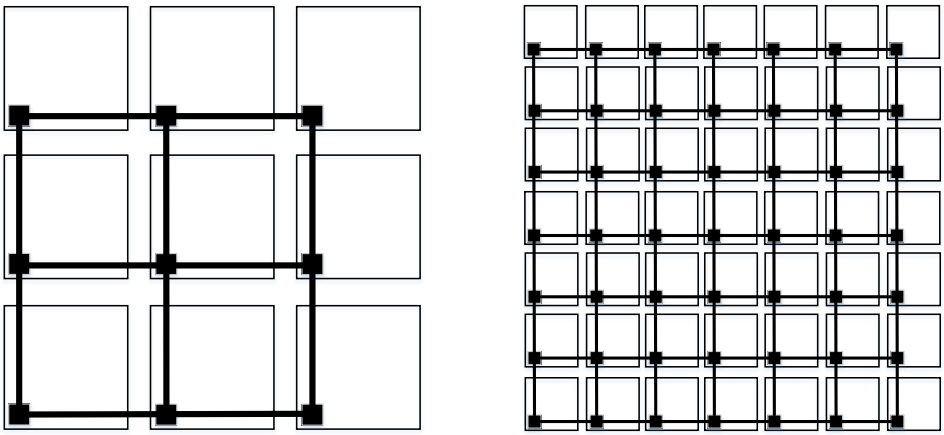
When talking about *homogeneity* we refer to the types of processing elements that exist on the platform. As mentioned before a tile can be of various types. Thus, in a *homogeneous* NoC, all the tiles are of the same type, whereas on a *heterogeneous* NoC the tiles can be of different types such as processor-memory tiles, DSP tiles or even FPGAs.

Granularity refers to the number of cores per surface. *Coarse-grained* materials or systems have fewer, larger discrete components than *fine-grained* materials or systems. A coarse-grained description of a system regards large subcomponents while a fine-grained description regards smaller components of which the larger ones are composed.



(a) Homogeneous **(b)** Heterogeneous

Figure 1.4: A homogeneous and a heterogeneous NoC design



(a) Coarse-grained **(b)** Fine-grained

Figure 1.5: A coarse-grained and a fine-grained NoC design.

1.2.4 The Intel SCC platform

Intel SCC is a NoC platform provided by Intel Corporation with 48 cores. Each tile consists of two x86 processors. This is extremely significant feature since the Linux operating system as well as C and C++ compilers can run on the platform. [4] and [5] describe the key elements of the platform. In short:

- Each tile consists of two blocks, each with a P54C core (second generation Intel Pentium processor), 16 KB instruction and data L1 caches plus a unified 256KB L2 cache.
- A Mesh Interface Unit (MIU) with circuitry to allow the mesh and the interface to run at different frequencies.

- 16 KB Message Passing Buffer.
- Two test-and-set registers.

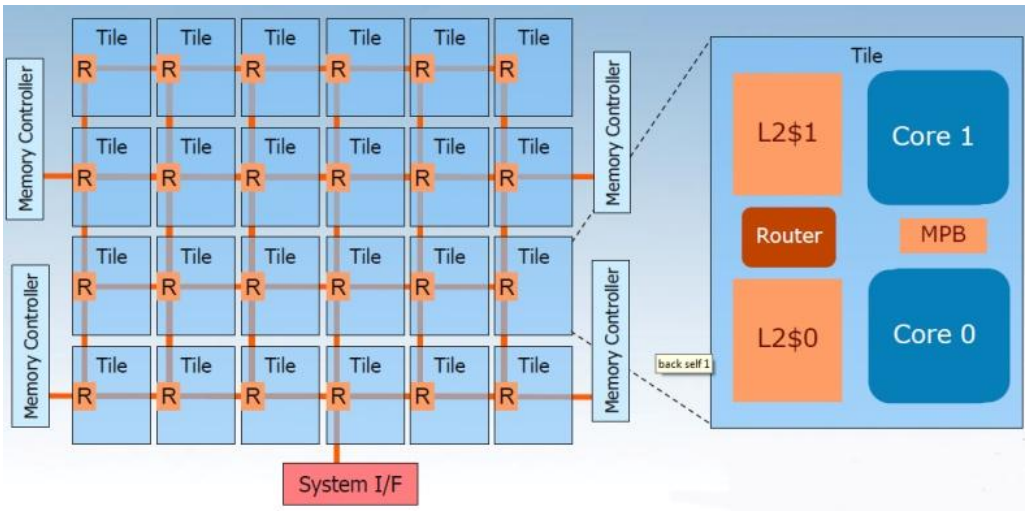


Figure 1.6: Overview of the Intel SCC platform[5].

Each tile has its own router. This router works with the Mesh Interface Unit (MIU) to integrate the tiles into a mesh. The job of the MIU is to packetize and depacketize data onto the mesh. In addition, the router is connected to an off-package FPGA to translate the mesh protocol to the PCI express protocol, thus allowing the chip to interact with a computer. As far as the memory is concerned each core has its own private DRAM, which can be accessed only by itself. Except for that, SCC provides a DDR3 off-chip RAM ranging from 16GB to 64GB and is controlled by the 4 memory controllers as shown in figure 1.6. In order for cores to communicate much faster, SCC includes a message passing buffer (MPB) in each tile. It provides a fast, on-die shared SRAM, as opposed to the bulk memory accessed through four DDR3 channels. While the processor does not offer any hardware-managed memory coherence, it features a new memory type to enable efficient communication between the cores called the Message Passing Buffer Type (MPBT). The memory architecture along with the MPB is shown in figure 1.7.

The MPB is 16KB in each tile. This memory is shared to all cores of the platform. Thus, when a core sends a message to another core, basically he writes the data to the shared memory of the latter. In order to achieve synchronization and data coherence a test&set register exists in each core. The operation on this register is atomic, providing a solution to the synchronization problem.

To help programmers to write programs for the SCC platform Intel provides RCCE. RCCE is a communication enviroment, which distributes evenly the MP address space to the 48 cores, thus assigning 8KB to each core. Additionally, RCCE provides two interfaces

for inter-node communication, *gory* and *basic*. For our implementation, we used the gory interface which offers the programmer greater control over the SCC but does not provide synchronization methods.

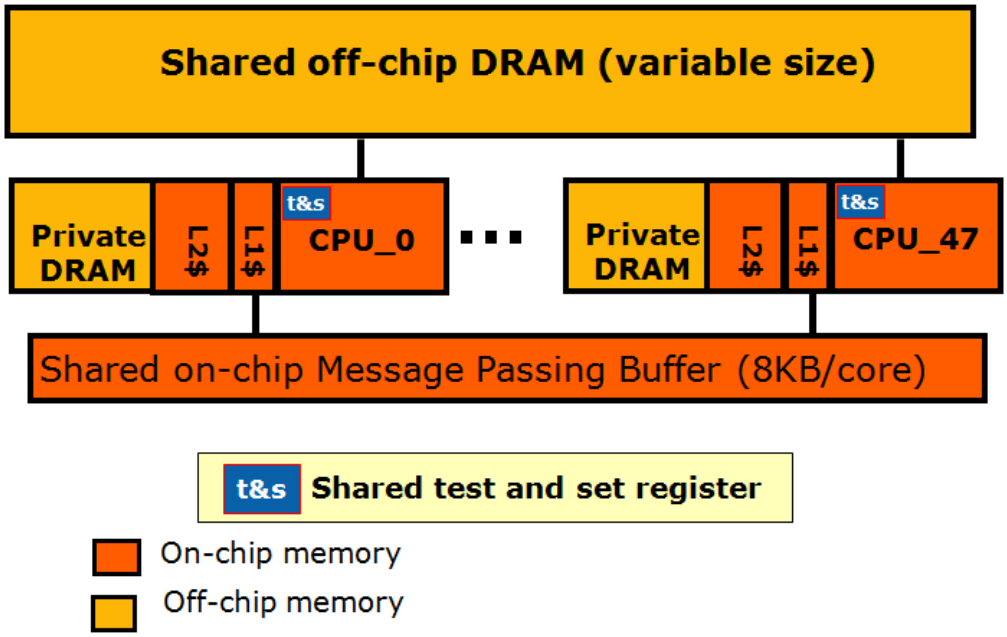


Figure 1.7: Memory architecture of SCC and the MPB.[4].

Every memory operation requires that we allocate memory in the MPB of MPBT data type. For that purpose, we use the *RCCE_malloc* function. After the allocation, we can exchange messages between cores by writing and reading from the allocated space. We use *RCCE_put* and *RCCE_get* to achieve these operations, where *RCCE_put* transfers the data from buffer to MPB and *RCCE_get* does the opposite.

RCCE library also provides Boolean flags in order to help programmers with synchronization issues. The *RCCE_flag_alloc* function allocates a flag in the MPB, which can be managed through the *RCCE_flag_write* function and its value can be either *RCCE_FLAG_SET* - which denotes the boolean TRUE - or *RCCE_FLAG_UNSET* - which denotes the boolean FALSE -. The modification of a flag of a node can be performed by any other node but this access is atomic using the test&set register. Lastly, in respect to Unix's *sem_wait* RCCE provides the *RCCE_wait_until* function which forces a core to stall waiting for a specific value of a flag.

As far as the execution of the application is concerned, the user specifies the number of cores to use from a given subset of cores on the chip. Identical executables are launched on all cores. Every executable is assigned a rank, which is a sequence number ranging from 0 to N-1 (N is the number of participating cores). This rank cannot be changed during the execution of the application and it uniquely identifies both the application and the core

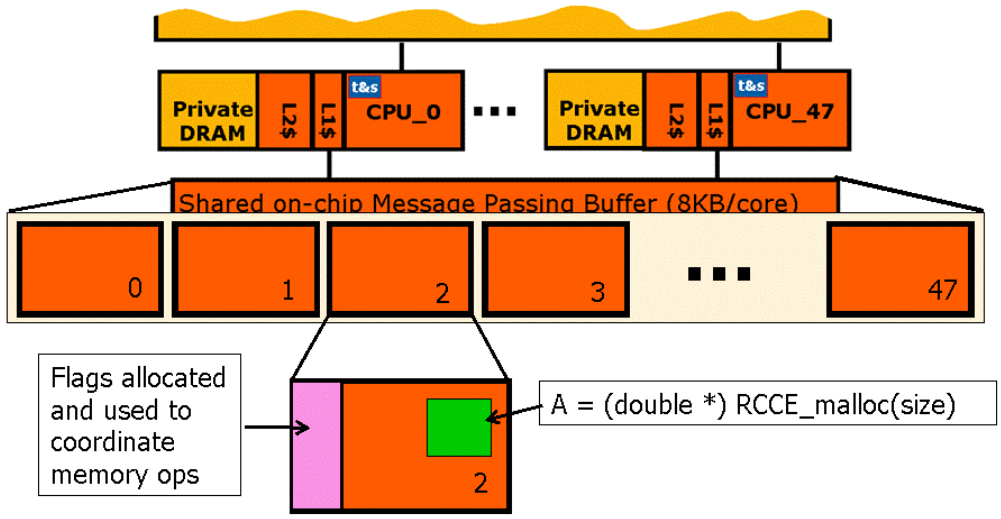


Figure 1.8: Symmetric name space model for the MPB as designed for RCCE library.[4].

it runs on. Finally, all cores can access a disk space in the Management Console which proved to be very helpful in the implementation of the proposed framework on the SCC platform.

1.3 Reliability, Fault Tolerance and Consensus

In distributed systems, programmers have to make sure the correctness of the system even in the case of faulty processes as well as ensure that the system provides the correct output.

Reliability refers into making a system reliable. The failure of a distributed system can result in anything from easily repairable errors to catastrophic meltdowns. Thus a reliable system, or reliable distributed system is designed to be as fault tolerant as possible. A system can be unreliable in any of the following cases: *component failures*, *processor failures*, *network failures* or *failure in reaching agreement*.

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively designed system in which even a small failure can cause total breakdown. Fault tolerance is really important in high-availability or life-critical systems. The ability of maintaining functionality when portions of a system break down is referred to as *graceful degradation*. *Component failures*, *processor failures* and *network failures* come under the objective of the programmer to make a system fault-tolerant.

On the other hand, *consensus* refers to the ability of the system to agree in a single value. This means that all nodes of a distributed systems must agree on the same value not only when the system is faulty but also when it is operating correctly. Examples of applications of consensus include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts. Several consensus algorithms have been suggested in the past years, with Paxos by Leslie Lamport [6] being the baseline for all of them.

1.4 Objectives and Contributions

As a result to the details we provided regarding distributed systems and network on chip systems we can conclude that there is a great amount of similarities between them, not only in the way the systems are organized but also in the way the communication works. Much research has been done in distributed systems about how to detect failures and reach consensus as well as in networks on chip about fault tolerance in hardware level. Some of this work is presented in chapter 2. In this thesis, our goals and contributions are going to be:

i. To implement some known algorithms for distributed systems regarding failure detection, deadlock detection and reaching consensus and combine them with a management framework for malleable application on NoC platforms (see chapter 5).

ii. To test the implemented algorithms on a NoC,architecture simulator.

iii. To present results regarding the implemented algorithms and how they scale with the beforementioned framework.

Chapter 2

Related Work

2.1 Google's Chubby

Chubby [7] is a fault-tolerant system at Google that provides a distributed locking mechanism and stores small files. In practice, there is one Chubby instance per data center. Several Google systems use Chubby for distributed coordination to store a small amount of metadata.

Chubby tolerates faults through replicas. Each Chubby instance runs on a dedicated machine and includes five replicas, which run the same code. Every object is stored as an entry in a database and it is Chubby's obligation to replicate this database. At any one time, one of the beforementioned replicas is considered as the *master* one. Chubby clients, request a Chubby instance for service. The master replica serves all Chubby requests. In addition, if a Chubby client contacts a replica that is not the master, it replies with the address of the master one. If the master replica fails, then a new replica is elected as the master one, which will continue to serve clients based on its local copy of replicated database.

The initial version of Chubby, was based on a commercial, third-party, fault-tolerant database. However, this database had several bugs regarding replication. Thus, Google's engineears decided to replace this third-party database with their tweaked Paxos algorithm as described in [8]. As stated in this paper, despite the existing literature on Paxos algorithm, building the actual system appeared to be non-trivial for the following reasons:

- In order to convert the algorithm into a practical system involved implementing many features and optimization some of which were not published in the literature.
- Even though many fault-tolerant short algorithms are proposed in the literature, building a real system and ensuring that it operates correctly required the usage of

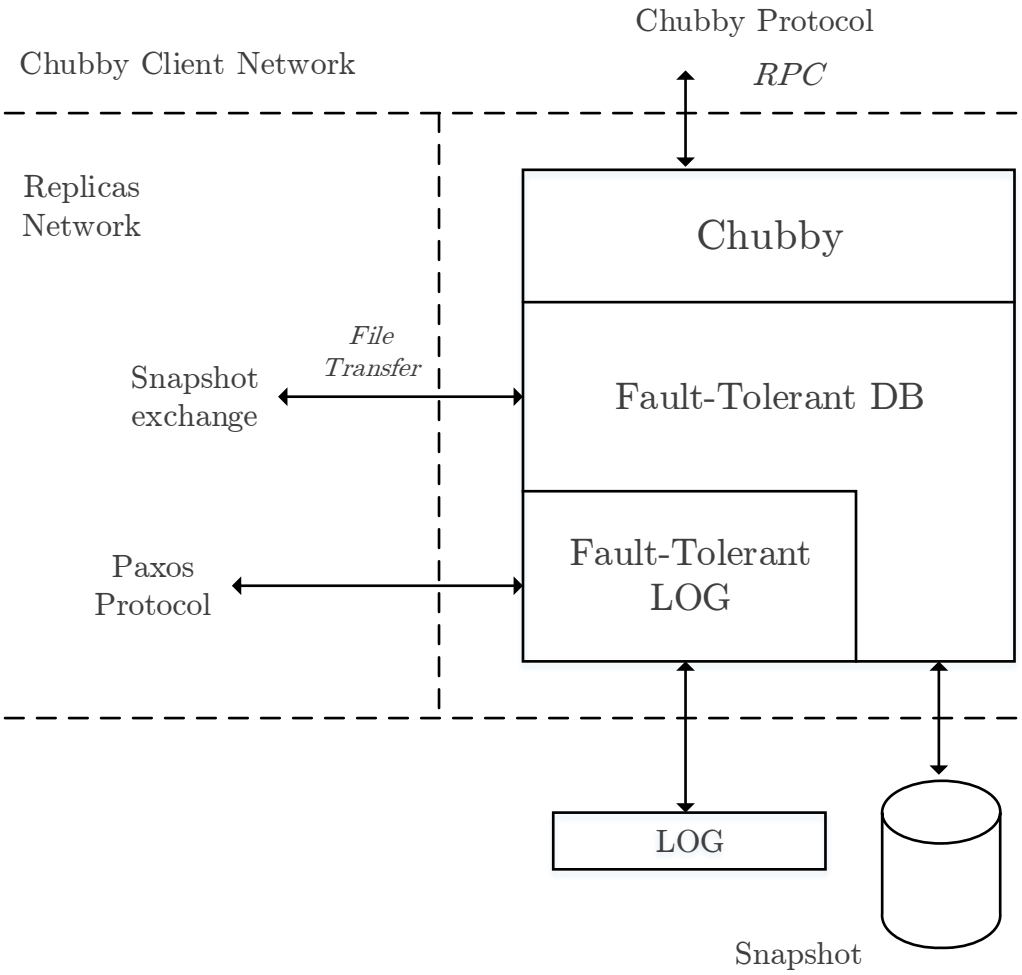


Figure 2.1: A single Chubby instance

different methods .

- Fault-tolerant algorithms tolerate specific faults. When building the real system however, you have to deal not only with a variety of unexpected failures, but also with bugs and errors in your code. Thus, additional effort has to be put.
- A real system is almost never specified in detail. As a result, a system might fail due to misunderstanding that occured during its specification phase.

2.2 Apache's Cassandra 2.0

Apache Cassandra is a free and open-source distributed database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple datacenters, with asynchronous masterless replication allowing low latency operations for all clients. It was initially developed at Facebook [9] to power their

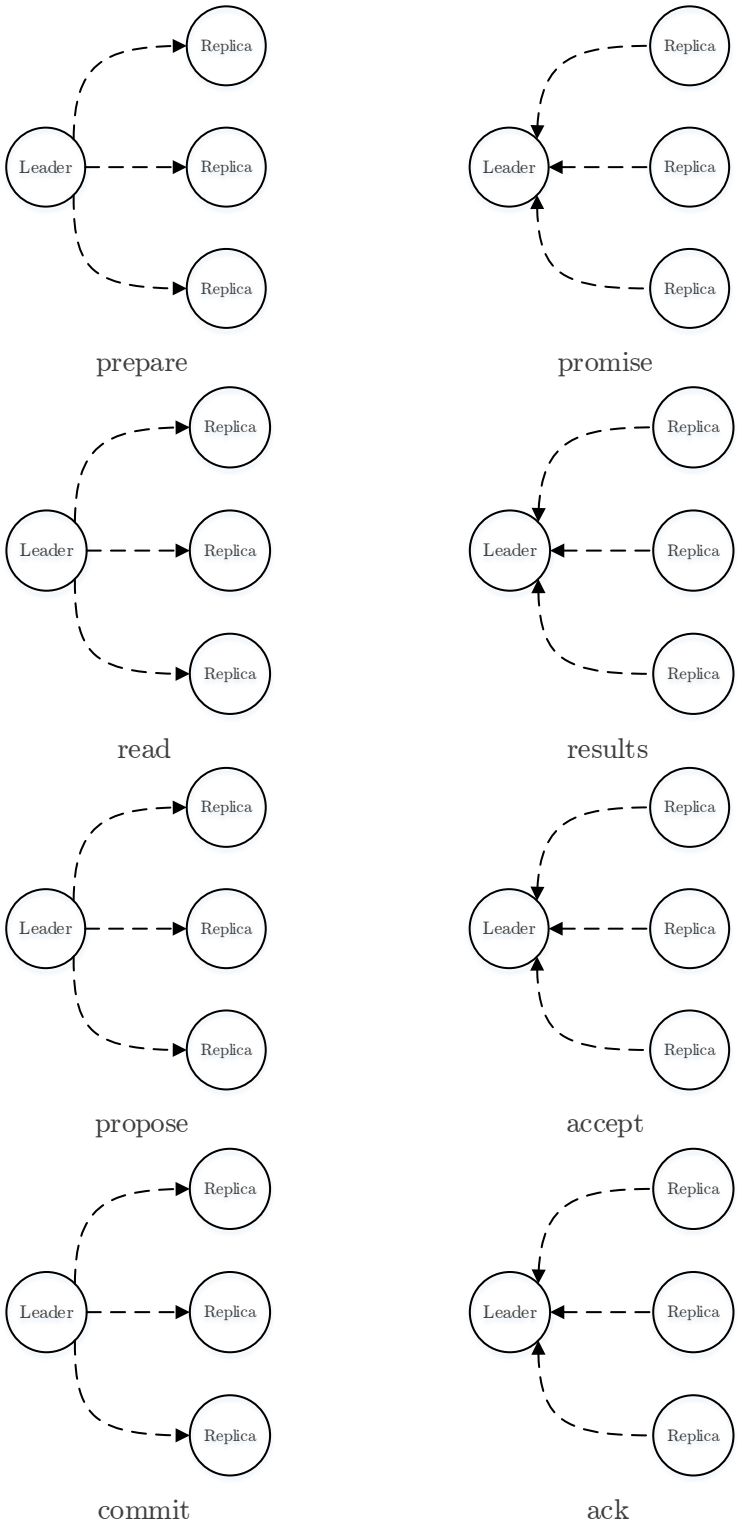


Figure 2.2: Cassandra's 2.0 tweaked Paxos algorithm

Inbox Search feature by Avinash Lakshman (one of the authors of Amazon's Dynamo) and Prashant Malik. It was released as an open source project on Google code in July 2008. Nowadays, many clients such as IBM, Netflix and Apple use Cassandra in their

systems.

As of Cassandra 2.0, in order to improve consistency, Apache's engineers decided to replace their early implentation of distributed locking with their extended Paxos algorithm. Even though Paxos will be described in section 6 their approach is summarized in the following figure:

We can see that they implemented a typical Paxos instance with an additional step in which they need to read the current value of the row to see if it matches the expected one.

2.3 Stochastic Communication: A New Paradigm for Fault-Tolerant Networks-on-Chip

In [10] Bogdan et al focused on on-chip fault-tolerant communication. Traditionally, data networks dealt with fault-tolerance by using complex algorithms, like the Internet Protocol or the ATM Layer. However, these algorithms require many resources that are not available on chip.

