INDIGO: AN INFRASTRUCTURE FOR OPTIMIZATION OF DISTRIBUTED ALGORITHMS

by

VALERIY KOLESNIKOV

B.S., Sumy State University, Ukraine, 1995
M.S., Slippery Rock University, 1998

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Abstract

Many frameworks have been proposed which provide distributed algorithms encapsulated as middleware services to simplify application design. The developers of such algorithms are faced with two opposing forces. One is to design generic algorithms that are reusable in a large number of applications. Efficiency considerations, on the other hand, force the algorithms to be customized to specific operational contexts. This problem is often attacked by simply re-implementing all or large portions of an algorithm.

We propose InDiGO, an infrastructure which allows design of generic but customizable algorithms and provides tools to customize such algorithms for specific applications. InDiGO provides the following capabilities: (a) Tools to generate intermediate representations of an application which can be leveraged for analysis, (b) Mechanisms to allow developers to design customizable algorithms by exposing design knowledge in terms of configurable options, and (c) An optimization engine to analyze an application to derive the information necessary to optimize the algorithms. Specifically, we optimize algorithms by removing communication which is redundant in the context of a specific application. We perform three types of optimizations: static optimization, dynamic optimization and physical topology-based optimization. We present experimental results to demonstrate the advantages of our infrastructure.

INDIGO: AN INFRASTRUCTURE FOR OPTIMIZATION OF DISTRIBUTED ALGORITHMS

by

VALERIY KOLESNIKOV

B.S., Sumy State University, Ukraine, 1995
M.S., Slippery Rock University, 1998

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Approved by:

Major Professor
Gurdip Singh

Copyright

Abstract

Many frameworks have been proposed which provide distributed algorithms encapsulated as middleware services to simplify application design. The developers of such algorithms are faced with two opposing forces. One is to design generic algorithms that are reusable in a large number of applications. Efficiency considerations, on the other hand, force the algorithms to be customized to specific operational contexts. This problem is often attacked by simply re-implementing all or large portions of an algorithm.

We propose InDiGO, an infrastructure which allows design of generic but customizable algorithms and provides tools to customize such algorithms for specific applications. InDiGO provides the following capabilities: (a) Tools to generate intermediate representations of an application which can be leveraged for analysis, (b) Mechanisms to allow developers to design customizable algorithms by exposing design knowledge in terms of configurable options, and (c) An optimization engine to analyze an application to derive the information necessary to optimize the algorithms. Specifically, we optimize algorithms by removing communication which is redundant in the context of a specific application. We perform three types of optimizations: static optimization, dynamic optimization and physical topology-based optimization. We present experimental results to demonstrate the advantages of our infrastructure.

Table of Contents

List of Figures

Acknowledgments

I am very grateful to my major professor Dr. Gurdip Singh for supervising this work. He introduced me to the field, guided my work through the years, gave me many ideas, was very patient, and provided financial support for my studies. It is difficult to imagine to be able to do this work without this kind of support.

I am also thankful to Dr. Masaaki Mizuno, Dr. Mitchell Neilsen, Dr. Bala Natarajan, and Dr. Xiang Fang for serving as my committee members. Their insightful comments and encouragement especially during and after my proposal defense gave me a lot of help.

I am thankful to many professors who taught me while I studied at K-State. Many thanks go to Dr. Dave Gustafson who was the supervisor of my teaching activities during my first years.

I am thankful to my extended family and friends who supported and encouraged my family during my studies.

I thank my beloved wife, Inna, and children, Timothy and Anastasia, for always being there for me, having faith in me, their support, enouragement and love.

But at the end, all the glory goes to my Creator. You gave me strength and perseverance. You comforted me and always loved me.

Dedication

To my big family.

Chapter 1

Introduction

1.1 The problem domain

With increased deployment of communication infrastructure, a number of areas such as embedded systems, sensor networks and peer-to-peer computing are emerging in which distributed programming is the natural way to program systems. Although distributed programming is highly desirable, putting together a correct and efficient distributed system has been recognized as a difficult task due to several factors such as presence of heterogeneous hardware and software, lack of adherence to standards, asynchrony, limited local knowledge, uncertainty in message delays and computation time, and failures. These factors make design of distributed systems a much more challenging task as compared to their centralized counterparts. The explosive growth of distributed systems makes it imperative to understand how to overcome these difficulties.