The faults that may appear in NoCs are either transient or permanent. The *transient faults* are caused by fluxes of neutron and alpha particles, power supply and interconnect noise, electromagnetic interference, or electrostatic discharge. They represent the most common problem for future VLSI circuits. Simply stated, if noise in the interconnect

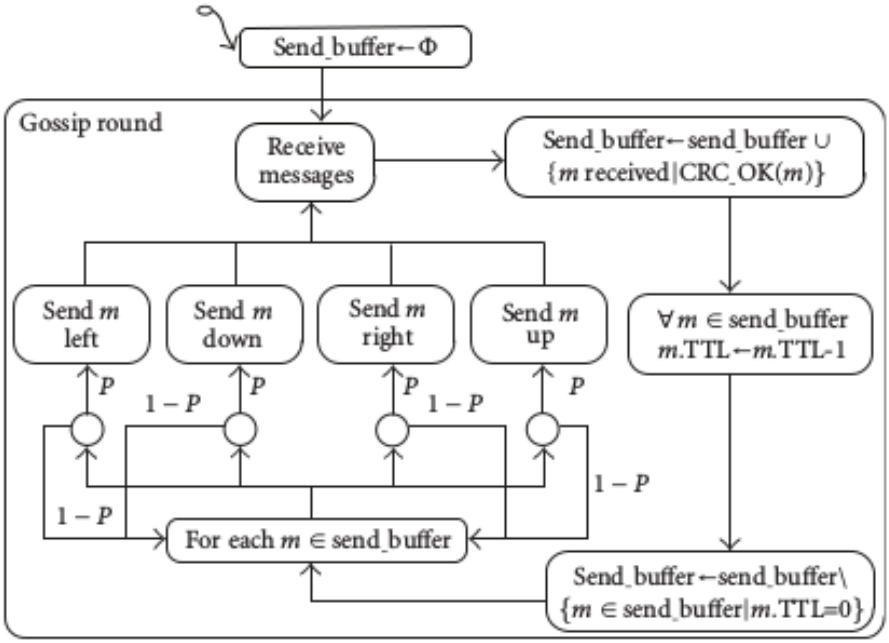


Figure 2.3: Stochastic Communication Algorithm

causes a message to be scrambled, a data upset will occur.

Permanent faults reflect irreversible physical changes in the structure of the circuit. They make recovery very hard or even impossible. However, while these errors occur infrequently and do not pose a serious threat to the mass production of VLSI chips, however this may change for future nanotechnologies.

Another common failure is when a message is lost because of *buffer overflow*. These faults are known as send/receive omissions or buffer overflows.

In this approach, Bogdan et al proposed a fast and computationally lightweight scheme for the on-chip communication , based on an error-detection/multiple-transmissions scheme. The key observation behind their strategy is that, at chip level, the bandwidth is less expensive than in traditional networks; this is due to the existing high-speed buses and interconnect fabrics which can be used to implement NoCs.

Chapter 3

Basic Abstractions

In this chapter, based on [11], we present the basic abstractions we use to model a distributed system which is formed by several entities that can execute commands and by communicate through message exchanging. We present abstractions concerning both the entitites that compose a distributed system as well as the types of communication between them.

3.1 Abstractions in Distributed Systems

3.1.1 Processes and Messages

For the rest of this book, we abstract the units which are able to execute commands in either a distributed system or a many-core system with the term *node*. We persume that our system is composed of N distinct processes, thus creating a set denoted by Π. In most cases, this set is static unless stated otherwise. In addition, each node has a unique identification number, named *id*, ranging in $\{1, ..., N\}$. In the description of an algorithm, this *id* number is used to denote the node with which we communicate with.

Processes communicate with each other by exchanging messages, which are uniquely identified either by using a sequence number or a local clock along with the sender's node id. In other words, we assume that all messages ever exchanged are unique, and messages are not duplicated in any way.

3.1.2 Safety and Liveness

Implementing and applying a distributed algorithm to a set of processes can be a really painful achievement. Throughout the whole process, we must seek to satisfy the properties of the abstraction in all possible executions of the algorithm. In other words, we have to foresee all the possible sequences of steps executed by the processes according to

the algorithm. The properties of the abstraction to be implemented needs to be satisfied, if not for all, for a large of possible interleavings of these steps. Most of the times, the properties that have to be satisfied are the following: *safety* and *liveness*. The distinction between these two usually helps to understand the level of the abstraction and to propose an efficient algorithm. However, the real challenge is to guarantee both liveness and safety. Indeed, useful distributed services are supposed to provide both liveness and safety properties. Reaching abstraction with only one kind of property is usually a sign for an improper implementation.

3.1.2.1 Safety

By the term *safety property*, we assume a property of a distributed algorithm that can be violated at some time t and never be satisfied again after that time. Informally speaking, the safety properties of an algorithm ensure that "something bad will never happen". For example, if we consider perfect links between processes (meaning that communication links should not vaguely create messages) then a process should never receive a message unless this message was indeed sent. If at some time t a process received a message that was never sent, we have to be able to correct this issue.

More specifically, a safety property is a property such that, if it is violated in some execution E of an algorithm then there is a partial execution E' of E such that the property will be violated in every extension of E'. This means that safety properties prevent a set of unwanted states from occuring.

3.1.2.2 Liveness

In respect to the safety property, the liveness property ensures that "eventually something good will happen". Taking the links between processes again as an example, we require that if a process sends a message to another process then the latter should eventually receive the message. To insinuate that a liveness property is violated, we have to prove that there is an infinite number of steps of the algorithm, thus resulting in a message never being delivered. More precisely, a liveness property is a property of a distributed system execution such that, for any time t, there is some hope that the property can be satisfied at some time $t' \geq t$.

3.2 Crashes and Failures

At any given time a process of our distributed system executes the steps of the algorithm which has been assigned to it. A *failure* occurs whenever the process does not behave according to the algorithm. At the time of failure all components of the process

become unavailable too. There are many kinds of failures. Strictly speaking, in case of a crash failure, the process simply stops executing any command and does not send any message to other processes, whereas more mildly speaking, in a case of an arbitrary failure the process just deviates from the algorithm. In figure 3.1 we see the types of process failures.

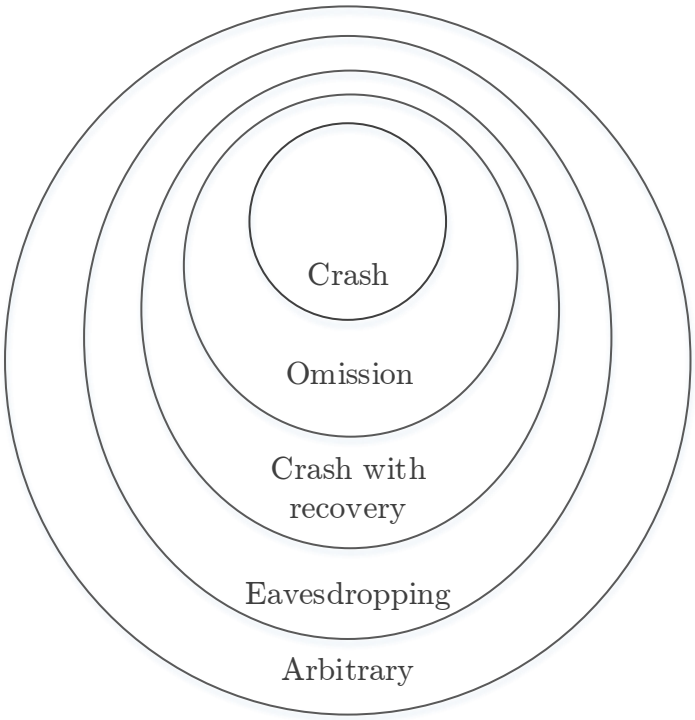


Figure 3.1: Types of process failures

3.2.1 Crashes

The simplest way that a process could fail, is by crashing. That means that at some time t the process just stops executing commands and sending messages. Until that time, the process executes its algorithm correctly, meaning that it executes its local workload without any problems and sends messages to other processes as expected. Thus said, a process is detected as *faulty* if it crashes at some time during the execution. If the process completes its work without any failure then it is said to be *correct*. In most cases, we create some form of agreement, in terms that only a limited number f of processes can be faulty. Assuming a bound on the number of faulty processes in the form of a parameter f means that any number of processes up to f may fail, but not necesseraly all of them.

In the crash-stop abstraction, a faulty process executes its algorithm correctly, but after it has crashed, it never recovers. In practice, a process that crashes can be restarted and recover from the faulty state, which is desirable. But with the crash-stop abstraction

recovered process is no longer part of the system. Nothing prevents recovered processes from getting informed about any results broadcasted in the system, however, and from participating again in subsequent instances of the distributed algorithm.

3.2.2 Omissions

A more general kind of fault is an *ommision* fault. An omission fault occurs when a process does not send or receive a message. Most of the times, this is caused due to buffer overflows or network congestion. With an omission, the process may not perform as it was supposed to since it dropped some messages that should have been exchanged.

3.2.3 Crashes with Recoveries

In the crash-recovery abstraction, we assume that a process is faulty if it either crashes and never recovers (as in 3.2.1), or the process keeps crashing and recovering during a given time. A process that keeps crashing and recovering is considered correct in this model, however, a process might suffer *amnesia* when it crashes, meaning that it loses its internal state and might send messages that oppose to previous messages that it might have sent. To overcome this problem, most of the times we assume that every process has its own log file stored in a stable storage, which can be accessed after a crash event. In this model, we assume that a process is aware that it has crashed after its recovery. Thus, the enviroment should be responsible to inform the recovered process for its state. In addition, the crashed process might have had some data which have been lost during the failure. This data should be also properly reinitialized.

3.2.4 Eavesdropping Faults

When a distributed system operates in an untrusted enviroment, some of its components may become exposed to outsiders. Therefore, the outsiders may *eavesdrop* on multiple processes and correlate all leaked pieces of information with each other. This kind of faults threaten the confidentiality of the data exchanged by the algorithm.

3.2.5 Arbitrary Faults

An arbitrary fault, is the most general kind of failure as shown in figure 3.1. In this type of failure, we make no assumpions on the behaviour of faulty processes. They are allowed to send any kind of message and generate any kind of output. When an arbitrary fault presents itself, the process deviate from the algorithm it executes in any possible way. Such failures are also called *Byzantine* (sec. 6.1) or *malicious* failures. Arbitrary faults are the most difficult to tolerate, since one does not know exactly what caused the failure.

This type of failure is also used when processes may become controlled by malicious users who try to prevent the normal operation of the system.

3.3 Timing Assumptions

One of the most important parts to categorize distributed systems is the relationship between the exchange of messages and the time elapsed. More specifically, if we can determine any time bound on communication delays or even execution time speeds could be proved as of great importance not only for designing a proper execution algorithm but also to be able to detect failures in our system. In respect to the time bounds of message delivery, a system can be categorized into three categories; *synchronous*, *asychronous* and *partially sychronous* system.

3.3.1 Sychronous System

To assume a sychronous system, one of the following properties have to be fulfilled:

- **Synchronous Computation**: The time that processes take to execute a single step is always less than a known upper bound. A single step includes not only the time that a process takes to make a computation but also the time to receive and send the appropriate messages.
- **Synchronous Communication**: In this type of system there is a known upper bound on the delivery delay of messages. That is, the time period between the moment that a signal is sent by a sender, and the moment that the signal is received by the receiver is always less than a known value.
- **Synchronous physical clocks**: Each process has its own local physical clock which differs from a global real-time clock only by a known upper bound.

In general, synchronous systems provide a lot of useful services for managing the system such as *timed failure detection*, *worst-case performance* etc.

3.3.2 Asynchronous System

In an asynchronous system, we make no timing assumptions regarding either processes' computation time nor message delays, meaning that processes do not have access to any sort of physical clock and there is no known upper bound on communication delays. However, if each process follows the steps below, time can be measured with respect to communication.

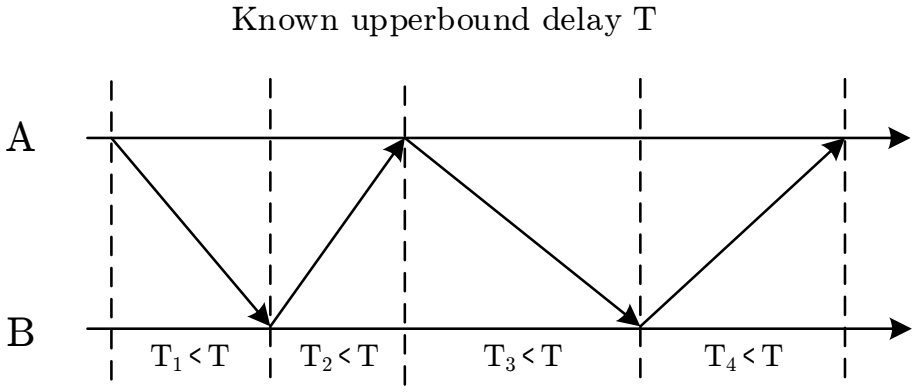
Step 1: Each process i keeps an integer l_i, initially set to 0. This is called a *logical clock*.

Step 2: Whenever an event at process i, we increase l_i by 1.

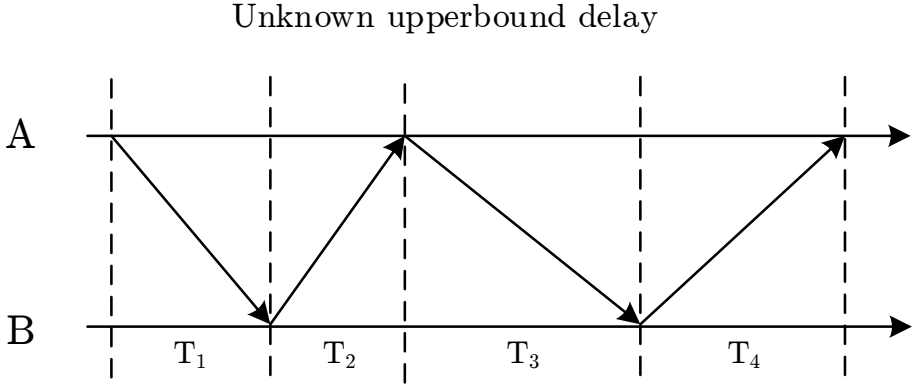
Step 3: Whenever a process sends a message, that message is accompanied by a timestamp $t(e)$, which is the value of the logical clock l_i at that time.

Step 4: When a process receives a message m with timestamp t_m, it updates its logical clock based on this expression: $l_p := max\{l_p, t_m\} + 1$.

This way of measuring time is called *logical time*.



(a) In a synchronous communication, message delays are not greater than an upper bound value T.



(b) In an asynchronous communication, there is no known upper bound delay.

Figure 3.2: Example of synchronous and asynchronous communication.

3.3.3 Partially Synchronous System

In practice, most distributed systems appear to be synchronous, meaning that we can always find an upper bound that is respected most of the time. However, there might be periods when the system is not synchronous. In example, the network could be overloaded resulting in slow communications, or even retransmissing a lost message could cause excess of the known time bounds. Keeping these in mind, we can define systems that are *partially*

synchronous. In short, in a partially sychronous system timing assumptions only hold eventually, without knowing when exactly.

3.4 Models in Distributed Systems

3.4.1 Combining Abstractions

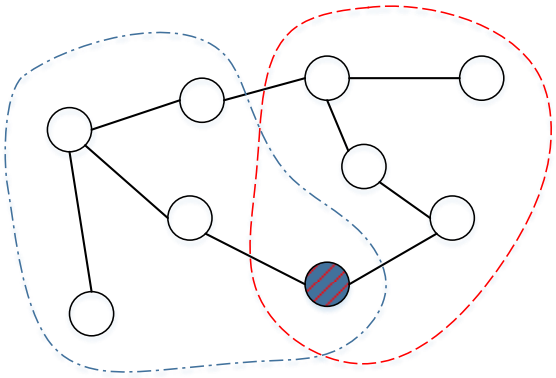
Clearly,we will not consider all possible combinations of basic abstractions.On the other hand, it is interesting to discuss more than one possible combination to get an insight into how certain assumptions affect the design of an algorithm. We have selected two speci?c combinations.

- Fail-stop. We consider the crash-stop process abstraction, where the processes execute the deterministic algorithms assigned to them,unless they possibly crash, in which case they do notr ecover.Aditionally, we assume the existence of a perfect failure detector $\mathcal{P}$.
- Fail-noisy. We consider the crash-stop process abstraction with the existence of the eventually perfect failure detector $\Diamond\mathcal{P}$.

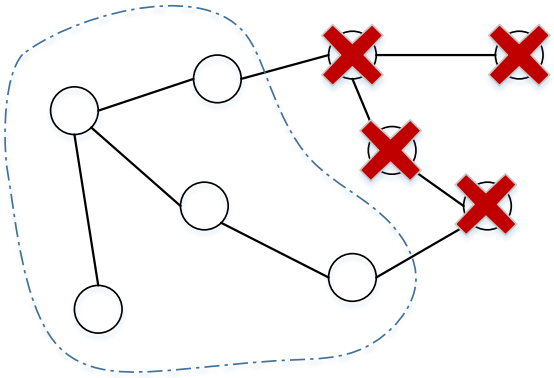
3.4.2 Quorums

A significant tool of many fault-tolerant algorithms are *quorums*.

A *quorum* in a system with N processes is any set of more than $N/2$ processes or equivalently any set of $\lceil \frac{(N+1)}{2} \rceil$ processes. It is easy to prove that every two quorums have at least one process in common and also that even if $f < N/2$ processes fail by crashing, there is always one quorum of non-crashed processes. Quorums are used in Paxos algorithm described in chapter 6.



(a) Two quorums and their common process



(b) A quorum in a system with $f < N/2$ failed processes.

Figure 3.3: Quorums' Properties

3.4.3 Performance

When talking about performance, there are mainly two metrics that we are concerned about:

1. The number of messages required to terminate an operation and
2. The number of communication steps required to terminate an operation.

In some algorithms there are two addition metrics that might help us measure the perfomance:

3. The total size of communication message, measured in bits and
4. The number of accesses to stable storage.

When designing an algorithm, we try to achieve best performance in case of failure-free executions, while ensuring that our algorithm will tolerate any failures that might show up.

Chapter 4

Deadlock and Failure Detection

Deadlock detection as well as failure detection are two important problems in distributed systems and much attention has been devoted in the research community.

Generally speaking, a deadlock situation is the possible result of competition for resources, such as several processes requesting exclusive access to particular data. Although deadlock problems first appeared as a problematic situation in multi-threading and multi-core systems, they are also a significant part of distributed systems, known as *distributed deadlocks*. Handling of deadlock becomes highly complicated in distributed systems because no process has accurate knowledge of the current state of the system and because every inter-process communication involves a finite and unpredictable delay. Deadlock handling can be separated into three main categories:

- *Deadlock Prevention*: This type of handling is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource. However, this approach is highly inefficient and impractical in distributed systems.
- *Deadlock Avoidance*: In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system). Due to several problems though, this approach is also impractical in distributed systems.
- *Deadlock Detection*: Deadlock detection seems to be the best approach to handle deadlocks in distributed systems. In this approach, we examine the interaction status of processes with resources and we try to find a cyclic wait.

A failure situation is the possible result of having one process experience any of the failures described in section 3.2.1. Failure detectors were proposed in 1996 as a mechanism for solving consensus in an asychronous message-passing system with crash failures by

distinguishing between slow processes and dead ones. The basis idea is that each process has attached to it a failure detector module that continuously outputs an estimate of which processes in the system have failed.

4.1 Deadlock Detection

As mentioned before deadlock detection is probably the best way of handling deadlocks in distributed systems. The correctness of a deadlock detection algorithm must satisfy the following two conditions:

1. *Progress*: The algorithm must detect all existing deadlocks eventually, meaning that there should be no undetected deadlocks.
2. *Safety*: The algorithm should not report deadlocks which do not exist, also known as *phantom* deadlocks.

In the following sections we briefly analyze the models of deadlocks as well as the classes of deadlock detection algorithms. A more detailed description can be found in [12].

4.1.1 Wait-For-Graphs

A *Wait-For-Graph (WFG)* is a mathematical model of resource requests. Each vertex V_i of the graph denotes a process $i \in \{1, ..., N\}$ of our distributed system and each directed edge E_{ij} represent blocking relations between processes. More precisely, an edge from node V_1 to node V_2 indicates that V_1 is blocked and is waiting for V_2 to release some resource or output some data. A *directed cyclic path* or a *knot* in this graph denotes that the system is deadlocked.

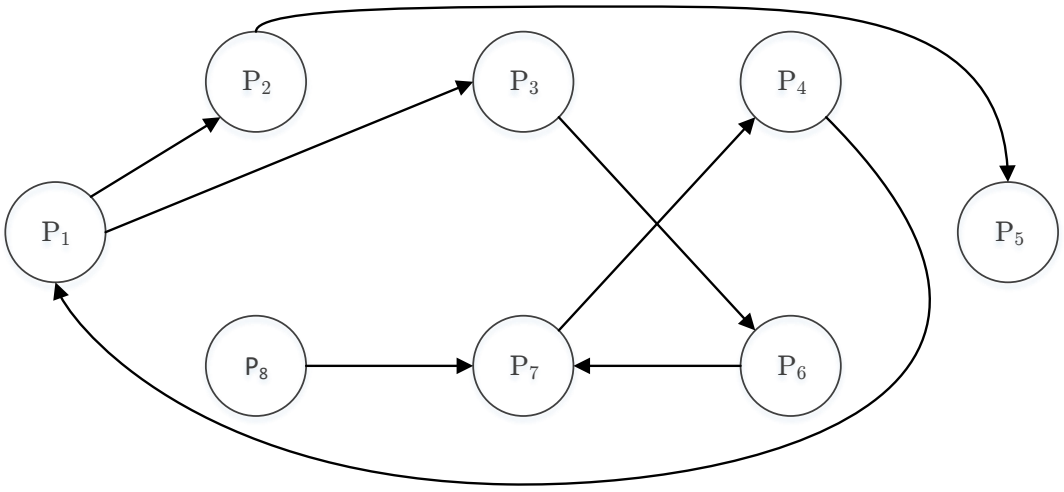


Figure 4.1: Example of a WFG

By the term *directed cycle path* we mean that starting from any vertex V_i and following any path given by the directed edges we can end up in our starting vertex V_i.

On the other hand, a vertex V_i is a knot if ($\forall V_j :: V_j$ is reachable from $V_i \Rightarrow V_i$ is reachable from V_j). More simply, no paths originating from a knot have "dead ends".

4.1.2 Models of Deadlocks

Depending on the application, the distributed system might allow different kinds of resource management. For example, a process might need to acquire a combination of resources in order to perform some operations. In this section we introduce the models of deadlocks which are used to classify deadlock detection algorithms in section 4.1.3.

4.1.2.1 Single-resource Model

In the single resource model, a process can have one outstanding request at most for only one unit of resource. Hence, the maximum out-degree of a vertex in a WFG for the single-resource model can be 1. To find a deadlock in this model, we just have to find a cycle in the WFG.

4.1.2.2 AND Model

In the AND-model a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the specific process (therefore, requests of this type are called AND requests). The AND model has been the traditional view of resource requests in distributed systems. The vertices of the WFG in this model are called *AND nodes* and may have outdegree more than 1. The presence of a cycle in the WFG indicates a deadlock in the AND model. For example, in figure 4.1 process P_1 has two outstanding requests and both must be satisfied before before P_1 continues its execution. This example depicts a deadlock situation corresponding to the cycle $P_1 \rightarrow P_3 \rightarrow P_6 \rightarrow P_7 \rightarrow P_4 \rightarrow P_1$.

4.1.2.3 OR Model

In the OR model a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted. Consider the following example in figure 4.1: If all vertices are *OR nodes* then process P_1 is not deadlocked because P_5 has no outgoing edges. Hence, once P_5 and P_2 are completed so can P_1. Thus, the presence of a cycle in the WFG does not imply a deadlock in the OR model. However, the presence of a knot does[13].

4.1.2.4 AND-OR Model

The AND-OR model is a generalization of the previous two models (AND model and OR model). In this model, a process can request for multiple resources in any combination of *and* and *or* types. For example, in the AND-OR model a request for multiple resources can be of the form $x \cap (y \cup z)$. To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

4.1.2.5 $\binom{n}{k}$ Model

The $\binom{n}{k}$ Model (called k out of n model) allows a process to obtain any k available resources from a pool of n resources. This model is a generalization of the AND-OR model, however it has the same in expressive power as the AND-OR model. That said, every request in the $\binom{n}{k}$ model can be expressed in the AND-OR model and vice-versa. Note that AND requests for k resources can be stated as $\binom{k}{k}$ whereas OR requests of k resources can be stated as $\binom{k}{1}$.

4.1.2.6 Unrestricted Model

This is the most general model of deadlock detection. In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests. The advantage of looking at the deadlock problem in this way is that it helps in the separation of concerns: Properties of the underlying database computations (e.g., degree of concurrency) are rigorously abstracted and separated from concerns about properties of the problem (stability of deadlock). Therefore, all the algorithms dealing with this general model can be used to detect other stable properties as well.

4.1.3 Classes of Deadlock Detection Algorithms

Distributed deadlock detection algorithms can be divided in four classes: *path-pushing*, *edge-chasing*, *diffusion computation* and *global state detection.*

4.1.3.1 Path-Pushing Algorithms

In *path-pushing* algorithms, distributed deadlocks are detected by maintaining an explicit global WFG. The basic idea is to build a global WFG for each cluster of the distributed system. In this class of algorithms, whenever deadlock computation is performed at any cluster,then the local WFG is sent to all the neighboring clusters. After the WFG of each cluster is updated, this updated WFG is then passed along to other clusters, until

some cluster has a sufficiently complete picture of the global state to announce deadlock or to announce that no deadlocks are present.

4.1.3.2 Edge-Chasing Algorithms

In an *edge-chasing* algorithm, the presence of a cycle in a distributed graph is found by sending special messages called *probes*, along the vertices of the graph. These messages are different that the request and reply messages that processes send. A cycle is detected if a process receives a probe message it has previously sent. In this approach, only blocked processes propagate probe messages along their outgoind edges. Whenever a process that is executing receives a probe message, it discards this message and continues.

4.1.3.3 Diffusing Computation

4.1.3.4 Global State Detection

A *global state* based deadlock detection algorithm exploit the following two facts:

1. A consistent snapshot of the distributed system can be obtained without halting the underlying computation
2. If a stable property holds in the system before the snapshot collection is initiated this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by talking a snapshot of the system and examining it for the condition of a deadlock.

4.2 Failure Detection

As mentioned before, failure detectors were proposed in 1996 by Chandra and Toueg [14]. The output a failure detector produces does not have to always be correct. In contrast, both [14] and [15] are explaining how bogus the output of a failure detector can be and still be useful.

4.2.1 Classification of failure detectors

Chandra and Toueg define eight classes of failure detectors, based on when they suspect faulty processes and non-faulty ones. In general, there are two classes of suspicion:

1. Completeness: suspicion of faulty processes.
2. Accuracy: suspicion of non-faulty processes.

4.2.1.1 Degrees of completeness

When speaking of completeness we have two types in our mind:

- ◇ **Strong Completeness**: Every faulty process is eventually permanently suspected by *every* non-faulty process.
- ◇ **Weak Completeness**: Every faulty process is eventually permanently suspected by *some* non-faulty process.

There are two temporal logic operators embedded in these statements: "eventually permanently" means that there is some time t_0 such that for all times $t \geq t_0$, the process is suspected. Note that completeness says nothing about suspecting non-faulty processes. Thus, a paranoid failure detector that permanently suspects everybody has strong completeness.

Table 4.1: Failure detectors classes

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\Diamond\mathcal{Q}$	*Eventually Weak* $\Diamond\mathcal{W}$

4.2.1.2 Degrees of accuracy

These describe what happends with non-faulty processes as well as with processes that have not crashed yet. So, when speaking of accuracy we have the following four types in our mind:

- ◇ **Strong Accuracy**: *No* process is suspected by anyone before it has indeed crashed.
- ◇ **Weak Accuracy**: *Some* non-faulty processes are never suspected.
- ◇ **Eventual Strong Accuracy**: After some initial period of confusion, no process is suspected before it crashes. This can be simplified to say that *no non-faulty process is suspected after some time*, since we can take end of the initial period of chaos as the time at which the last crash occurs.
- ◇ **Eventual weak accuracy**: After some initial period of confusion, *some non-faulty process is never suspected.*

Note that *strong* and *weak* mean different things for accuracy versus completeness; for accuracy, we are quantifying over suspects, and for completeness, we are quantifying over

suspectors. Even a weakly-accurate failure detector guarantees that all processes trust the one visibly good process.

4.2.2 Classes of failure detectors

Combining the above degrees of accuracy and completeness results in eight classes of failure detectors. A failure detector is said to be perfect if it satisfies both strong completeness and strong accuracy. We call this set of failure detectors as *class of Perfect failure detectors* and we denote it by $\mathcal{P}$. Similar denotions are provided for the rest classes and are shown in the table 4.1.

Chapter 5

The DRTRM framework

The DRTRM, whose initials came from "Distributed Run-Time Resource Management", is a framework designed to run malleable applications on many-core platforms by Iraklis Anagnostopoulos et al. [16]. The framework is responsible for both allocating initial cores for an application to be executed on and providing a way that the application is managed throughout execution time. The goal of the framework is to minimize messages between nodes and also attempt to equally distribute resources between applications. In the following sections we will try to give a brief explanation of this framework by emphasizing on its main aspects. The main idea behind DRTRM is that it classifies cores in disjoint sets. A really detailed description of the framework can also be found in [17]. In addition, in [18] it is shown that resource allocation is highly affected not only by the internal decision mechanisms but also from the incoming application interval rate on the system. Based on this observation, an effective admission control strategy is proposed, utilizing Voltage and Frequency Scaling (VFS) of parts of the DRTRM which eventually retains the distributed decision making thus improving system performance in combination with significant gains in its consumed energy.

5.1 Cores Types

In this section we will give a brief explanation of the DRTRM framework. As mentioned before the framework classifies cores in 5 sets *controller, initial, manager, worker* and *idle*. In addition, a communication scheme is used between cores in order to exhange information needed to execute the application.

5.1.1 Controller Core

The main job for a controller core is to handle all the unoccupied cores inside his cluster. A *cluster* is a predefined region set at the initialization of the platform and

cannot be modified during run-time. An example of two controller cores along with their clusters is shown in figure 5.1. In addition to handling unoccupied cores, the controller core is also responsible for keeping a list with all manager cores that own a core inside hus cluster. This list is called *Distributed Directory Service (DDS)*. When a manager requests a DDS register, he also sends his controller core his current working cores (manager cores are analyzed in section 5.1.2). However, some working cores might not be in the same cluster as the controller and the manager. It is responsibility of the controller to add the manager core to his DDS list as well as inform the controllers of the working nodes not in his cluster in order to register the manager to their DDS lists too. Except for that, controllers also keep an array with the *ids* of all the controller cores. This array is called *controller_list*.

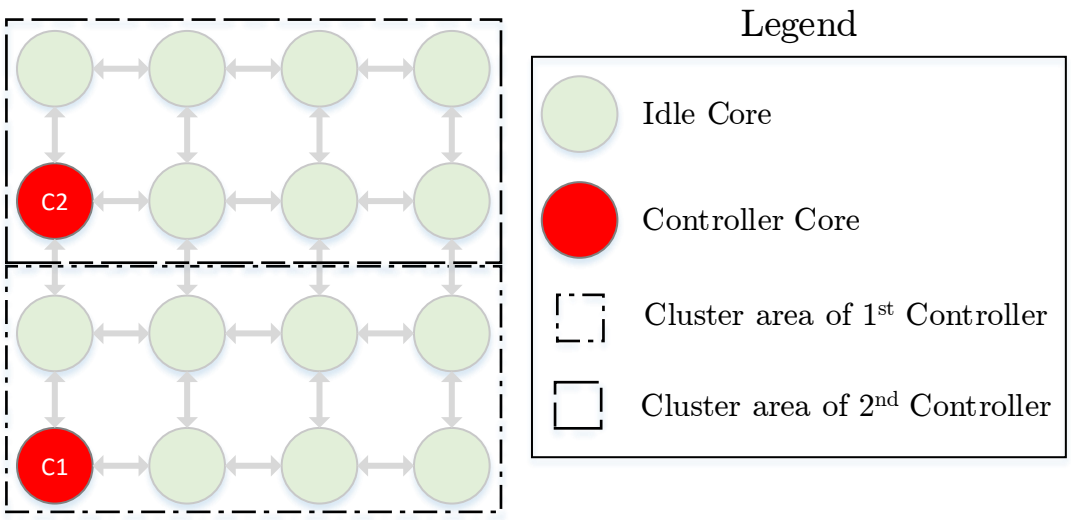


Figure 5.1: Cluster definition

5.1.2 Manager Core

A manager core is responsible for managing an application during its life cycle. More analytically, he informs the working nodes of the application that he is their manager and assign the same amount of workload to each worker. The relation between a manager and an application is one on one, meaning that one manager can only handle exactly one application, and the management of an application can not be devided to several managers.

An example of two manager cores along with their working nodes is shown in figure 5.2. Another task of the manager core is to instruct the resizing of the application. By the term resizing, we mean that the application has either more or less available cores to run on. Thus, the manager has to find out the remaining workload of his working nodes. When he collects this information, then he has to reassign the remaining workload evenly

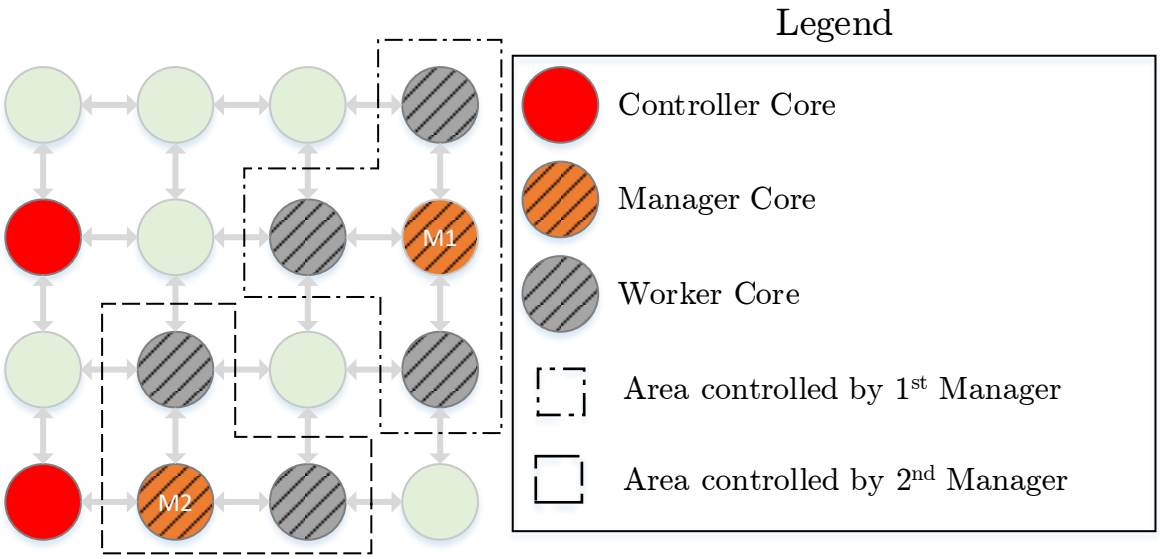


Figure 5.2: Manager cores and their working nodes

to the new set of working nodes. Lastly, the manager core writes the remaining workload as well as important information about the application in an application log file. This is important in order to recover from a manager failure as it will be described in section 7.3.2.

Thus, in order for the manager to accomplish the abovementioned he has to follow the next steps:

Step 1: Sends a signal to his controller core so the latter can add him to his DDS list. In addition, he also sends to his controller his set of working cores.

Step 2: Calculates the workload of his working cores and evenly distributes the application's workload among cores.

Step 3: Signals every working in order to send them their workload as well as inform them that he is their manager.

Step 4: Decides whether he will begin a self-optimization process. If he decides to do so the following steps take place:

- **Substep 1**: Requests workers from controller and manager cores and waits for their offers.
- **Substep 2**: Checks the replied offers. If no offers are received he continues to Substep 6. Otherwise, he picks the offer with the most cores.
- **Substep 3**: Replies to the offers either positevely or negatively.
- **Substep 4**: Initiates the resizing process.

– **Substep 5**: Sends his controller the new working cores in order to be registered to the appropriate DDS lists.

– **Substep 6**: Sets a timer, upon whose expiration, the manager will decide whether he will initiate a new self-optimization process.

5.1.3 Initial Core

The job of an initial core is to find the initial set of cores in which the application will start to be executed on. So, when a new application arrives on the platform an Initial core is chosen randomly and receives a message with the characteristics of the new application. At that point, he temporarily stops whatever he was doing and performs the following steps:

Step 1: Sends requests to controllers and managers and waits for their offers.

Step 2: Checks how many offers he has received. If he has received no offers he repeats Step 1 until at least one core has been found, so he can determine the manager core of the application.

Step 3: By examining the offers he received, he chooses a manager core so that the distance between all of the offered cores is minimized.

Step 4: Replies to all the offers he received, either positevely, meaning that he accepts their offer or negatively.

Step 5: Informs the new manager core that he hass to initialize an application by sending him a signal along with the characteristics of the application and the set of the working cores.

Step 6: Continues executing the tasks he had before the new application arrived, if any.

In the following figure we can see an example of the initial core requesting cores:

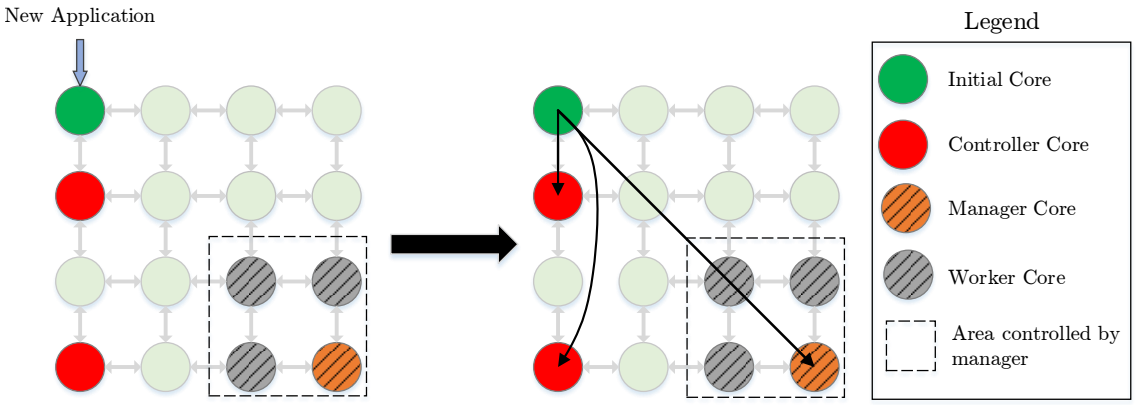


Figure 5.3: Initial core requesting cores

5.1.4 Worker Core

A working core simply executes any workload given by his manager. He should always keep the core ID of his manager in order to inform him if he has completed his workload. As we said before, at any time he could stop the execution of his workload and become an initial core of a new application.

5.1.5 Idle Core

An idle core waits still unless he receives a signal to execute one of the following tasks:

- A new application arrived on the platform and he has to become the initial core of this application.
- He has been voted as the new manager core of an application so he has to follow the steps mentioned in section 5.1.2.
- He was offered as a worker core to a manager by his controller so he has to execute some workload

5.2 Core Lists

As mentioned before each controller core keeps a list called *Distributed Directory Service (DDS)* with all the manager cores that own cores inside their cluster. Except for the DDS list each controller also keeps a list with all the cores inside his cluster called core list.

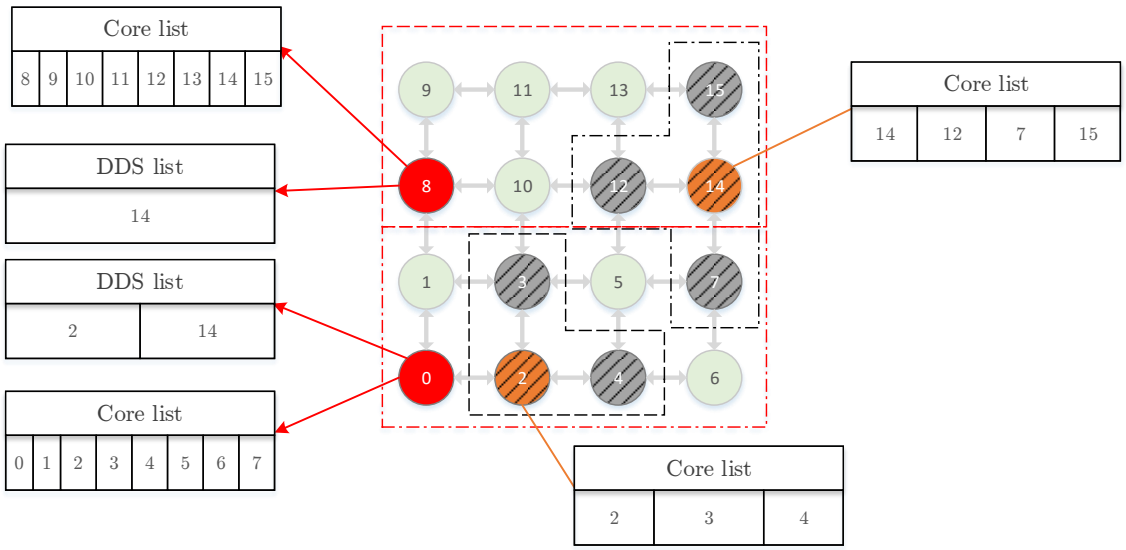


Figure 5.4: Core and DDS lists of managers and controllers

As far as the managers are concerned, they also keep a core list with all the worker cores they own. Inside each element of this list the id of the worker is kept along with the id of his manager. A snapshot of the platform with both controller and manager cores along with their lists is shown in the figure below.

5.3 Primitives of Deadlock Prevention in DRTRM

Whenever a core waits for data from another core, he sends a $\langle SIG\_ACK \rangle$ signal and then stalls waiting for data. However, it is possible that at some time two cores, lets say A and B, send a signal one to each other requesting cores for example. Then, both A and B will send $\langle SIG\_ACK \rangle$ signals to each other, thus stalling waiting for data forever. The solution of DRTRM framework to this problem is by keeping an interaction type between nodes. More specifically, at one point core A can have only one interaction with node B and each core keeps an interaction list in his local memory. If a new signal arrives (meaning that a new interaction type might show up) before the previous has finished it is queued up and the appropriate signal will be sent after the first interaction has been completed. In figure 5.5 we can see a sample of the interaction list of a single core in a platform with N cores.

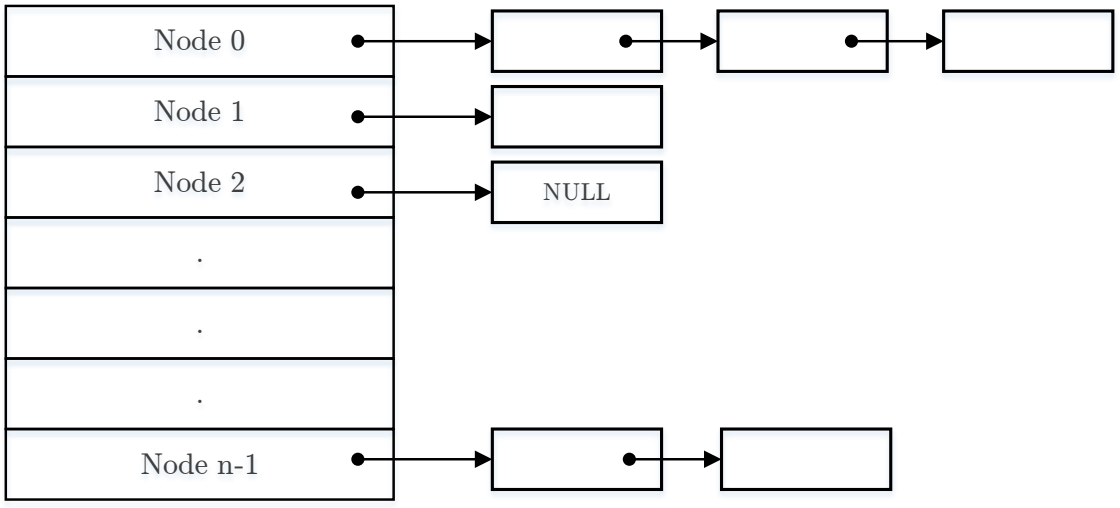


Figure 5.5: Interaction Lists of DRTRM

5.4 Overview

In the following figure one can see an overall flow of the DRTRM framework:

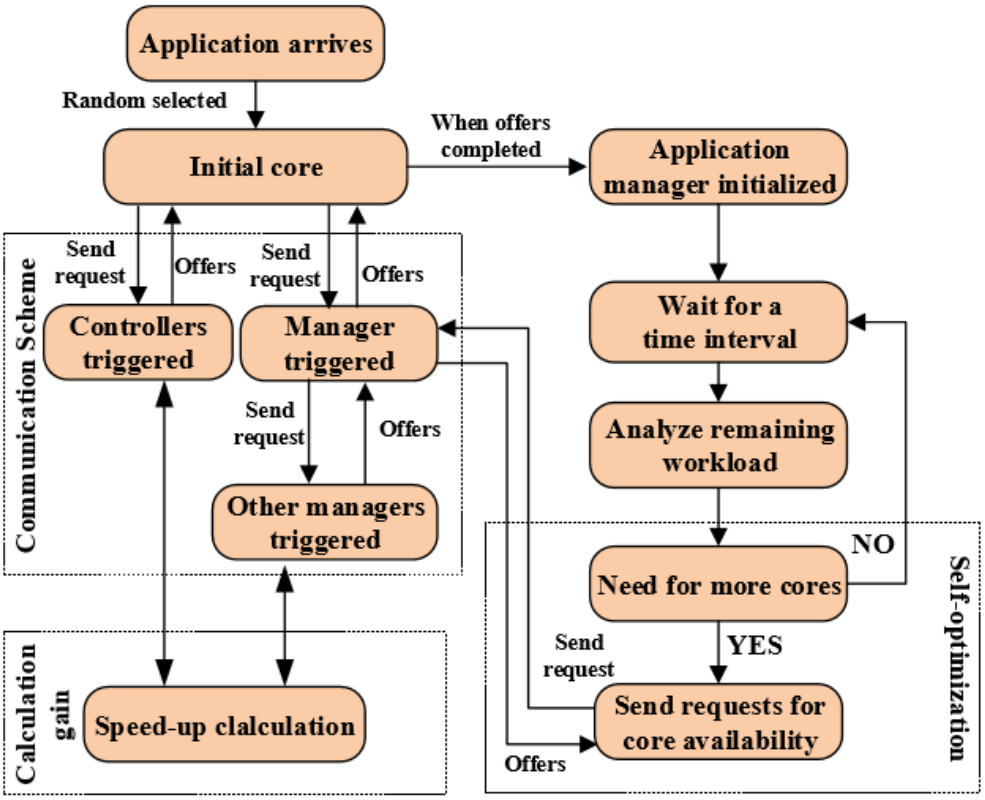


Figure 5.6: Overview of DRTRM [17]

Chapter 6

The Paxos Algorithm

Paxos, as proposed by Leslie Lamport in the late 80s [19] and further analyzed and explained in 2001 [6], is a family of protocols for solving to solve the consensus in a network of unreliable processors. Its name was derived from the Greek island of Paxos where the Paxos' Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber for trading. The algorithm was discovered by Lamport while trying to prove that a complicated algorithm for handling byzantine failures used in a program was impossible. To summarize, a system governed by Paxos is usually talked about in terms of the value, or state, it tracks. The system is build to allow many processes to store and report this value even if some fail which is handy for building highly available and strongly consistent systems.

6.1 Byzantine Fault Tolerance

In fault-tolerant computer systems, and in particular distributed computing systems, Byzantine Fault Tolerance is the characteristic of a system that tolerates the class of failures known as the Byzantine Generals' Problem [20]. The objective of Byzantine fault tolerance is to be able to defend against Byzantine failures, in which components of a system fail with symptoms that prevent some components of the system from reaching agreement among themselves, where such agreement is needed for the correct operation of the system. Correctly functioning components of a Byzantine fault tolerant system will be able to provide the system's service, assuming there are not too many faulty components.