Several frameworks have been proposed for component-based development to ease the design and development of distributed systems with the goal of isolating the developers from lower-level details.[1–10] As shown in Figure 1.1, in many such frameworks, an application developer is responsible for designing components and assembling systems from components using the underlying services. The task of a middleware developer is to provide a library of distributed algorithms and communication abstractions for common tasks such as mutual exclusion and termination detection to simplify programming and the deployment of the

applications.

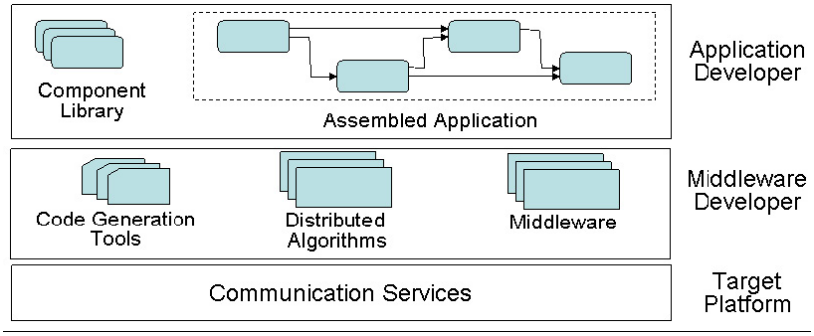

Figure 1.1: *A distributed computing framework*

Distributed systems middleware developers are often faced with two conflicting forces. One is to make the algorithms reusable in a number of different contexts, which forces them to be more generic. Efficiency considerations, on the other hand, force the algorithms to be customized to specific operational contexts (specific applications and target platforms). To make algorithms generic, designers often make weak assumptions about the application. For example, one can provide a generic algorithm for mutual exclusion which allows application entities to request a shared resource in any possible order. Each application, however, may have a specific pattern of resource usage (e.g., specific groups of components may alternate access to a shared resource), or may wish to give priority to specific types of processes. In such cases, the generic algorithm may be inefficient (may perform redundant or unnecessary communication), may require additional application level programming, and one may want to customize the algorithm so that it only performs the required control. As another example, algorithms for total ordering of events typically perform ordering without making any assumptions regarding the order in which the application may issue events.[1,11] However, if the application itself is issuing events in a certain order (e.g., an event e_a is only issued in response to e_b), then the event ordering algorithm may be performing duplicated work. To improve performance in such cases, we would like to optimize the algorithm to perform only the required ordering. This, for instance, is similar to a compiler optimizing a library function based on its usage (e.g., replacing a parameter with constant c if all in-

vocations to the function use c as the actual parameter). Along the same lines, to target a larger class of platforms, algorithms often make weak assumptions regarding the target platforms (e.g., they may assume that all channels take bounded but unpredictable time), even though stronger properties might be true in specific cases. Such stronger properties of the underlying platform may constrain the application events to interleave in a restricted manner, which may make some computation or message passing in the algorithm unnecessary. This, for instance, is similar to a compiler optimizing a program during the code generation phase by exploiting properties of the target architecture (e.g., re-arranging the instructions to minimize delays). This conflict is especially problematic in product line architectures wherein a fixed middleware infrastructure is made available to develop a family of similar applications (e.g., the class of tele-conferencing applications). In such systems, irrespective of their structure or size, all applications are forced to use the same underlying distributed algorithms to satisfy their requirements (even though the various applications may differ in their structure).