The following practical, concise definitions are helpful in understanding Byzantine fault tolerance[21][22]:

Byzantine fault: Any fault presenting different symptoms to different observers.

Byzantine failure: The loss of a system service due to a Byzantine fault in systems

that require consensus.

6.2 Basic Paxos

Assume a collection of processes that can propose different values or not propose values at all. A consensus algorithm is obligated to choose a single value among the ones proposed, or, if no value was proposed, choose no value. In case a value has been chosen, then processes should be able to learn this value. Thus, the necessary requirements for consensus are the following:

- A value may be chosen only if this value has been previously proposed.
- Only a single value should be chosen.
- Processes learn the chosen value if and only if the value has actually been chosen.

In order for the consensus algorithm to function properly we choose three types of agents, each one of them representing a different role in the scenario. More specifically:

- **proposer**: This type of agent proposes a value that it wants agreement upon to acceptors by sending a proposal containing this value.
- **acceptor**: This type of agent decide whether to accept tha value or not. Each acceptor chooses a value independently. It may receive multiple proposals, each from different proposer.
- **learner**: This type of agent typically does nothing. Once a value is chosen, all learners should be notified of this value.

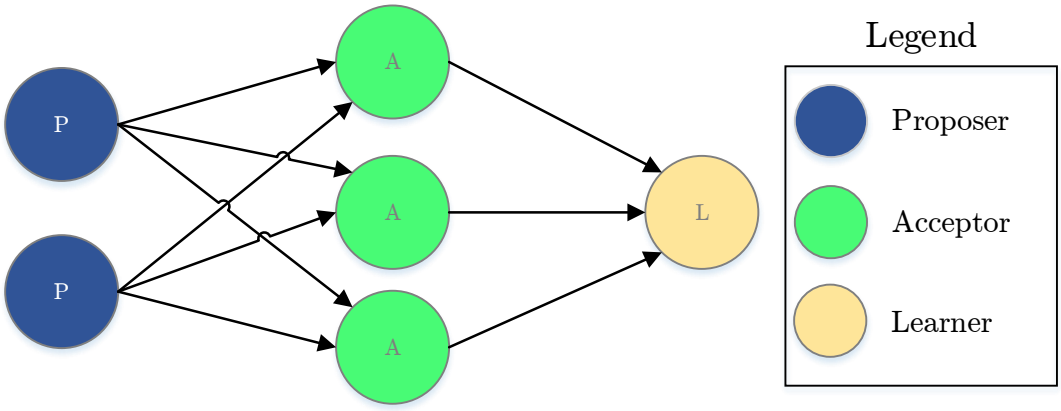


Figure 6.1: Basic Paxos Architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

Figure 6.1 shows a basic Paxos architecture. Agent types are further analyzed in section 6.2.1. Just for reference, in an implementation, a single process may have more than one of the above roles but the mapping from agents to processes does not affect the general idea of the algorithm. For Paxos to function properly we choose the **asychronous**, **non-Byzantine** model in which:

- Agents may fail at any time by either stopping or restarting and additionally they operate at arbitrary speed. In case all agents fail after a value has been chosen it is necessary that some information can be remembered by an agent that has failed and restarted in order to achieve a solution.
- Messages are not corrupted, even though they can be duplicated or lost and may take tremendously long time to be delivered.

6.2.1 Agent Types

In this section, we further analyze each type of agent in the paxos algorithm.

6.2.1.1 Proposer

As mentioned before this type of agent proposes a value that it wants agreement upon. To achieve that it sends a proposal to the set of all acceptors containing the proposed value. Different proposers may propose different values. Thus, we have to keep track of the different proposals issued by different proposers. We can achieve that by assigning a proposal number to each of these proposals such that:

$\Rightarrow$ Each proposer has his own unique proposal number

$\Rightarrow$ Every new proposal has a higher proposal number than any previous proposal

In addition, this type of agent is responsible for broadcasting a chosen value once it has been accepted. By accepted we mean that a majority of the acceptors have agreed on this value. Therefore, proposers should keep track of the decisions acceptors make. Succintly, the steps a proposer follows are:

Step 1: Sends a request along with his proposal number n to each member of a set of acceptors aksing it to respond with:

(a) A promise never again to accept a proposal numbered less than n

(b) The proposal with the highest proposal number less than n that it has accepted, if any.

This is called a **prepare request signal** with number n.

Step 2: Waits until he receives positive reply from a majority of acceptors.

Step 3: If he does, he then sends his proposed value to the same set of acceptors. We call this a **prepare accept signal**.

Step 4: Waits until he receives positive reply from a majority of acceptors.

Step 5: If he does, he then has to inform learners about the chosen value. Thus, he sends a signal to them with the chosen value. We call this a **learn signal**.

The full operation of proposers is described analytically in section 6.2.3.

6.2.1.2 Acceptor

As referenced above, this type of agent decides whether to accept a value or not. Each acceptor chooses a value independently. Additionally, he may receive proposals from different proposers assigned with different proposal numbers. A value is chosen when a majority of the acceptors accept the same value. The steps an acceptor typically follows are:

Step 1: Waits until he receives a signal from a proposer. This signal can be any of these two types: *prepare request* or *prepare accept*.

Step 2: An acceptor can ignore any of these requests without compromising safety.

Step 3: If he decides to reply, he then has to obey the following two rules:

R1. An acceptor can accept a proposal numbered n iff it has not responded to a *prepare request* having a number greater than n.

R2. An acceptor ignores any *prepare request* with proposal number n less than any proposal number of any proposal he has previously replied to as well as *prepare requests* for a proposal it has already accepted.

Step 4: Assuming the acceptor always obeys R1 and R2, he:

- **Step 4^a**: responds to a *prepare request* with a promise not to accept any more proposals numbered less than n and with the the highest-numbered proposal, if any, that it has accepted.
- **Step 4^b**: accepts a *prepare accept* proposal unless it has already responded to a *prepare request* having a number greater than n.

6.2.1.3 Learner

A learner, typically does nothing. He just hangs still until he receives a *learn signal* with the value that has been accepted. However, as mentioned before, an agent could be of multiple types so a learner could be at the same time proposer and acceptor as well. The obvious algorithm to learn the value is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. This allows learners to find out about a chosen value as soon as possible, but it requires each acceptor to respond to each learner a number of responses equal to the product of the number of acceptors and the number of learners. A faster way could be all the acceptors to send an *accepted* message to the proposer, thus making the proposer responsible for broadcasting the chosen value making the complexity equal to the sum of the number of acceptors plus the number of learners.

6.2.2 Distinct Proposal Numbers

There are multiple ways of how we could assign distinct proposal numbers to each proposal. Indicatively, three methods could be the following:

1. Using mutual exclusion each proposer could increase a counter and then pick the newest value as his proposal number.

2. We can assign disjoint sets of numbers for each proposer and have them only choose numbers from their own set. For example, we could assign each node a unique prime number, they choose multiples of that prime number.

3. if we are assuming static membership of participants, we can assign each node a unique number i between 0 and k, where k is the total number of participants, and $n = i + (k * round\_number)$.

6.2.3 Phases

The Paxos algorithm is divided into three phases, the *prepare*, the *accept* and the *learn* phase.

6.2.3.1 Prepare Phase

In the *prepare* phase, the proposer initially selects a unique proposal number n, which is greater than any n previously sent and sends a *prepare request* to a quorum of acceptors. We denote *A(j,k)* as the j^{th} majority set of acceptors with k acceptors in it. The pseudocode below shows the actions of the proposer in the prepare phase:

Algorithm 1 Prepare Phase - Proposer Side

```
1: Pick unique proposal number pn
2: for i ← 1, k in A(j, k) do
3:     Send: prepare_request(i,pn)
4: end for
```

On the acceptor side, if he ever receives a *prepare request* with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted. Algorithm 2 shows the actions an acceptor does in *prepare phase*.

Algorithm 2 Prepare Phase - Acceptor Side

```
1: procedure PREPARE_REQUEST_HANDLER(n)
2:     max_pn ← highest proposed proposal number seen
3:     max_n ← highest accepted proposal number
4:     max_v ← Corresponding value of max_n
5:     if n > max_pn then
6:         max_pn ← n
7:         Reply: prepare_accept(max_n,max_v)
8:     else
9:         Reply: prepare_reject()
10:    end if
11: end procedure
```

6.2.3.2 Accept Phase

After a proposer has received a response to his *prepare_requests* from a majority of the acceptors, he then sends an accept_request to those acceptors with a value v, where v is the value of the highest numbered proposal among the responses, or any value if the responses reported no other proposals. The steps followed by a proposer in the accept phase are shown in algorithm 3.

Algorithm 3 Accept Phase - Proposer Side

```
1: max_n ← 0
2: max_v ← 0
3: cnt ← 0
4: pn ← proposal number from algorithm 1
5: procedure PREPARE_ACCEPT_HANDLER(n,v)
6:     cnt ← cnt + 1
7:     if n > max_n then
8:         max_n ← n
9:         max_v ← v
10:    end if
11:    if cnt ≥ k then
12:        if max_v = 0 then
13:            max_v ← proposer's proposing value
14:        end if
15:        for i ← 1, k in A(j, k) do
16:            Send: accept_request(i,pn,max_v)
17:        end for
18:    end if
19: end procedure
```

By contrast, when an acceptor receives an *accept* request, it always accepts it unless it has already promised not to in the prepare phase.

Algorithm 4 Accept Phase - Acceptor Side

```
1: max_pn ← highest proposed proposal number seen
2: max_n ← highest accepted proposal number
3: max_v ← Corresponding value of max_n
4: procedure ACCEPT_REQUEST_HANDLER(n,v)
5:     if n > max_pn then
6:         max_pn ← n
7:         max_n ← n
8:         max_v ← v
9:         Reply: accepted()
10:    else
11:        Reply: rejected()
12:    end if
13: end procedure
```

6.2.3.3 Learn Phase

On the last phase, if a proposer receives *accepted* messages from a majority of acceptors he decides that the value he proposed is accepted, so he sends a *learn* message to all learner nodes.

Algorithm 5 Learner Phase - Proposer Side

```
1: v ← Value proposed in accept_request
2: cnt ← 0
3: procedure ACCEPTED_HANDLER
4:     cnt ← cnt + 1
5:     if cnt ≥ k then
6:         for each learner i in learners do
7:             Send: learn(i,v)
8:         end for
9:     end if
10: end procedure
```

6.2.4 Overview

A brief overview of the Paxos algorithm is shown in the following figure:

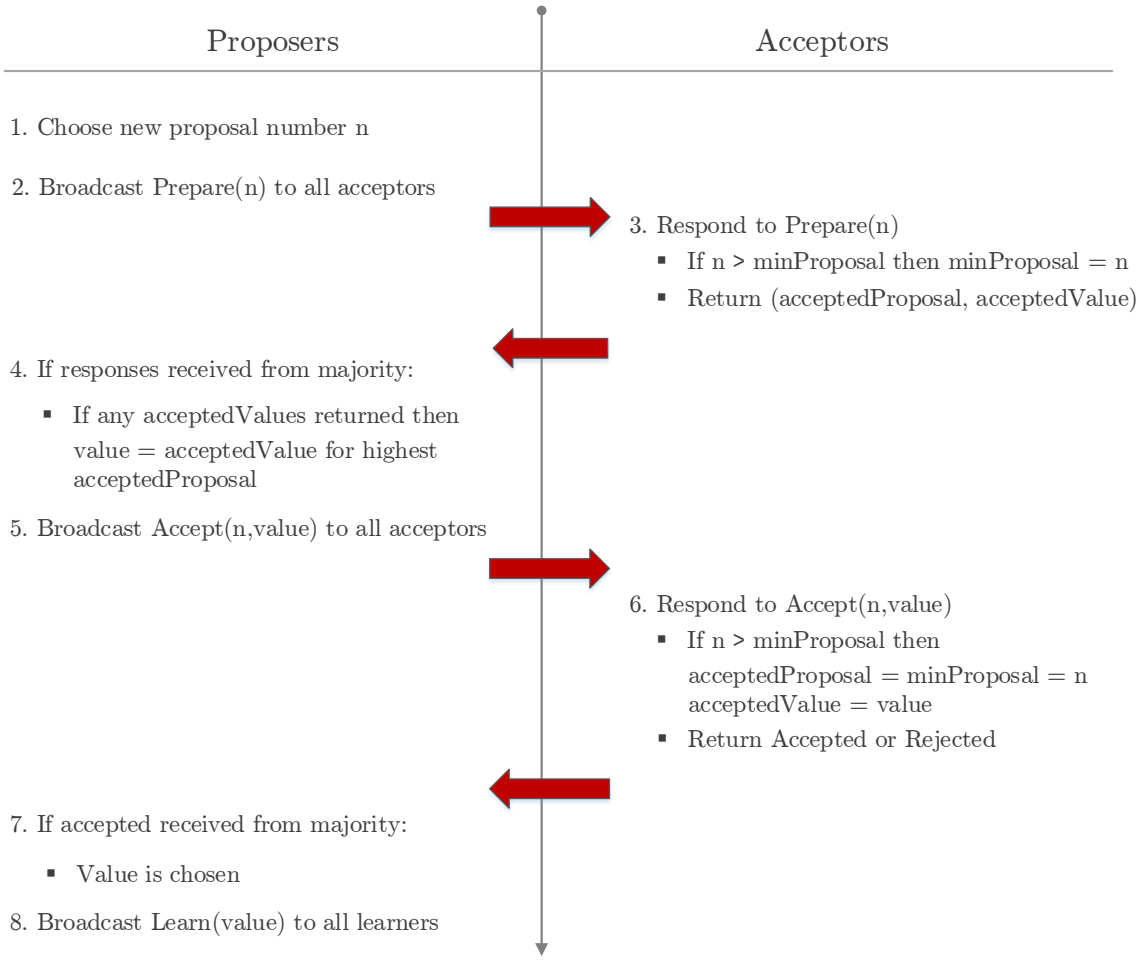


Figure 6.2: Overview of the Paxos algorithm.

6.2.5 Paxos By Example

In this section we will try to give a simple example of how the paxos algorithm works. In this example there are two proposers *P1* and *P2*, both making prepare requests. In addition, we have three acceptors *A1*, *A2* and *A3*. The request from proposer *P1* reaches acceptors *A1* and *A2* before the request from proposer *P2* but the request from *P2* reaches acceptor *A3* first.

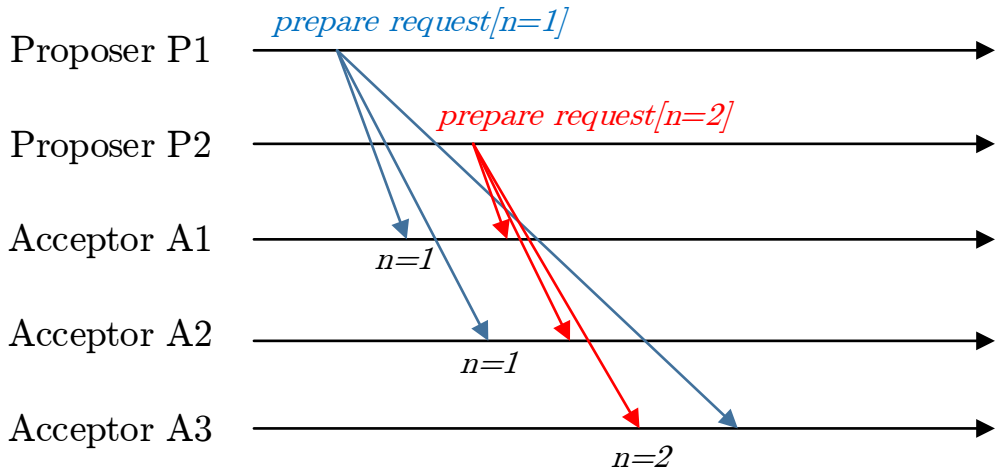


Figure 6.3: Prepare Phase: Proposers *P1* and *P2* each send prepare requests to every acceptor. Proposer *P1*'s request reaches acceptors *A1* and *A2* first, and proposer *P2*'s request reaches acceptor *A3* first.

If the acceptor receiving a prepare request has not seen another proposal, then he responds with a prepare response and promises never to accept another proposal with a lower proposal number. This is illustrated in 6.4, which shows the responses from each acceptor to the first prepare request they receive.

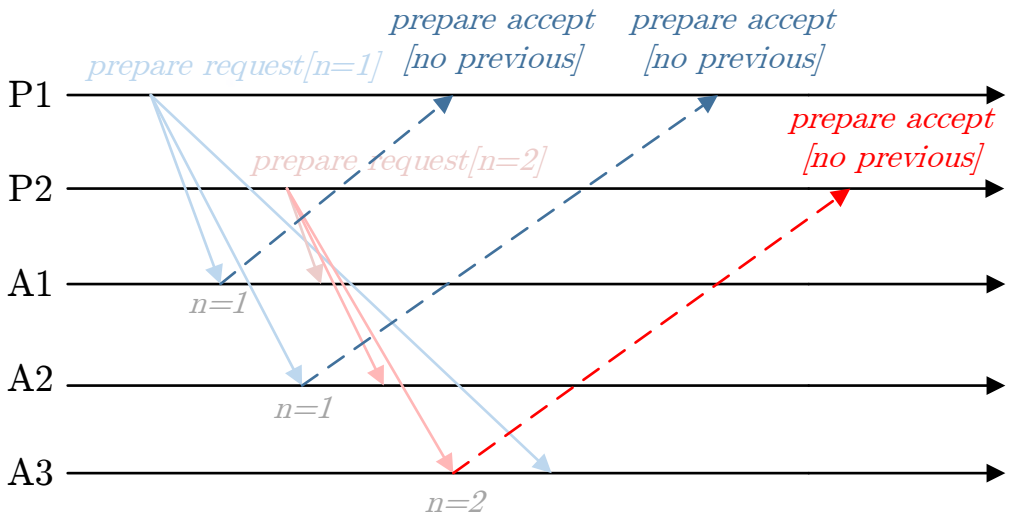


Figure 6.4: Prepare Phase: Each acceptor responds to the first *prepare request* message that it receives with a *prepare accept* message and a promise not to accept any requests numbered with less than n.

Eventually, acceptor *A3* receives proposer *P1*'s request, and acceptors *A1* and *A2*

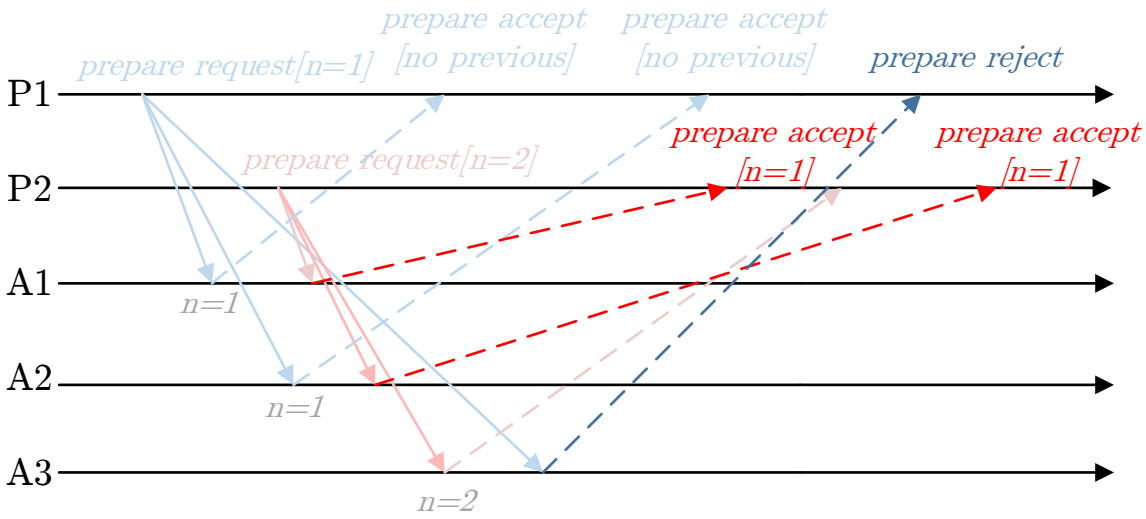


Figure 6.5: Prepare Phase: If the acceptor has not seen a request with a higher proposal number, the prepare request is rejected. Otherwise, he sends back the highest n he has seen.

receive proposer *P2*'s request. If the acceptor has already seen a request with a higher proposal number, the prepare request is ignored, as is the case with proposer *P1*'s request to acceptor *A2*. If the acceptor has not seen a higher numbered request, it again promises to ignore any requests with lower proposal numbers, and sends back the highest-numbered proposal that it has accepted along with the value of that proposal, if any. Otherwise, he sends back the highest n he has seen.

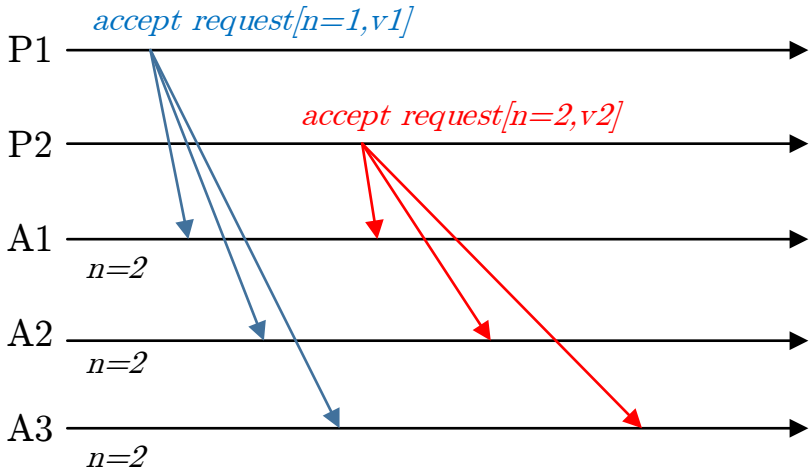


Figure 6.6: Accept Phase: Once a proposer receives acceptance from a majority of acceptors he then sends an *accept* message. Both *P1* and *P2* received replies from a majority of acceptors so they both send *accept* messages.

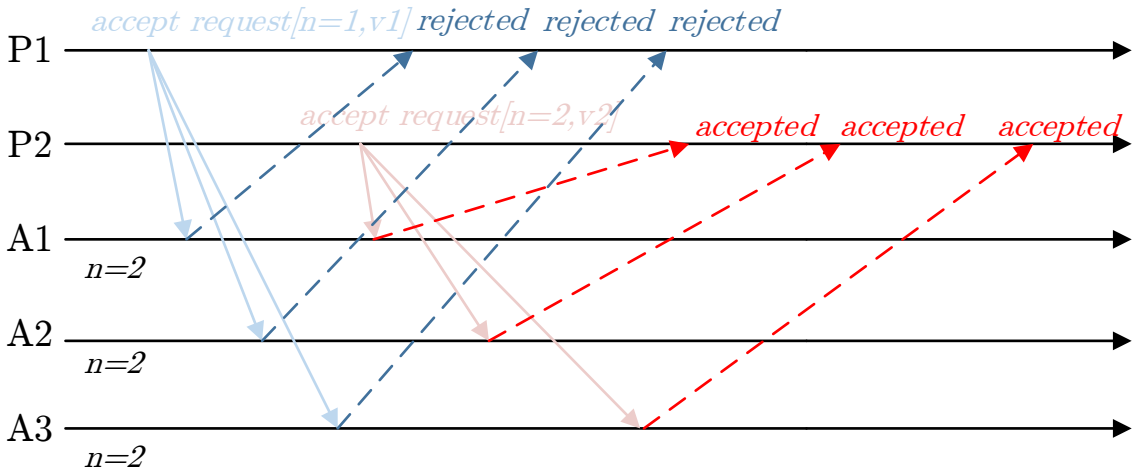


Figure 6.7: Accept Phase: requests from proposer *P1* are ignored by every acceptor.

Once a proposer has received prepare responses from a majority of acceptors he can issue an accept request. The acceptors haven't accepted any values yet, so proposer *P1* only received responses indicating that there were no previous proposals. Therefore he sends an accept request to every acceptor with the same proposal number as his initial proposal, thus $n = 1$, and a value v_1 that he proposes. Suchlike, proposer *P2* sends an accept request to each acceptor containing the proposal number it previously used ($n = 2$) and the value v_2 he proposes.

However, requests from proposer *P1* are ignored by every acceptor because they have all promised not to accept requests with a proposal number lower than 2 (in response

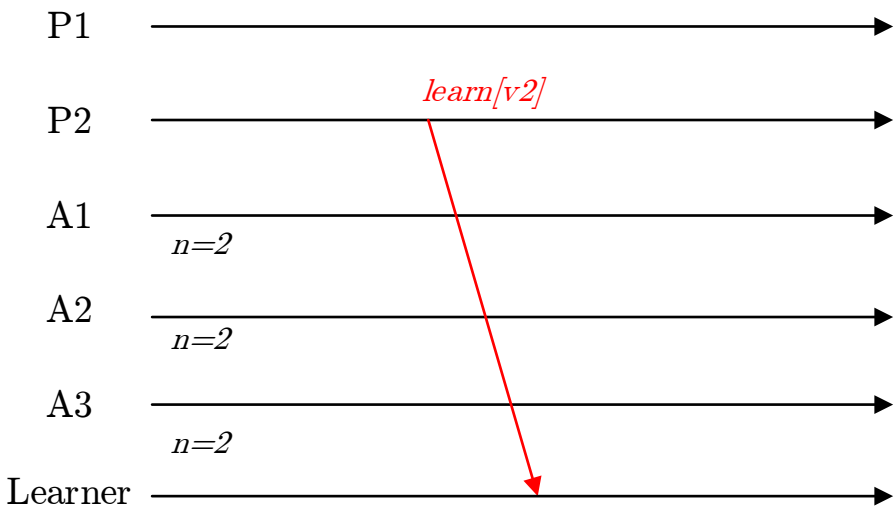


Figure 6.8: Learn Phase: Proposer *P2* received *accept* messages from majority of acceptors so he broadcasts the elected value to all learners.

to the prepare request from proposer *P2*). On the other hand, all requests from *P2* are accepted by every acceptor.

Lastly, proposer *P2*, after receiving an accepted message from a majority of acceptors, he then sends a *learn* message to all *learners* with the value chosen; in this case v_2.

Once a value has been chosen by Paxos, further communication with other proposers cannot change this value. If another proposer, proposer *P3*, sends a *prepare* request with a higher proposal number than has previously been seen, and a different value, i.e. $n = 3, v_3$, each acceptor responds with the previous highest proposal $[n = 2, v_2]$. This requires proposer *P3* to send an *accept* request containing $[n = 2, v_2]$, which only confirms the value that has already been chosen. Furthermore, if some minority of acceptors have not yet chosen a value, this process ensures that they eventually reach consensus on the same value.

Chapter 7

Applying Detectors and Paxos to DRTRM

Our implementation of the detectors as well as the Paxos algorithm was included into the source code of DRTRM. Thus, we used the exact same design decisions with DRTRM as described extensively in chapter 4 of [17] combined with some additions we made. Summarily, the following decisions were used:

1. We assume a coarse-grained, homogeneous, partially-sychronous system, where possible failures can only be caused by crashes (see chapters 1 and 3 for more details).
2. For the internal state of nodes we used exactly the same states as DRTRM. Three more states were added; **NEW_IDAG**, **NEW_MANAGER** and **PAXOS_ACTIVE**. Their use is described in section 7.3.
3. As far as the states of an application are concerned no modifications were made.
4. For the Inter-node communication, we further implemented the signals described in tables 7.1, 7.2, 7.4, 7.3 .
5. Deadlock prevention was completely changed as described in section 7.1.
6. Inter-node synchronization is reached through semaphores. These semaphores are also stored in shared memory so that processes can access semaphores of other nodes.

7.1 Deadlock Detection

As described in section 5.3 the DRTRM framework uses an interaction between nodes in order to prevent deadlocks. Even though this approach works in deadlock prevention between two nodes, in case of more complex deadlock scenarios, this implementation

is problematic. As mentioned in chapter 4 the best approach to handle deadlocks in distributed systems is *deadlock detection*. Assuming that the SCC platform is based on message passing communication, we can treat it as a distributed system, where each core is a process of the system. Thus, in order to handle deadlocks we applied to DRTRM the Chandy-Misra-Hass detection algorithm [23].

7.1.1 Chandy-Misra-Hass Detection Algorithm

7.1.1.1 Theory

The algorithm proposed by Chandy, Misra and Hass (1983) is considered an edge-chasing, probe-based algorithm (see sec. 4.1.3) and is considered one of the best deadlock detection algorithm for distributed systems. The algorithm consists of the following steps:

Step 1:If a process makes a request for a resource which fails or times out, the process generates a probe message and sends it to each of the processes holding one or more of its requested resources. Each probe message contains he following information:

- The *id* of the process that is blocked. That's the process that initiated the probe message.
- The *id* of the process that is sending this particular version of the probe message and
- The *id* of the process that should receive this probe message.

Step 2: When a process receives a probe message, it checks to see if it is also waiting for resources. If not, it is currently using the needed resource and will eventually finish and release the resource. If it is waiting for resources, it passes on the probe message to all processes it knows to be holding resources it has itself requested. The process first modifies the probe message, changing the sender and receiver ids.

Step 3: If a process receives a probe message that it recognizes as having initiated, it knows there is a cycle in the system and thus, **deadlock**.

In figure 7.1 we can see an example of how deadlock detection is achieved using this algorithm:

In this example, $P1$ initiates the probe message, so that all the following messages have $P1$ as the initiator. When the probe message is received by process P3, it modifies it and sends it to two more processes and so on. Eventually, the probe message reaches $P1$, thus, deadlock.

The algorithm of Chandy-Misra-Hass has many advantages which make it one of the best deadlock detection algorithms:

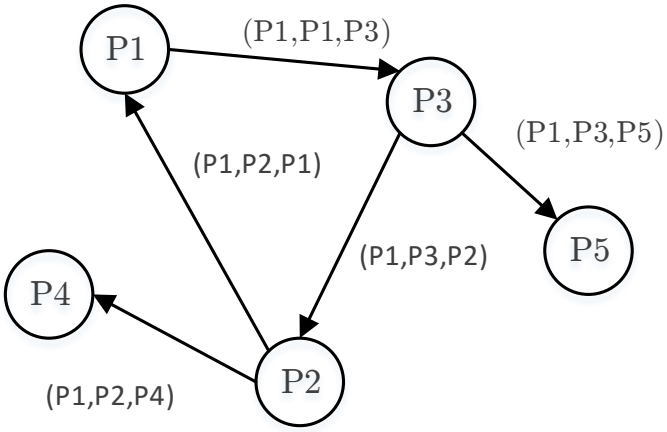


Figure 7.1: Example of Chandy-Misra-Hass deadlock detection algorithm

⊕ It is easy easy to implement.

⊕ Each probe message is of fixed length

⊕ There is very little computation

⊕ There is very little overhead

⊕ There is no need to construct a graph, nor to pass graph information to other processes.

⊕ It does not detect *phantom* deadlocks.

⊕ There is no need for special data structures.

7.1.1.2 Implementation

The deadlocks were basically happening when a core had to write data to the memory of another core. In such a situation the former locks a semaphore in order to access the memory exclusively. The semaphore is unlocked when the latter successfully reads the data from his memory. However, in a situation where node $P1$ waits for $P2$ who waits for $P3$ the semaphores will never be unlocked, leading to a deadlock. For that reason, before we wait on a semaphore, we first send a signal identical to that described in the previous section. If we do not receive any similar signal back we lock the semaphore and proceed to write to the memory of the core needed. For this implementation, all we need is an extra signal called $\langle SIG\_DLOCK\_DETECT \rangle$.

7.2 Failure Detection

As described in section 4.2 there are eight different classes of failure detectors. For the DRTRM framework we first implemented two different classes, the perfect failure detector $\mathcal{P}$ and the eventually perfect failure detector $\Diamond\mathcal{P}$ as presented in [11]. However, after consideration we also implemented a third failure detector called tweaked perfect failure detector $\mathcal{P}_t$, which is based on $\mathcal{P}$ but uses much less computational effort.

7.2.1 Perfect Failure Detector $\mathcal{P}$

7.2.1.1 Theory

For the perfect failure detector we assume the crash-stop process abstraction (see sec. 3.2.1) and that our system is synchronous. Thus, crahses can be accurately detected using *timeouts*. For example, if a process sends a message to another process, then if the former does not receive a responce within a time period equal to the worst-case delivery time then the latter would have eventually crashed. So, all we need to figure out is the maximum delivery time of messages.

The perfect failure detector $\mathcal{P}$ never changes its mind about failures and detections are permanent. In other words, once a process p is detected by some process q, the process p remains detected by q forever. Hence, $\mathcal{P}$ satisfies both *strong accuracy* and *strong completeness*. In algorithm 6 we present the operation of $\mathcal{P}$ (*check sec. 3.1.1 for notations*).

7.2.1.2 Implementation

As shown in algorithm 6 in order to implement $\mathcal{P}$ we need two extra signals and a timer. For the timer implementation we used POSIX timers. POSIX timers are provided by Linux for an efficient way to have a timer per process. A timer is set and upon its expiration, a signal is sent to the respective process. This signal is handled by the appropriate signal handling function as any other common signal. Inside this handling function, when a core detects another core as faulty, except for the Paxos initiation, it also checks if the failed core was inside his cluster and removes him from his DDS list, if the detector was a controller core. On the other hand, if the detector is a manager core he checks if the faulty core was a worker of him. If yes, he reappoints the workload of the failed core to another worker of his.

To present a closest to real-life implementation we do not start all timers at the same time. Instead, we set an initial delay of $rand(node\_id \mod 10)$ and after that initial delay we set the delay Δ of the failure detector.

Algorithm 6 Perfect Failure Detector $\mathcal{P}$

```
1: procedure PFD_INIT
2:     alive := Π
3:     detected := ∅
4:     starttimer(Δ)
5: end procedure

6: procedure TIMEOUT_HANDLER
7:     for each p in Π do
8:         if (p ∉ alive) ∧ (p ∉ detected) then
9:             detected := detected ∪ {p}
10:        end if
11:        Send: heartbeat_request(q,p)
12:    end for
13:    alive := ∅
14:    starttimer(Δ)
15: end procedure

16: procedure HEARTBEAT_REQUEST_HANDLER(q,p)
17:    Send: heartbeat_reply(p,q)
18: end procedure

19: procedure HEARTBEAT_REPLY_HANDLER(p,q)
20:    alive := alive ∪ {p}
21: end procedure
```

As far as the signals are concerned we added two signals: $\langle SIG\_HEARTBEAT\_REQ \rangle$ and $\langle SIG\_HEARTBEAT\_REP \rangle$ the handlers of whom are shown in algorithm 6.

7.2.2 tweaked Perfect Failure Detector $\mathcal{P}_t$

7.2.2.1 Theory

Since messages are sent between timer explotions, one improvement that could be made is to send a HEARTBEAT request only if we have not received any message from a node during the time Δ. This will decrease the overhead of messages of $\mathcal{P}$. In algorithm 7 we see the pseudocode of the tweaked Perfect Failure Detector

Algorithm 7 tweaked Perfect Failure Detector $\mathcal{P}_t$

```
1: procedure TPFD_INIT
2:     alive := Π
3:     detected := ∅
4:     suspected := ∅
5:     starttimer(Δ)
6: end procedure

7: procedure TIMEOUT_HANDLER
8:     for each p in Π do
9:         if (p ∈ suspected then
10:            detected := detected ∪ {p}
11:        end if
12:        if (p ∉ alive) ∧ (p ∉ detected) ∧ (p ∉ suspected) then
13:            suspected := suspected ∪ {p}
14:            Send: heartbeat_request(q,p)
15:        end if
16:    end for
17:    alive := ∅
18:    starttimer(Δ)
19: end procedure

20: procedure HEARTBEAT_REQUEST_HANDLER(q,p)
21:    Send: heartbeat_reply(p,q)
22: end procedure

23: procedure ANY_SIGNAL_RECEIVED_HANDLER(p,q)
24:    alive := alive ∪ {p}
25:    if (p ∈ suspected) then
26:        suspected := suspected \ {p}
27:    end if
28: end procedure
```

7.2.2.2 Implementation

For implementation, the same principles as $\mathcal{P}$ were used. An extra addition was needed in the code to update the suspected list any time we received a message.

7.2.3 Eventually Perfect Failure Detector $\Diamond\mathcal{P}$

7.2.3.1 Theory

For the eventually perfect failure detector we assume the crash-stop process abstraction (see sec. 3.2.1) and that our system is partially synchronous. An eventually perfect failure detector detects crashes accurately after some time, but may make mistakes before that time. This happens due to the fact that timeout delays have to be adjusted so they can lead to correctly detected crashes.

Algorithm 8 Eventually Perfect Failure Detector $\Diamond\mathcal{P}$

```
1: procedure EPFD_INIT
2:     alive := Π
3:     suspected := ∅
4:     delay := Δ
5:     starttimer(Δ)
6: end procedure

7: procedure TIMEOUT_HANDLER
8:     if alive ∩ suspected ≠ ∅ then
9:         delay := delay + Δ
10:    end if
11:    for each p in Π do
12:        if (p ∉ alive) ∧ (p ∉ suspected) then
13:            suspected := suspected ∪ {p}
14:        else if (p ∈ alive) ∧ (p ∈ suspected) then
15:            suspected := suspected \ {p}
16:        end if
17:        Send: heartbeat_request(q,p)
18:    end for
19:    alive := ∅
20:    starttimer(Δ)
21: end procedure

22: procedure HEARTBEAT_REQUEST_HANDLER(q,p)
23:    Send: heartbeat_reply(p,q)
24: end procedure

25: procedure HEARTBEAT_REPLY_HANDLER(p,q)
26:    alive := alive ∪ {p}
27: end procedure
```

More specifically, an eventually perfect failure detector also uses timeouts, but in this case increasing ones, and suspects processes that did not send heartbeat messages within this time delay. The duration of the timeout is crucial if we need a quick detection. Obviously, a suspicion may be wrong in a partially synchronous system. A process p may suspect a process q, even if q has not crashed, simply because the timeout delay chosen by p to suspect the crash of q was too short. In this case, p's suspicion about q is false.

However, p can change its suspicion if he receives a message from q. Algorithm 8 shows the pseudocode of $\Diamond\mathcal{P}$.

7.2.3.2 Implementation

As for both $\mathcal{P}$ and $\mathcal{P}_t$ we also used here a posix timer and two signals as described in 7.2.1.2.

7.3 Paxos

For the Paxos algorithm, as a first step we created the necessary signal handlers for the algorithm to work. That is to say, signals regarding the prepare and accept phases of the algorithm. A Paxos instance is ran inside the cluster of the failed core in case of a controller failure or among the workers of a manager, inside of a manager failure. Thus, all signals regarding Paxos itself are sent between cores in the same cluster except for $\langle SIG\_LEARN \rangle$ which is spread throughout the platform in order for all the cores to learn the outcome of Paxos. This is shown in figure 7.2

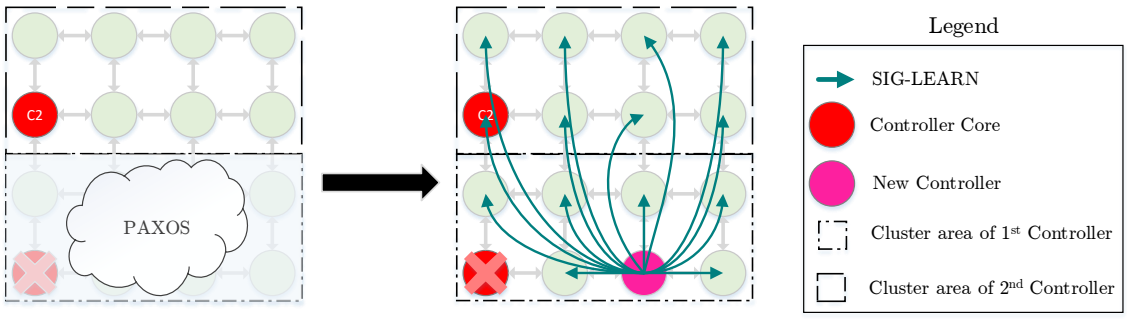


Figure 7.2: A Paxos instance is ran inside a single cluster, whereas the SIG_LEARN is spread throughout the platform

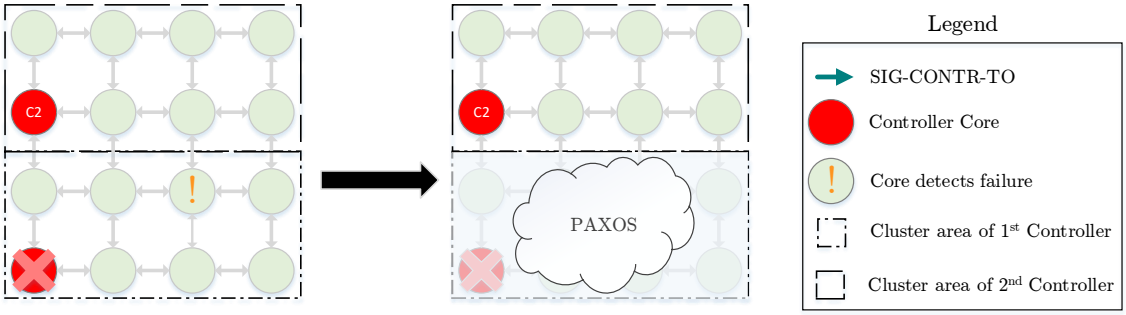
However, making Paxos work at its own was not enough. We had to do further modifications on the code for DRTRM to work properly after a Paxos instance. In detail, we had to overcome issues regarding cases of failures for all different types of core nodes, meaning *controllers*, *managers* and *workers*. For that purpose, a very important modification was the addition of a feature to the cores list of each manager in order to keep the workload of each worker core. This will help us to recover in several failure scenarios that will be stated below.

7.3.1 Controller Failure

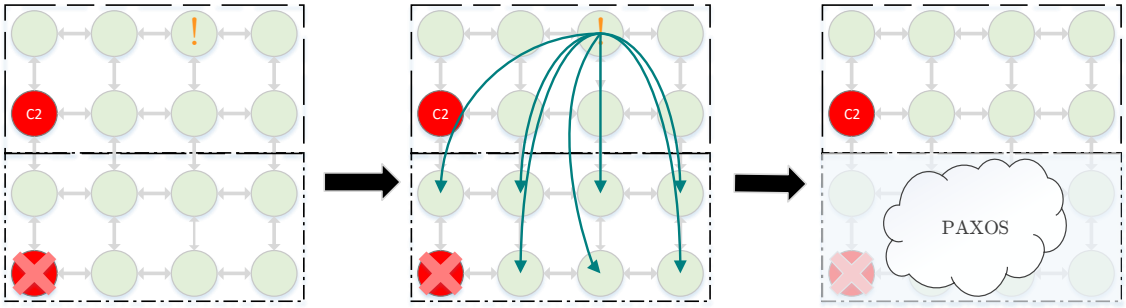
In case of a controller failure, there are two possible scenarios:

(1) The crashed controller core is in the same cluster as the core that detected the failure. In this case, the core that detects the failure sends a $\langle SIG\_CONTR\_TO \rangle$ to all cores inside the cluster of the crashed controller so they can start a Paxos instance.

(2) The crashed controller core is in a different cluster as the core that detected the failure. In this case, a Paxos instance is initiated as long as a core realizes that his controller has failed.



(a) A core inside the failed controller's cluster detects the failure and a Paxos instance is began



(b) A core outside the failed controller's cluster detects the failure so he sends a SIG_CONTR_TO signal to all cores inside C1's cluster

Figure 7.3: Failure Detection Scenarios

After this point, the workaround of each scenario is the same as described in chapter 6 and a new controller will eventually be elected. When the new controller is elected we have to ensure the correctness of DRTRM before proceeding.

First of all, the new controller has to create his *controller_list* and *DDS* list (see sec. 5.1.1). Thus, when a controller on another cluster receives a $\langle SIG\_LEARN \rangle$ from the new controller he replies with $\langle SIG\_LEARN\_ACK\_CONTR \rangle$. That way, the new controller will eventually find out all the other controllers to create his *controller_list*. In similar, when a manager receives a $\langle SIG\_LEARN \rangle$ he checks his *core_list*. If he owns a core inside the cluster of the new controller he replies with $\langle SIG\_ADD\_TO\_DDS \rangle$, so the new

controller can add him to his *DDS* list. Last but not least if any manager was waiting for offers (that is his state was **IDLE_AGENT_WAITING_OFFERS** he changes his state to **IDLE_AGENT**. These are shown in figure 7.4

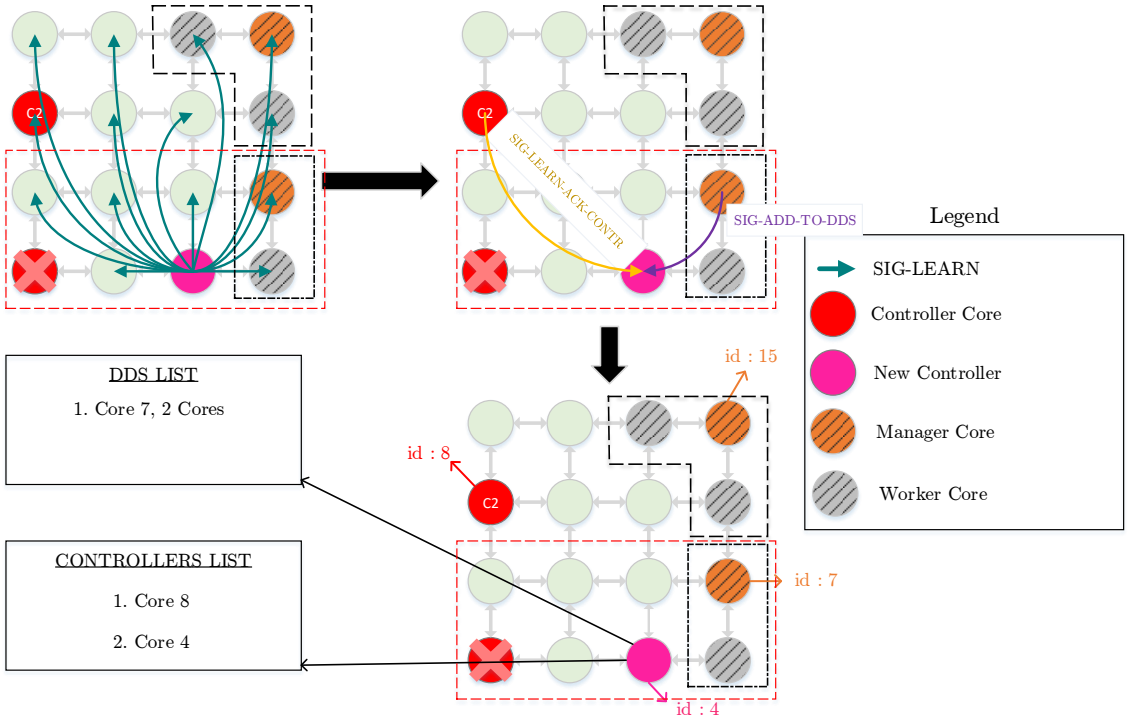


Figure 7.4: Update of DDS and controllers list in case of a controller failure. Controllers and managers inform the new controller by sending a SIG_LEARN_ACK_CONTR and SIG_ADD_TO_DDS respectively.

Secondly, we have to check which was the previous state of the new controller. Three possible roles are available: *manager*, *worker* and *idle* core, because the new controller is elected from inside the cluster of the previous one and controllers in each cluster are unique.

In practice, the new controller can never be a manager. In case a manager receives a *prepare-accept* from a majority of acceptors, he does not propose himself as the new controller. Instead, he proposes a worker of his as shown in figure7.6.

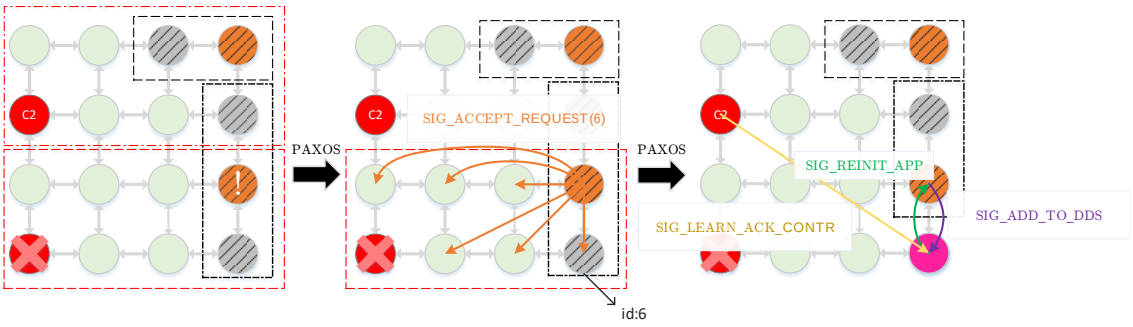


Figure 7.5: Workaround when new controller was a manager.

In case the new controller was a worker core, then he has to reassign the workload he was executing to a new core. Thus, he sends a $\langle SIG\_REINIT\_APP \rangle$ to edw kati so that he can find a new worker core for this workload.

Finally, in case the new controller was an idle core nothing more has to be done. Both *DDS* and *controller* lists are already created previously. After ensuring the above are fulfilled, the state of the new controller is changed to **NEW_IDAG** and framework continues to work properly.

7.3.2 Manager Failure

In order to manage the manager failure we added a new type of list to each worker called the *coworkers list*. Each time a manager sends a $\langle SIG\_APPOINT\_WORK \rangle$ signal in order to assign workload to his worker cores, he also informs them who their coworkers are. On their side, the workers save the ids of their coworkers in the coworkers list mentioned above. This is very important in order to recover from a manager failure.

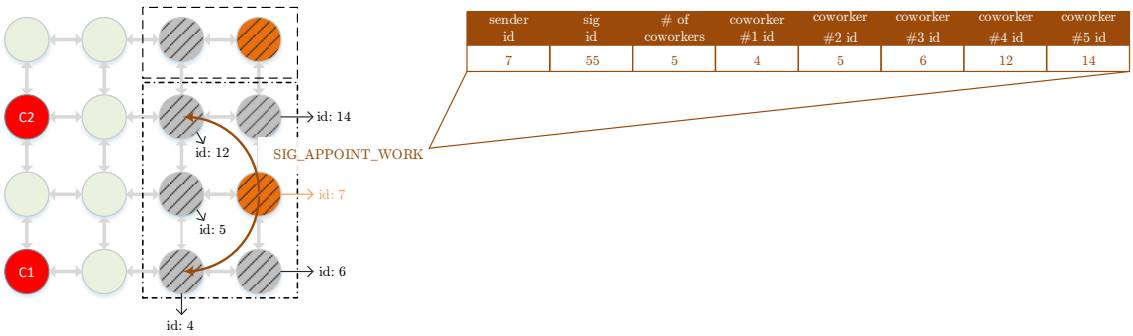


Figure 7.6: When a manager sends a $\langle SIG\_APPOINT\_WORK \rangle$ he also sends the coworkers of the node.

So, in case of a manager failure all cores will eventually detect the failure with $\mathcal{P}$ or $\Diamond\mathcal{P}$. Once they do, they take the appropriate actions based on their type, meaning *controller*,

manager or *worker*.

If a worker realizes that his manager has failed, he initiates a Paxos instance. The Paxos algorithm is ran between the workers of the manager and the new manager will be one of the current workers. As we can see, the coworkers list helps us here in order to begin the Paxos algorithm as we can easily find which nodes we will send the $\langle SIG\_PREPARE\_REQUEST \rangle$ signal to. As soon as the new manager is elected, a $\langle SIG\_LEARN \rangle$ is spread throughout the platform. In addition, the new manager has to find out the remaining workload of the application, as well as other information regarding the application itself. For that purpose, he scans the log file of the application (as described in section 5.1.2) in order to find out the information he needs.

If a controller realizes that a manager core has timed out, he checks his *DDS* list to see if that manager was owning cores inside his cluster. If he was, he removes him from his *DDS* list. Additionally, if the manager was inside his cluster, he also has to remove him from his *cores* list. Lastly, when a controller receives a $\langle SIG\_LEARN \rangle$ he updates his DDS list again, by adding the new manager to it, if he has to.

If a manager finds out that another manager has failed, no action has to be taken. The manager continues to work as before. The above are shown in figure 7.7

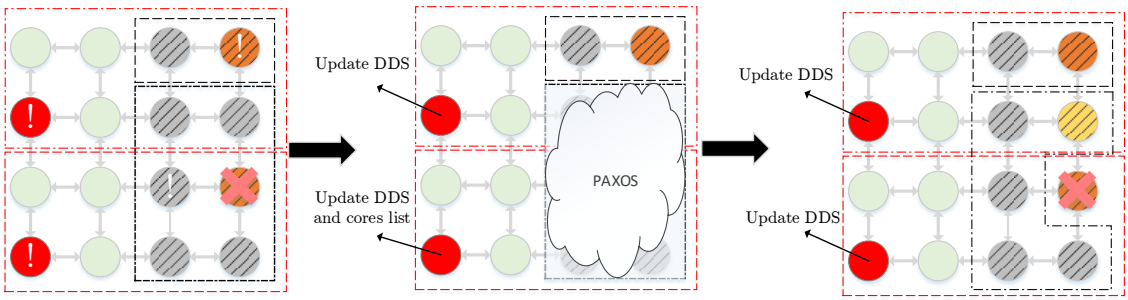


Figure 7.7: Workaround when a manager fails.

7.3.3 Worker Failure

In case of a worker failure, there is no need to run a Paxos instance. A manager will eventually realize that a worker he owns has crashed (using either $\mathcal{P}$ or $\Diamond\mathcal{P}$). As soon as he finds out, he checks his *core list* to determine which worker is currently not doing any job and reappoints the workload of the failed node to this core.

7.4 Signals Summary

In the following tables we summarize all the signals used to implement the detectors as well as the Paxos algorithm and ensure the correct operation of the framework after the failure:

Table 7.1: Deadlock Detection Signals

Name	Description
$\langle SIG\_CONTR\_TO \rangle$	If a core detects that a controller of another cluster C_j has crashed, it sends this signal to all the cores inside C_j in order to start a Paxos instance.

Table 7.2: Failure Detection Signals

Name	Description
$\langle SIG\_HEARTBEAT\_REQ \rangle$	If the timeout period Δ and a core is suspected to have crashed, this signal is sent in order to check the state of the core.
$\langle SIG\_HEARTBEAT\_REP \rangle$	If a core receives a $\langle SIG\_HEARTBEAT\_REQ \rangle$ he replies with this signal.
$\langle SIG\_CONTR\_TO \rangle$	If a core detects that a controller of another cluster C_j has crashed, it sends this signal to all the cores inside C_j in order to start a Paxos instance.

Table 7.3: Recovery after Paxos

Name	Description
$\langle SIG\_LEARN\_ACK\_CONTR \rangle$	Once a controller from a different cluster learns that a new controller has been elected he sends this signal to the new controller so that he can configure his *idag array*.
$\langle SIG\_ADD\_TO\_DDS \rangle$	Once a manager receives $\langle SIG\_LEARN \rangle$ from a core that is inside a cluster, in which he owns cores he sends this signal to the new controller so that he can update his DDS list.
$\langle SIG\_REINIT\_APP \rangle$	If the new controller was a worker core, he sends this message to his manager core in order to reassign the workload to another worker.

Table 7.4: Paxos Signals

Name	Description
$\langle SIG\_PREP\_REQ \rangle$	Once a core realizes that his controller or manager has crashed, he picks a proposal number n and sends this signal to all corresponding cores along with n, thus becoming a *proposer*, p.
$\langle SIG\_PREP\_ACC\_NO\_PREV \rangle$	If an *acceptor* has not accepted any values yet he replies with this message to the $\langle SIG\_PREP\_REQ \rangle$ of p.
$\langle SIG\_PREP\_ACC \rangle$	If an *acceptor* has accepted a value, he replies with this message to the $\langle SIG\_PREP\_REQ \rangle$ sent by p, along with the highest proposal number accepted and the corresponding accepted value.
$\langle SIG\_ACC\_REQ \rangle$	Once a proposer has received $\langle SIG\_PREP\_ACC \rangle$ and $\langle SIG\_PREP\_ACC\_NO\_PREV \rangle$ from a quorum of acceptors, he sends this signal along with his proposal number n and his proposing value as described in section 6.2.3.2.
$\langle SIG\_ACCEPTED \rangle$	If the acceptor has not seen a higher proposal number than n he accepts the proposed value and sends this signal to p.
$\langle SIG\_LEARN \rangle$	As long as p receives $\langle SIG\_ACCEPTED \rangle$ from a quorum of acceptors, his proposed value is chosen so he broadcasts the chosen value to all cores.

Chapter 8

Theoretical and Experimental Results

Before proceeding, we first denote the following symbols which will be used for the theoritical analysis of our implementation:

Table 8.1: Definition Table

Denotement	Description
Δ	Delay of Failure detectors as denoted in 7.2.
T	Time until all applications have finished.
N	Number of nodes in the platform.
K	Number of clusters which equals with the number of controllers.
n_i	i_{th} node.
c_i	If i_{th} node is a controller core then $c_i = 1$ else $c_i = 0$.
m_i	If i_{th} node is a manager core then $m_i = 1$ else $m_i = 0$.
n_{ik}	If node n_i belongs to cluster k then $n_{ik} = 1$ else $n_{ik} = 0$
w_{ji}	If node j is worker of node i then $w_{ji} = 1$ else $w_{ji} = 0$

8.1 Detectors Overhead

In all scenarios we use either the Perfect Failure Detector $\mathcal{P}$ or the Eventually Perfect Failure Detector $\Diamond\mathcal{P}$ which we described in 7.2. In each case, each core sends a $\langle SIG\_HEARTBEAT\_REQ \rangle$ to all other cores and waits for a $\langle SIG\_HEARTBEAT\_REP \rangle$ every Δ seconds.

When using either $\mathcal{P}$ or $\Diamond\mathcal{P}$ the overhead of our implementation in terms of messages exchanged is:

$$\Big(\tfrac{T}{\Delta}\Big)\Big(2 \times \big(N-1\big) \times \big(N-1\big)\Big) = 2\big(\tfrac{T}{\Delta}\big)\big(N-1\big)^2$$

For $\mathcal{P}$ this overhead is always the same. On the other hand, for $\Diamond\mathcal{P}$ this is the worst case scenario where each time the timer explodes there is at least one suspected core.

8.2 Scenarios

8.2.1 Controller Timeout

For the case of a controller timeout we denote:

$$N_k = \sum_{i=0}^{N} n_{ik}$$

as the number of cores inside the k_{th}, cluster as well as:

$$M = \sum_{i=0}^{N} m_i$$

as the total number of managers in the platform. As described in chapter 8.3 in a controller timeout there are two possible workarounds:

1. **A core outside the cluster f of the failed controller detects the failure:** In this case the core that detected has to send a $\langle SIG\_CONTR\_TO \rangle$ to all cores inside f except the failed node c_f. In the worst case where every node outside the cluster detects the failure, then

   $$\big(N - N_f\big) \times \big(N_f - 1\big)$$

   messages are exchanged.

2. **A core inside the cluster f of the failed controller detects the failure:** In this case no messages are sent and a PAXOS instance is initiated.

For PAXOS, each core that received a $\langle SIG\_CONTR\_TO \rangle$ or detected the failed controller. sends a $\langle SIG\_PREPARE\_REQ \rangle$ to all other nodes inside the cluster except for the controller. Similarly with above that is:

$$\big(N_f - 1\big) \times \big(N_f - 1\big) = \big(N_f - 1\big)^2$$

messages. In the worst case scenario where the messages arrive in order compared to the proposal number then each core will accept the *prepare_request* and reply with a $\langle SIG\_PREPARE\_ACC\_NO\_PREV \rangle$. That is an additional $\big(N_f - 1\big)^2$ messages.

In the above case, all nodes will receive a majority of responses in their *prepare_request*. So they are all going to broadcast a $\langle SIG\_ACC\_REQ \rangle$ to all nodes inside the cluster which makes us another $\big(N_f - 1\big)^2$ messages. However, only one of this requests will be accepted,

the one with the higher proposal number. The other will be rejected resulting in $\big(N_f - 1\big)$ messages. When the new controller finally receives $\langle SIG\_ACCEPTED \rangle$ from a majority of acceptors he spreads a $\langle SIG\_LEARN \rangle$ throughout the platform. Summing the above, we find that the PAXOS overhead in terms of messages exchanged is:

$$\big(N_f - 1\big)^2 + \big(N_f - 1\big)^2 + \big(N_f - 1\big)^2 + \big(N_f - 1\big) + \big(N - 1\big) =$$

$$3\big(N_f - 1\big)^2 + N_f + N - 2$$

After the new controller has been elected, each other controller sends a $\langle SIG\_LEARN\_ACK\_CONTR \rangle$. Thus, $\big(K - 1\big)$, because each cluster has one controller.

Similarly, if a manager utilizes a core inside cluster f he sends a $\langle SIG\_ADD\_TO\_DDS \rangle$ to the new controller. Assuming that every manager utilizes at least one core inside f, that makes us an additional M messages. Finally, if the new controller was a worker core he sends a $\langle SIG\_REINIT\_APP \rangle$ to his manager in order to reappoint the remaining workload.

Summarizing, in case of a controller timeout the worst case overhead in terms of message exchanged in order to return to stability is:

$$\big(N - N_f\big) \times \big(N_f - 1\big) + 3\big(N_f - 1\big)^2 + N_f + N - 2 + \big(K - 1\big) + M + 1 =$$

$$\mathcal{O}\Big(\big(N_f\big)^2\Big)$$

As we see, for our implementation the size of the cluster is the value which affects the most the performance of the protocol.

8.2.2 Manager Timeout

For the manager failure scenario we denote:

$$W_i = \sum_{j=0}^{N} w_{ji}$$

as the number of workers of node i. In a manager failure, a PAXOS instance is initiated only when a worker finds out that his manager f has failed. We found the overhead of PAXOS in the previous section. By replacing N_f with the number of workers of the failed manager we result in:

$$\big(W_f\big)^2 + \big(W_f\big)^2 + \big(W_f\big)^2 + W_f + N = 3\big(W_f\big)^2 + W_f + N =$$

$$\mathcal{O}\Big(\big(W_f\big)^2\Big)$$

After the new manager has been elected, no more messages have to be sent.

8.2.3 Worker Timeout

In case of a worker timeout there is no actual overhead. Only one additional message is sent from the manager of the failed worker to one of his other workers in order to reappoint the workload.

8.3 Experimental Setup

In this section, we present our experimental results on the NoC simulator. As mentioned again in chapter the code of detectors and Paxos was included into the source code of DRTRM. The setup of the simulator in order to imitate a real MPSoC was:

- Each core runs as a distinct process. This is achieved by the following procedure; When we run the NoC simulator executable, there is an initial process triggered. This process acts as node 0 and is always a controller core in every simulation. This node forks all other controller cores. Subsequently, all controller cores fork the nodes they are responsible for. This is shown in figure 8.1

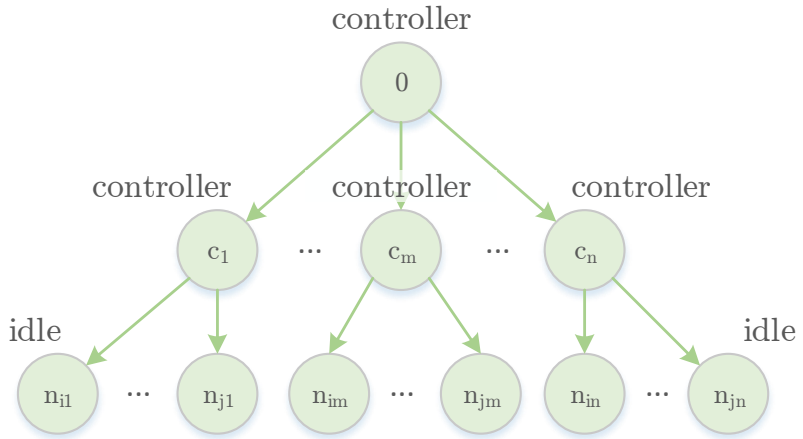


Figure 8.1: Fork nodes in NoC simulator

- For the implementation of $\mathcal{P}$, $\mathcal{P}_t$ and $\Diamond\mathcal{P}$ a POSIX timer was used at each process. To present a closest to real-life implementation we do not start all timers at the same time. Instead, we set an initial delay of $rand(node\_id \mod 10)$ and then set the original delay Δ.
- As described in section 1.2.4, the SCC platform uses a Message Passing Buffer to

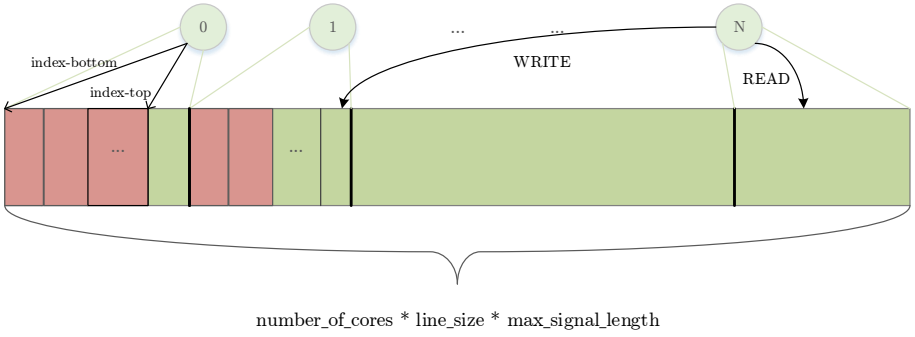
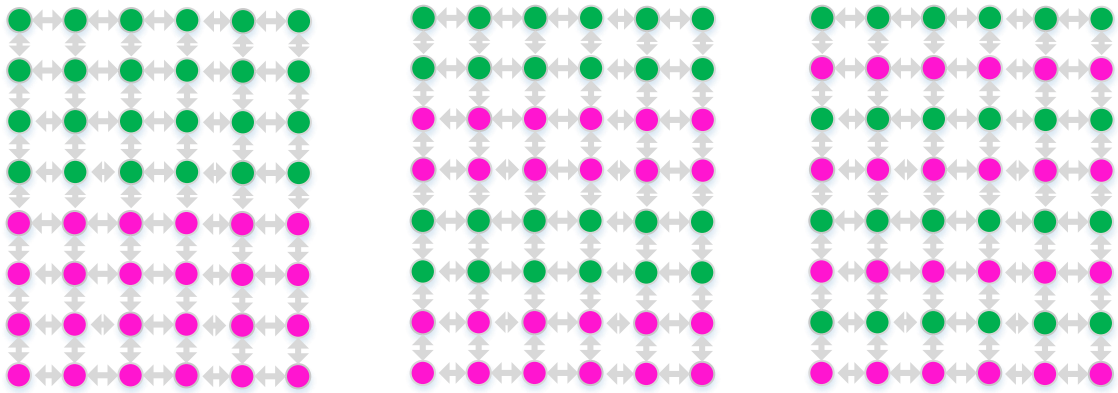


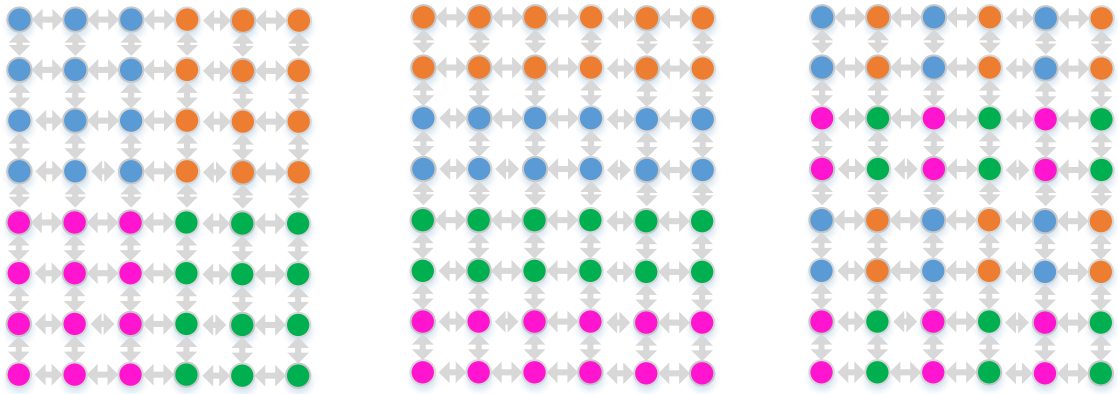
Figure 8.2: Memory in Noc Simulator

allow cores to communicate with each other. Thus, in order to give an accurate representation of the platform in the simulator we allocate a block of memory equal to $number\_of\_cores \times line\_size \times max\_signal\_length$. Each core keeps a pointer to the beginning and the end of the block that belongs to him. Messages are written to this block of memory in a Round-Robin way. Each core can write to any space in this memory, but can read only from his own. This is shown in figure 8.2. Index top keeps track of the most recent message in the memory, whereas index bottom keeps track of the older message in the message, which is the message to be processed next.

For almost all graphs we executed the corresponding measurements 10 times and the results presented are the average outputs. The grid size we used was $6x8$ with different topologies, some of which are shown in the following figure:

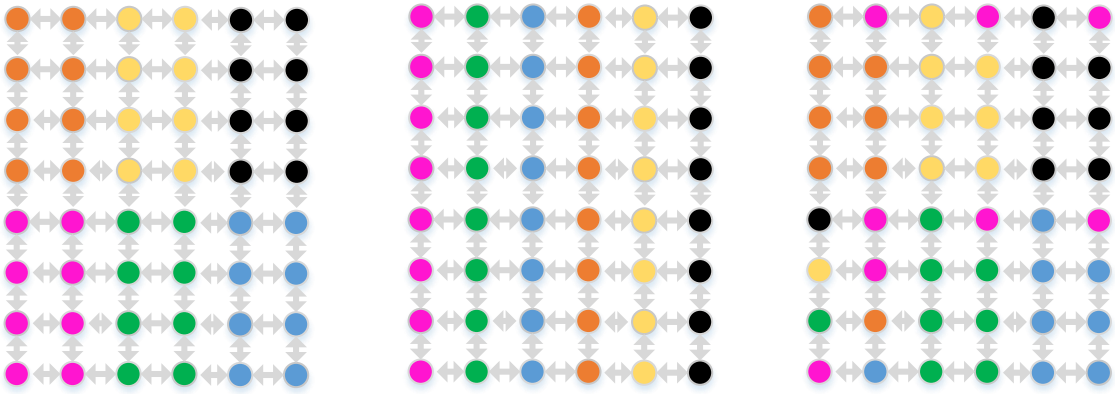


(a) 2 controllers sample grids

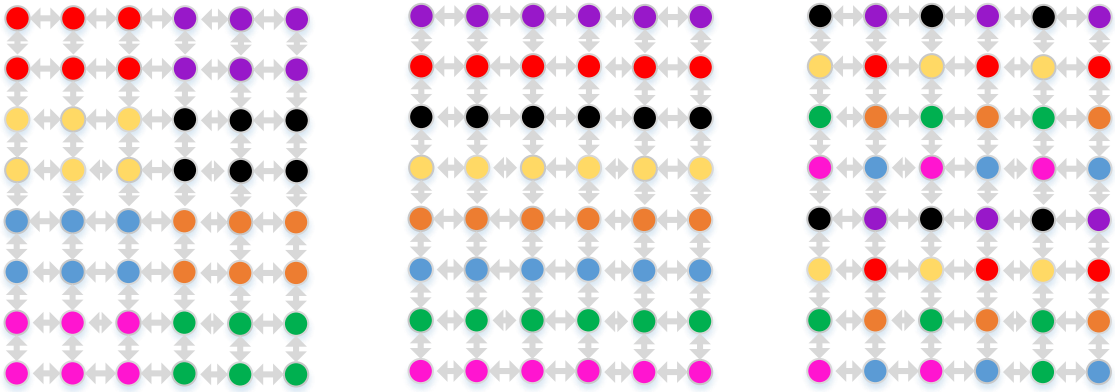


(b) 4 controllers sample grids

Finally, all simulations were made in a system composed by an Intel Xeon Processor E5-2658 v3 with 12 cores and 24 threads as well as 30MB of cache memory. This setup presents a closer approximation to a real MPSoC due to the fact that the processes are spread throughout more cores. In addition, the high capacity of the cache memory results in less context switches inside the memory. Thus, the delays of the memory access are



(c) 6 controllers sample grids



(d) 8 controllers sample grids

Figure 8.3: Different grid formations

closer to those of an on-Chip system.

8.4 Results

8.4.1 Different failure detectors

In figure 8.4 we can see the average $log_{10}(messagecount)$ vs. the cluster size. In this graph we can see that the message count for $\mathcal{P}$ is almost identical to the DRTRM message count, or even sometimes greater. That is the main reason we implemented the tweaked Perfect failure detector $\mathcal{P}_t$. In this graph we also confirm our theoretical results that Paxos message count is proportional to the cluster size, but we will show detailed information in later figures. As far as $\Diamond\mathcal{P}$ is concerned, the results are identical to those of $\mathcal{P}$. This happens because the upper bound delay in communications on chip is much less than 4 seconds, which is the lowest delay we measured. In cases where we used delays that would make a difference between these two, we experienced memory overflow issues because messages were written faster than they were read and processed.

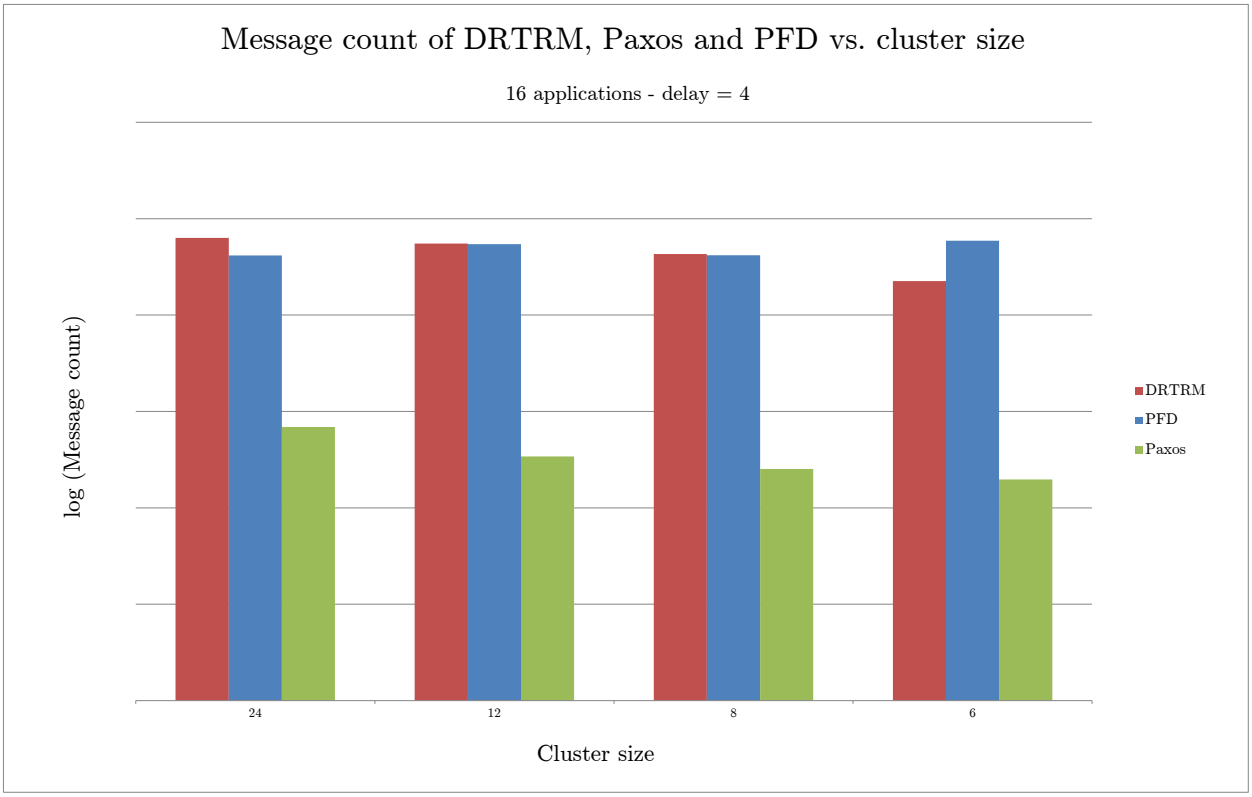


Figure 8.4: DRTRM, Paxos and PFD message count vs. cluster size

In figure 8.5 we see a comparison between $\mathcal{P}$ and $\mathcal{P}_t$. More specifically, we present the average message count per second for different Δ in the Failure Detector algorithm. As we can see $\mathcal{P}$ uses 3 times more messages than $\mathcal{P}_t$ in both 16 and 32 application. In addition, for more applications $\mathcal{P}$ exchanges more messages per second, whereas $\mathcal{P}_t$ exchanges less. This happens because when more applications are executed in the platform, more cores

are utilized, thus leading in more DRTRM messages exchanged between cores. As a result, $\mathcal{P}_t$ has to send less messages than $\mathcal{P}$.

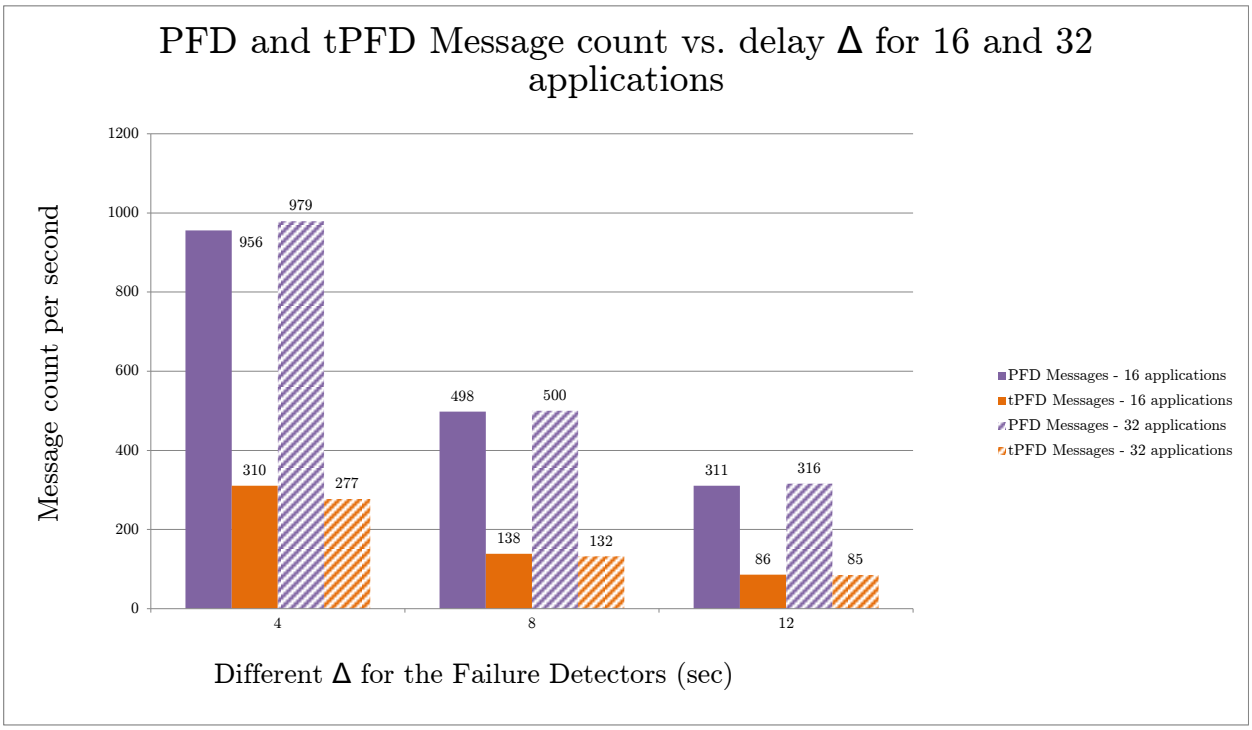


Figure 8.5: Comparison between $\mathcal{P}$ and $\mathcal{P}_t$

Although $\mathcal{P}_t$ seems better than $\mathcal{P}$, there is a drawback in the delay of failure detection and stability. As we see in figures 8.6 and 8.7 $\mathcal{P}_t$ takes more time to detect the failure and thus return to stable state. This happens because practically $\mathcal{P}_t$ needs 2Δ time to detect the failure. In the first Δ seconds it detects if a message has been exchanged with another core, and if not it sends a $\langle HEARTBEAT\_REQ \rangle$ signal to the suspected core. In the next Δ seconds it waits for a $\langle HEARTBEAT\_REP \rangle$. If it does not receive one, it detects the core as faulty.

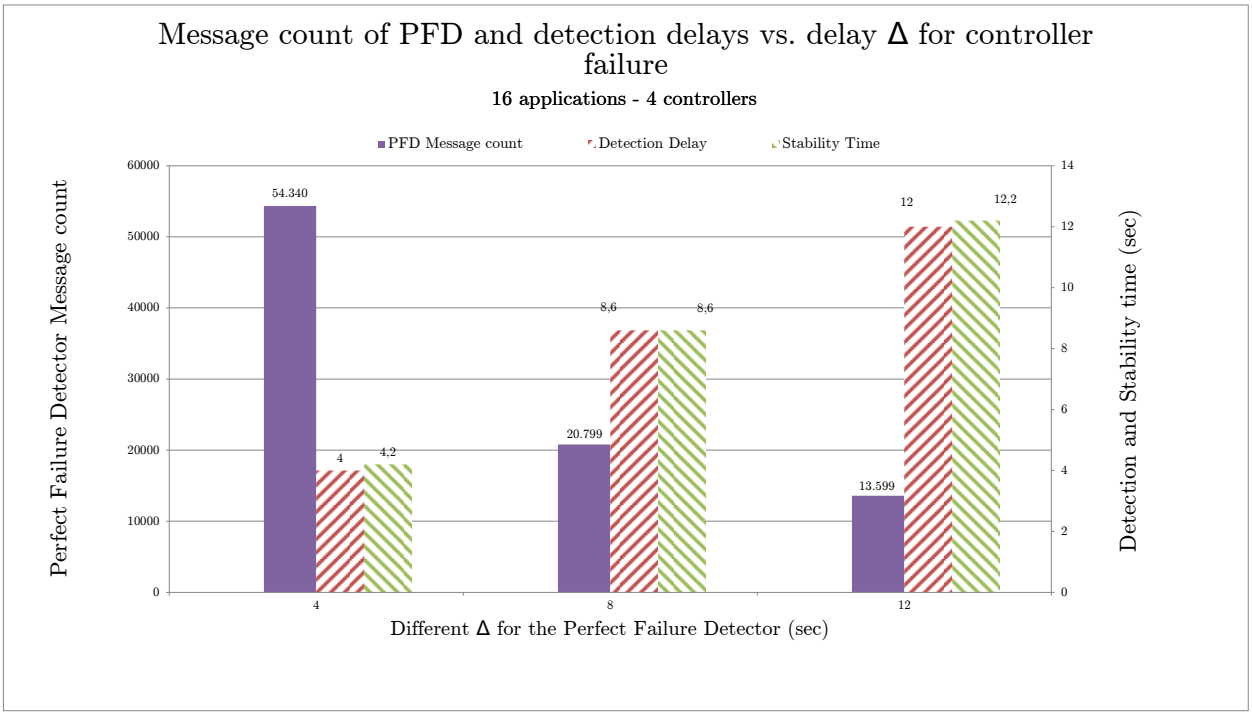


Figure 8.6: Failure detection and stability delay of $\mathcal{P}$ for different Δ.

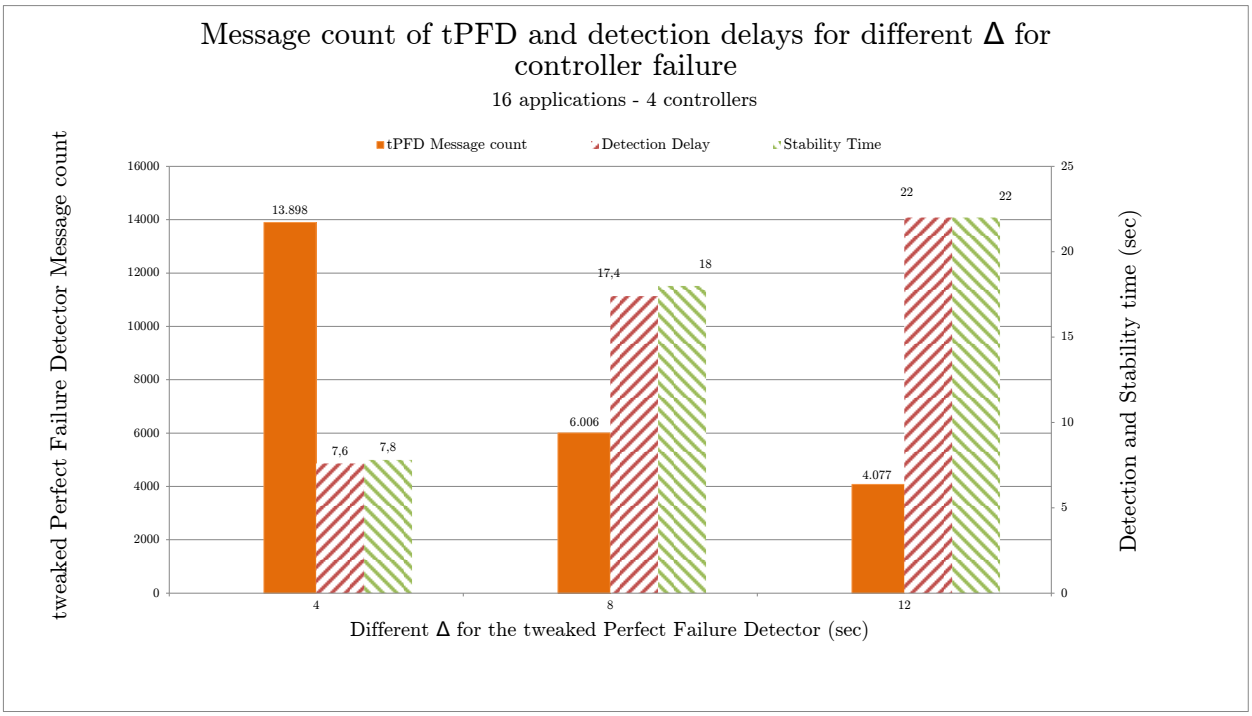


Figure 8.7: Failure detection and stability delay of $\mathcal{P}_t$ for different Δ.

8.4.2 Different failure scenarios

As explained in 8.2 the message count for Paxos depends on the size of cluster in case of a controller failure, whereas in case of a manager failure it is propotional to the number of coworkers. Similarly with the failure detectors, we sacrifice detection and stability delay to achieve less message transaction between cores in the case of manager failure. In figure 8.8 we see the average of messages exchanged in order to recover after a controller failure as well as the probability that a core inside or -outside the failed core's cluster detects the failure. As we expect the messages exchanged when detected from inside the cluster are much more less than when detected from outside the cluster, because in this situation we come through some of the $(N-N_f)\times(N_f-1)$ messages needed for the $\langle SIG\_CONTR\_TO\rangle$ signal. In addition, it is pretty obvious that shortening the cluster size results in higher probability of outside the cluster detection.

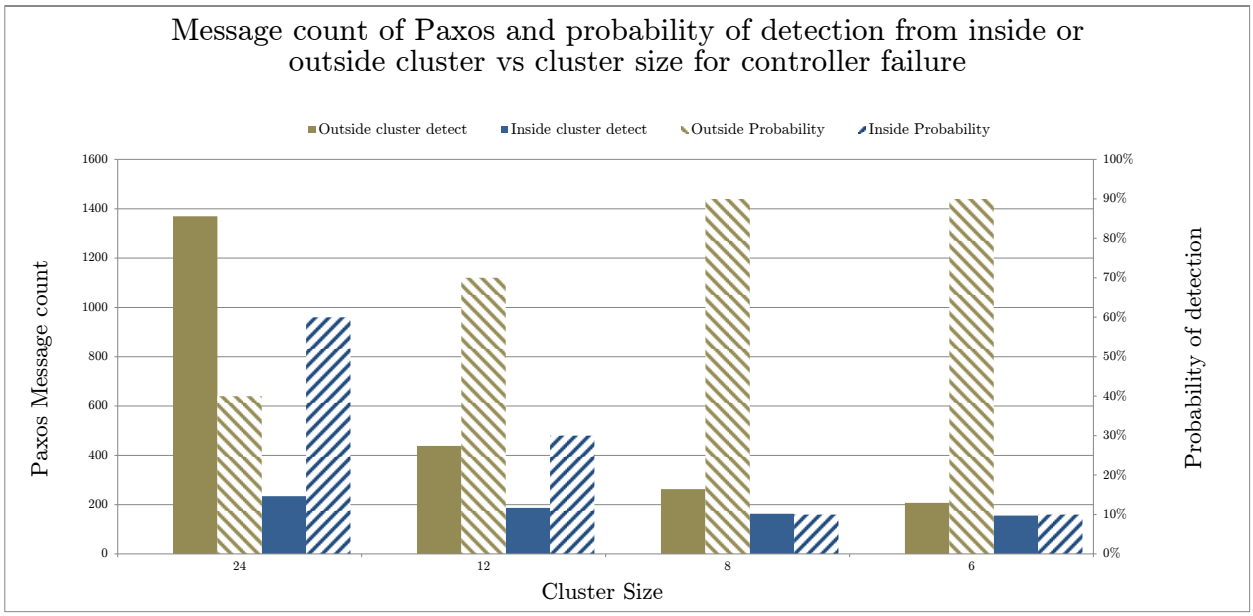


Figure 8.8: Paxos message count vs. cluster in case of controller failure

However, this is less of a concern because in smaller cluster sizes N_f gets smaller resulting in less messages. In figures 8.9 and 8.10 we see how the message count scales for different cluster sizes in both manager and controller failure as well as the delay until detection and stability.

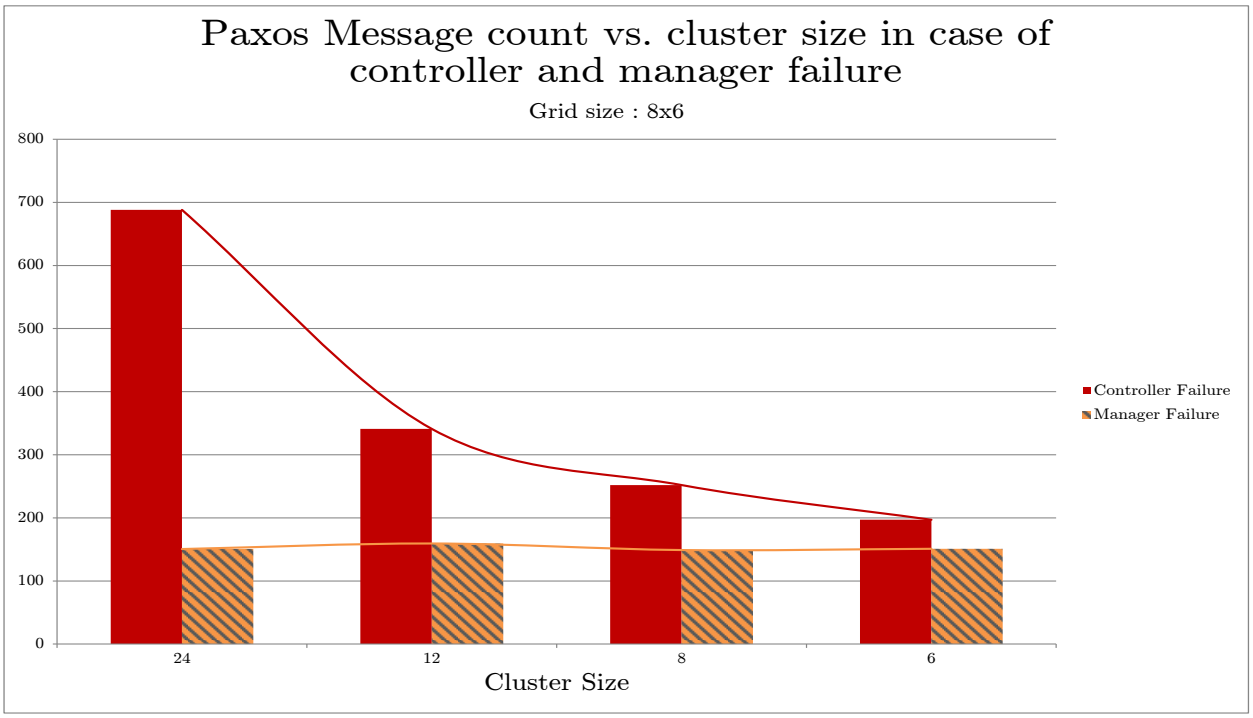


Figure 8.9: Message count vs. cluster size for controller and manager failure

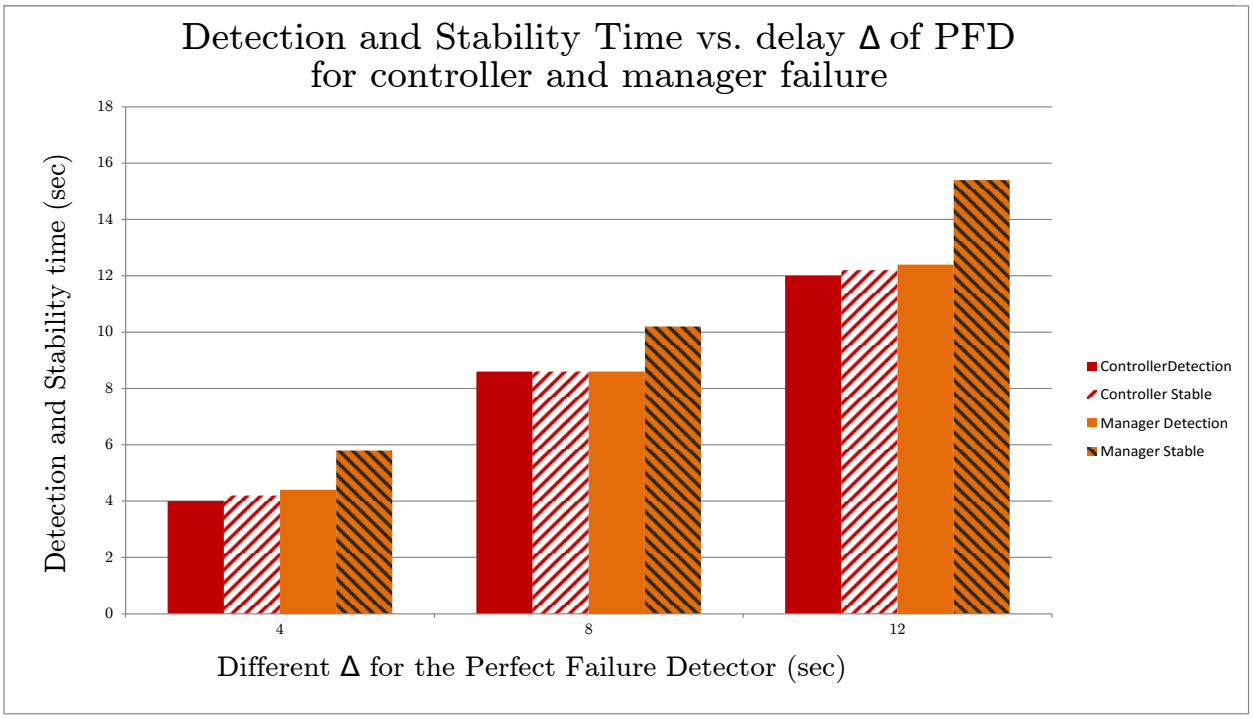


Figure 8.10: Detection and Stability time vs. cluster size for controller and manager failure

8.4.3 Larger grid size

In DRTRM the maximum number of coworkers a core can have is 8. Therefore, when simulating larger grid sizes where $N >> (W_f)^2$ the message count in case of a manager failure depends on the number of cores in the platform than on the number of coworkers. In figure 8.11 we present the message count for a larger grid of 16×12. As we can see the message count of Paxos in case of a manager failure increases compared to the message count of 6×8 grid. This is due to the reason we explained above.

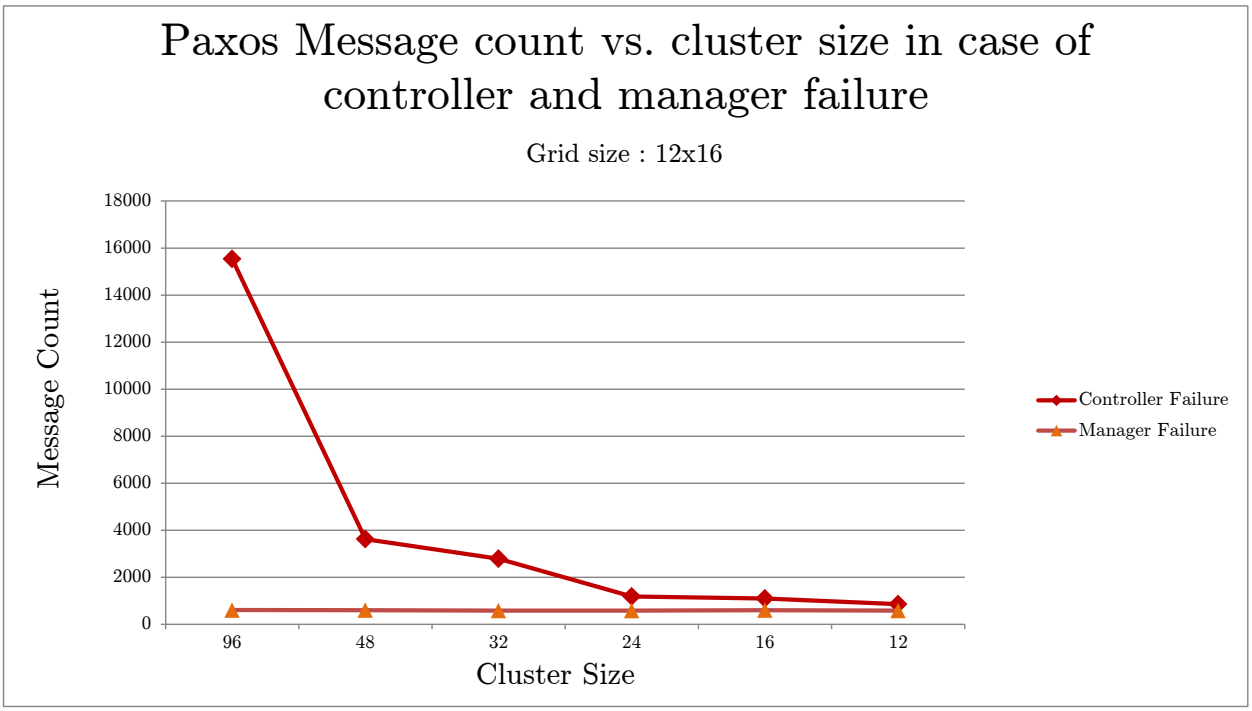


Figure 8.11: Message count vs. cluster size in 16×12 grid

Chapter 9

Conclusion and future work

9.1 Summary

In the current thesis, we were involved with the problem of deadlock detection, failure detection and reaching consensus on Network-on-Chip Multi-Processor System-on-Chip. Firstly, we pointed out the similarities between Networks on Chip and Distributed Systems by analyzing the main features of both of them. Subsequently, we presented some known algorithms and protocols about handling failures and reaching consensus which are used in distributed systems, as well as a resource management framework for NoC platforms which is known as DRTRM. As the final stage of the thesis, we implemented the abovementioned algorithms and combined them with the DRTRM framework.

As DRTRM targets both homogeneous and heterogeneous platforms, so does our implementation, because it basically works as an extension to the framework. The modified framework was tested on a simulator and on an actual NoC platform, that is the Intel SCC platform. As far as our experimental results are concerned, we examined different failure scenarios for different type of cores and counted the overhead of the algorithms in order to return to stable state.

9.2 Future Work

The following sections present ways to further modify the DRTRM framework in order to support additional features in the future.

9.2.1 Additional Failure Scenarios

In the current thesis we only examined cases where only one failure occurred. In addition, the type of the failures were only crash-stop ones (as described in section 3.2.1). However, in real life platforms more than one core may fail to communicate not only

by crashing but also by experiencing high latency in communication network. Taking these into consideration, in future research we could examine additional scenarios where multiple nodes fail simultaneously or nodes experience crashes with recoveries. Managing continuous failures can be achieved through Multi Paxos as described below.

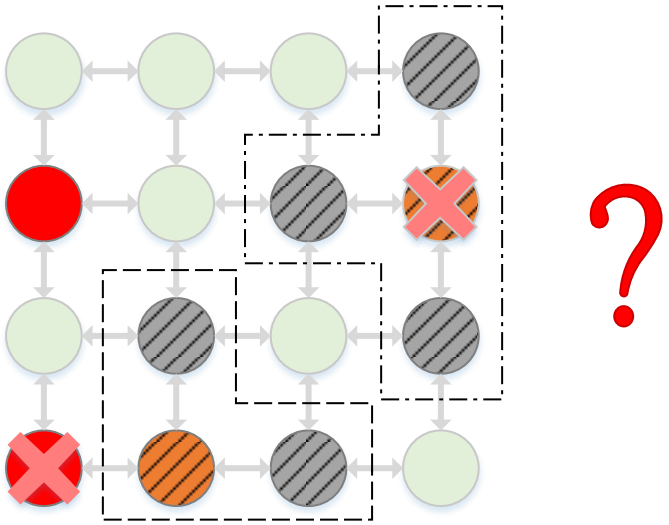


Figure 9.1: Future Work: What happens when multiple failures occur

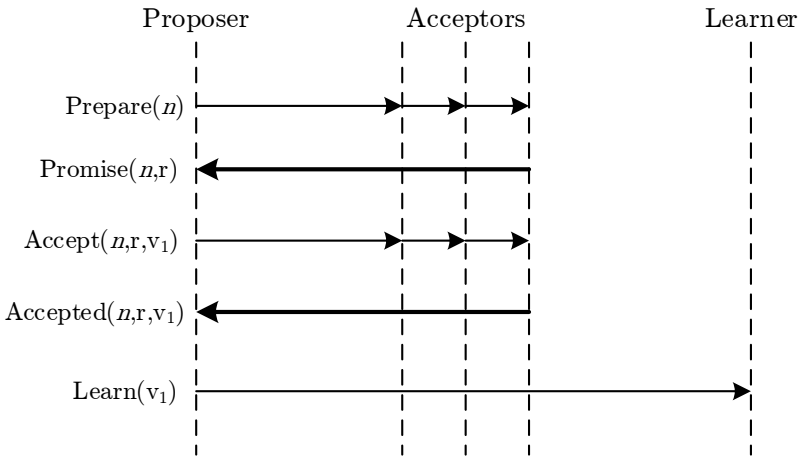
9.2.2 Paxos Optimizations

A number of optimizations can reduce message complexity and size as well as allow multiple instances of Paxos. These optimizations are summarized below:

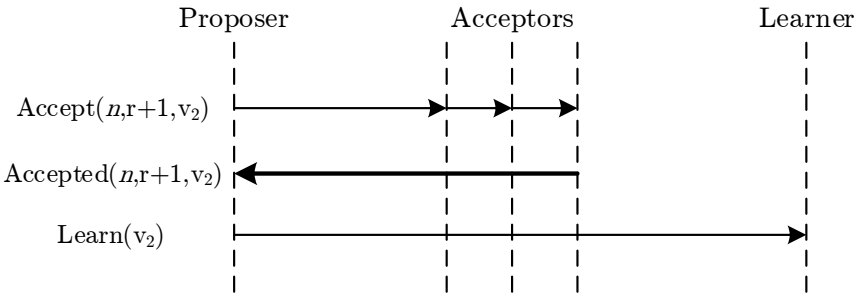
9.2.2.1 Multi-Paxos

We mentioned in the introduction that one of the main useful applications of the Paxos application is having the group of participants decide on a sequence of numbers. Since one round of Paxos results in a decision of one value, the naive way to go about finding a sequence of numbers would be to run Paxos many times.

One optimization that can be made in this case, assuming a single stable leader, is to skip the prepare phase. If we assume that the leadership will unchanged, there is no need to continue sending out proposal numbers - the first proposal number sent out will never be overridden since there is only one leader.



(a) In the first round Multi-Paxos acts as basic Paxos.



(b) In consequent rounds the new leader only has to send the accept messages.

Figure 9.2: Example of Multi-Paxos.

Thus, we only need to do the *prepare* phase once. In subsequent rounds of Paxos, we can just send the *accept* messages, with n as the proposal number used in the original *prepare* request and an additional parameter that indicates the sequence number, meaning the current round we are in. We do not have to worry about the worst case where leadership is not stable, because the algorithm will degrade gracefully into the general Paxos algorithm (both prepare and accept phases for each round).

9.2.2.2 Cheap Paxos

Cheap Paxos[24] extends Basic Paxos to tolerate F failures with $F+1$ main processors and F auxiliary processors by dynamically reconfiguring after each failure.

This reduction in processor requirements comes at the expense of liveness; if too many main processors fail in a short time, the system must halt until the auxiliary processors can reconfigure the system. During stable periods, the auxiliary processors take no part in the protocol.

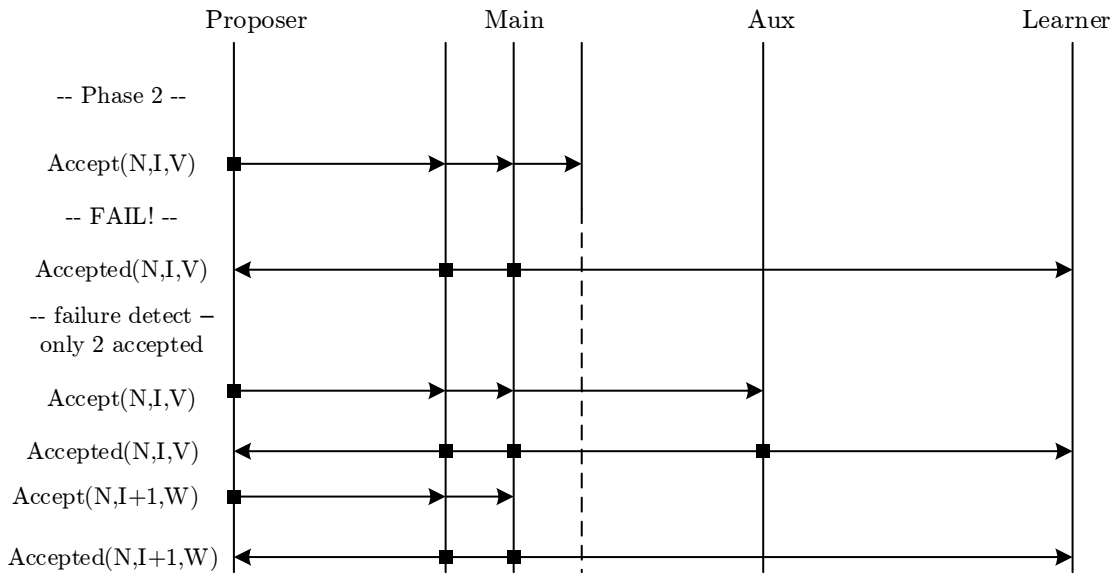


Figure 9.3: Example of Cheap Paxos

9.2.2.3 Fast Paxos

Fast Paxos [25] generalizes Basic Paxos to reduce end-to-end message delays. In Basic Paxos, the message delay from client request to learning is 3 message delays. Fast Paxos allows 2 message delays, but requires the Client to send its request to multiple destinations.

Intuitively, if the leader has no value to propose, then a client could send an Accept message to the acceptors directly. The acceptors would respond as in Basic Paxos, sending accepted messages to the leader and every Learner achieving two message delays from Client to Learner.

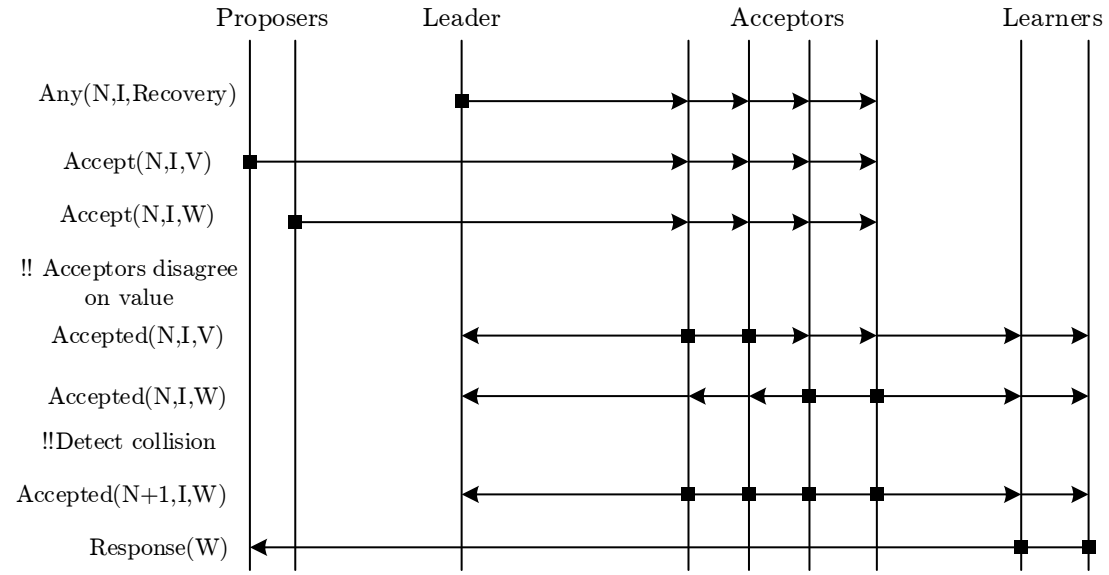


Figure 9.4: Example of Fast Paxos

If the leader detects a collision, it resolves the collision by sending accept messages for a new round which are accepted as usual. This coordinated recovery technique requires four message delays from Client to Learner.

The final optimization occurs when the leader specifies a recovery technique in advance, allowing the Acceptors to perform the collision recovery themselves. Thus, uncoordinated collision recovery can occur in three message delays (and only two message delays if all Learners are also Acceptors).

9.2.3 More Paxos

Except for managing failures Paxos can also be used to agree on values proposed by the SoC cores. For example, imagine a system with multiple sensors which detects the temperature in different locations of a room. Also, consider that the sensors are connected to a SoC. Different sensors will detect different values of the temperature. However, if we want to calculate some results based on the temperature of the room, the cores have to agree on a single temperature before proceeding.

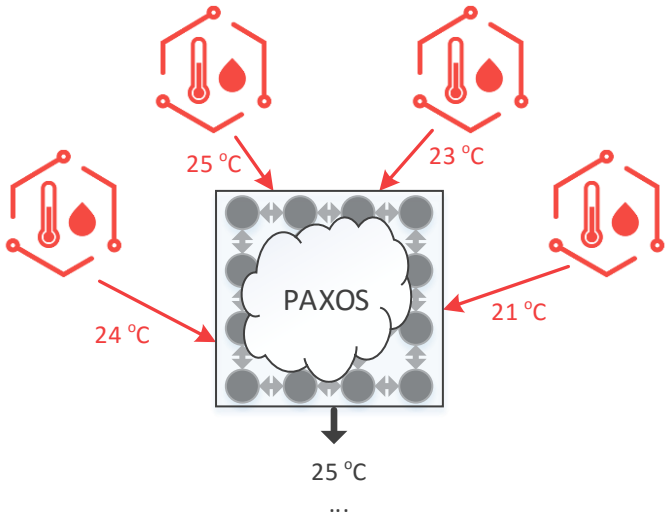


Figure 9.5: Example of multiple sensors proposing different values on a SoC

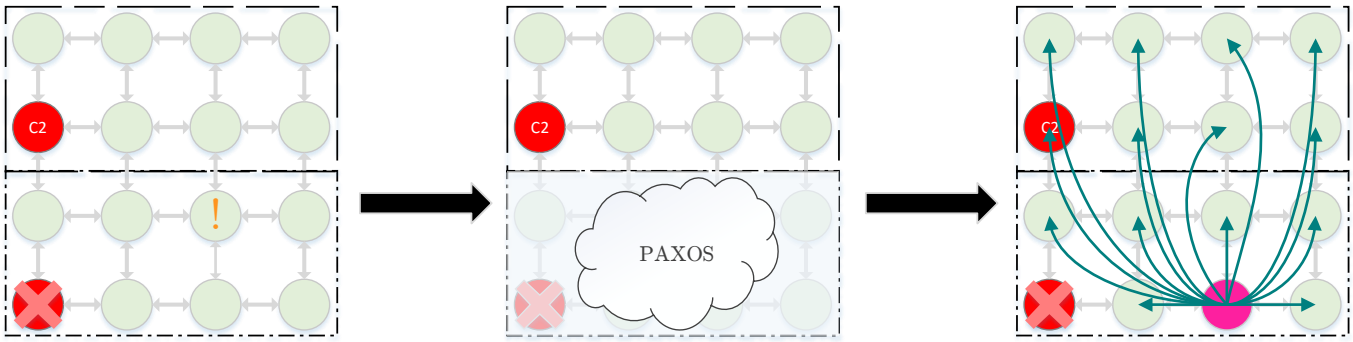


Figure 9.6: Example of multiple sensors proposing different values on a SoC

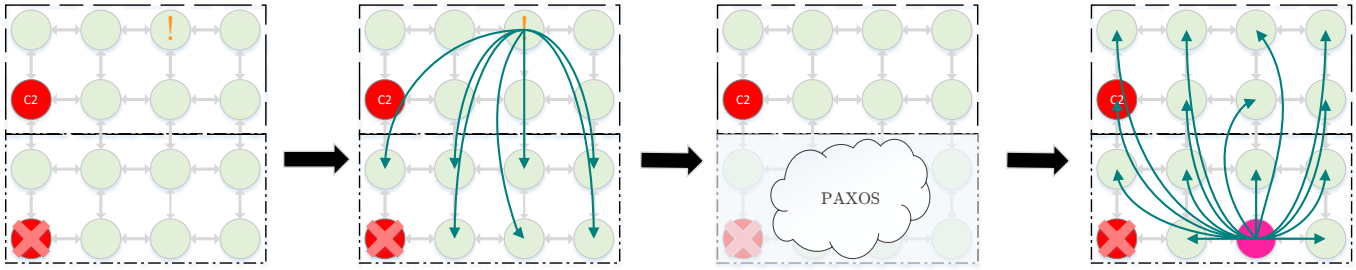


Figure 9.7: Example of multiple sensors proposing different values on a SoC

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems Concepts and Design*. Pearson Education, fifth ed., 2005.

[2] M. Salem, "Facebook distributed system case study for distributed system inside facebook datacenters," *International Journal of Technology Enhancements and Emerging Engineering Research*, pp. 152–160, 2014.

[3] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Survey Vol. 38*, p. 1–51, 2006.

[4] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: The programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[5] T. Mattson and R. van der Wijngaart, "Rcce: a small library for many-core communication," *Intel Corporation, May*, 2010.

[6] L. Lamport, "Paxos made simple," *ACM SIGACT News 32*, pp. 18–25, 2001.

[7] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335–350, 2006.

[8] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live - an engineering perspective," *Proceeding PODC '07 Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pp. 398–407, 2007.

[9] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review Volume 44*, pp. 35–40, 2010.

[10] P. Bogdan, T. Dumitraş, and R. Marculescu, "Stochastic communication: A new paradigm for fault-tolerant networks-on-chip," *VLSI design*, vol. 2007, 2007.

[11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming.* Springer Berlin, second ed., 2010.

[12] E. Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys (CSUR) Volume 19*, pp. 303–328, 1987.

[13] R. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys (CSUR) Volume 4*, pp. 179–196, 1972.

[14] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM) Volume 43*, pp. 225–267, 1996.

[15] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pp. 147–158, 1992.

[16] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, "Distributed run-time resource management for malleable applications on many-core platforms," pp. 1–6, Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE, 1905.

[17] V. Tsoutsouras, "Design and implementation of a run-time resource manager for malleable applications on network-on-chip (noc) architecture," diploma thesis, National Technical University of Athens, 2013.

[18] V. Tsoutsouras, S. Xydis, and D. Soudris, "Job-arrival aware distributed run-time resource management on intel scc manycore platform," in *Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference on*, pp. 17–24, IEEE, 2015.

[19] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems 16*, pp. 133–169, 1998.

[20] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems 4*, pp. 382–401, 1982.

[21] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona, "The real byzantine generals," *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd (Volume:2)*, pp. 6.D.4 – 61–11 Vol.2, 2004.

[22] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, *Computer Safety, Reliability, and Security: 22nd International Conference, SAFECOMP 2003, Edinburgh, UK, September 23-26, 2003. Proceedings*, ch. Byzantine Fault Tolerance, from Theory to Reality, pp. 235–248. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.

[23] M. Chandy, J. Misra, and L. Haas, "Distributed deadlock detection," *ACM Transactions on Computer Systems (TOCS) Volume 1*, pp. 144–156, 1983.

[24] L. Lamport and M. Massa, "Cheap paxos," in *Dependable Systems and Networks, 2004 International Conference on*, pp. 307–314, IEEE, 2004.

[25] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.