The problems described above are often attacked by simply re-implementing all or large portions of the existing algorithms. This is time-consuming and tedious, and it limits the ability to quickly develop new systems for emerging technologies. It may result in multiple variants of an algorithm, each having a rigid, inflexible interface offering limited variability. Furthermore, to simplify and speed up implementations, algorithms such as those based on centralized control may be employed. For example, there has been done a study on the Boeing Bold Stroke product line a platform for developing avionics applications.[12] Bold Stroke developers have identified several special application contexts where specialized configurations of the event service middleware are preferred for performance optimizations. While identifying such cases may be possible for application scenarios involving small number (10-20) of components, it is tedious and error-prone for larger scenarios. Therefore, tools that can identify and perform customizations in an automated manner are needed. There has been a considerable amount of work done in optimizing compilers for sequential and paral-

lel programs where library modules are optimized to specific usage semantics. Techniques such as aspect-oriented programming and feature-oriented programming have been used recently to optimize and adapt systems.[13,14] In the context of distributed systems, algorithms utilizing application semantics have been proposed for specific problems such as transaction processing, multicasting, and check-pointing.[15–18] However, tools and methodologies to customize distributed algorithms in a systematic manner are still lacking. Even if such methodologies were available, these may be ineffective due to lack of algorithms amenable to customization.

1.2 Overview of the approach

To address this, we propose *InDiGO*, an *In*frastructure for *Di*stributed al*G*orithm *O*ptimization. In our framework, algorithm designers develop generic, but customizable algorithms, and the infrastructure provides the tools necessary to customize such algorithms for specific applications. The architecture diagram for our approach is shown in Figure 1.2.

The important aspects of this framework are the following:

• *Tools to extract application information*: We specify applications in Cadena, an integrated modeling and development environment for component-based systems. As shown in Figure 1.2, a designer develops an application by identifying the component instances and specifying their port interconnections (assembly specification).

A component property specification (CPS) file is maintained for each component, which contains information relevant to the internal structure of the component (e.g., order in which it executes various actions). We have developed a tool that uses Cadena component and assembly specification files and the CPS files to construct an *application dependency graph* (ADG) which captures information pertaining to the application structure. We have also developed an analysis infrastructure to support a set of basic queries on the ADG to query for application-specific properties. These queries essentially ask for ordering information on events. The semantics of the data structures for customizing the algorithms as described below is defined in terms of these basic queries.

• *Development of customizable distributed algorithms*: To enable customization, an algorithm developer must expose *design knowledge* pertaining to an algorithm in a form which can be leveraged for analysis. In this work, we explore techniques to expose knowledge related to the *communication structure of an algorithm*. In an algorithm, a process may have to perform a number of interactions to accomplish various tasks such as accumulating states of other processes or obtain permissions. To accommodate arbitrary applications, designers often develop algorithms by including communication between all processes that could potentially participate in an interaction. In specific applications, however, some of

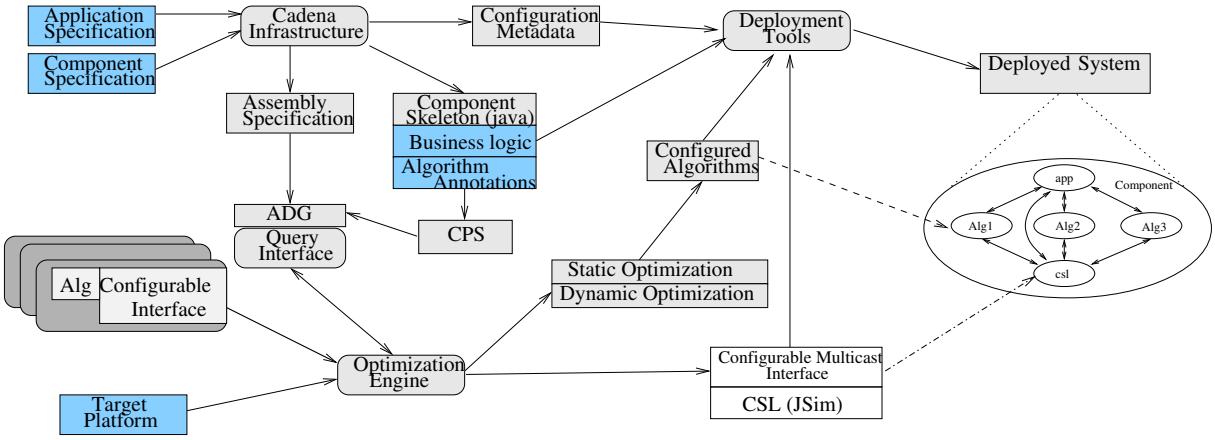

Figure 1.2: *Architecture Diagram of InDiGO*

this communication may be unnecessary. Therefore, we require a designer to follow the following steps:

(a): For each algorithm *alg*, the designer first identifies the interaction sets, denoted by *alg.interaction_set*, which characterize its communication structure and specify the processes participating in each interaction. The designer then writes *alg* in terms of these sets. For example, one interaction set for the termination detection algorithm is *send_marker_to* and the algorithm is written so that process i sends a marker message only to nodes belonging to *send_marker_to*, rather than all neighbors as in a generic algorithm. The interaction sets provide a way to expose the communication structure of an algorithm for possible optimization.

(b): In the next step, the designer defines the *membership criterion* for each set; that is, the criterion for a process to be involved in an interaction. This criterion is a problem-specific property a process must satisfy to participate in the interaction. The criteria must be defined in terms of the queries supported by the analysis infrastructure. For example, the designer may specify that component j belongs to *send_marker_to* set for i only if it is possible for j to be active when i is passive in the application.

(c): Next, the designer supplies information for *dynamic updates to the interaction sets*. In an algorithm, as a result of message passing, a process may obtain knowledge of the states

of the application entities at other processes. For example, when process i receives a request message from j, it knows that the application entity at j is in the requesting state. The designer exposes this information by identifying a set of assertions, $alg.app\_assert$, and two sets of control points, α_pos and α_neg, for each assertion $\alpha \in alg.app\_assert$. α_pos is the set of control points where α is known to be true and α_neg represents the control points where α may no longer be true. As shown later, this information can be used for dynamic updates to the interaction sets. Our main goal in using this additional design information is to demonstrate that by exposing more design knowledge, the designer can enable more optimization opportunities.

• Given an application *App*, an algorithm *alg*, and a physical topology, the optimization engine optimizes *alg* by removing communication from *alg* which is redundant in the context of *App* and the physical topology. These optimizations include the following:

⋆ Static optimization: This involves analyzing the ADG to compute the initial values of each set in $alg.interaction\_set$. For example, for the termination detection algorithm, if the analysis of the ADG reveals that j is always passive whenever i is passive, then j will be excluded from the interaction set $send\_marker\_to$ for component i. This is different from conflict sets which are based on semantics of message types (rather than application analysis).[3,17,19]

⋆ Dynamic optimization: For each assertion α in $alg.app\_assert$, the optimization engine generates a set of dynamic optimization rules which specify whether any set in $alg.interaction\_set$ can be further constrained when α is true. The dynamic optimization rules indicate the processes that can be removed from various sets when specific assertions hold. The algorithm *alg* is then transformed to keep track of when the assertions are true, and the dynamic optimization rules are applied to update the interaction sets.

⋆ Physical topology-based optimization: We are using the J-Sim simulator[20] for evaluation. We have extended the Core Service Layer (CSL) of J-Sim to perform multi-

destination routing. Using information about the network topology, the extended CSL layer can remove redundant messages when the same message is to be sent to a set of processes. Our infrastructure analyzes the *physical topology graph* (PTG), which describes the underlying network topology on which the application is to be deployed, to derive information necessary to initialize the CSL layer for optimized multi-destination routing. We use this information to initialize data structures in the CSL layer of J-Sim to enable efficient communication (for example, if a request in the mutual exclusion service is to be sent from i to both j and k, and j is in the communication path from i to k, then we combine the request message to be sent to j and k).

Metadata for all of the optimization are generated in the form of XML files. The deployment tools use this metadata to configure the algorithms and the CSL layer and to deploy the component code at each site. We have performed extensive experimental studies to demonstrate the advantages of InDiGO. Clearly, as an application becomes more constrained (that is, the application itself imposes more constraints on the order in which the components can perform actions), one would expect more optimization opportunities. We demonstrate this by conducting a series of experiments by incrementally adding constraints to an application and showing that InDiGO tools can extract and exploit this information to improve the performance of the underlying algorithms. The types of optimizations (both static and dynamic) identified in these applications are non-trivial and difficult to arrive at by manual inspection of the application (especially when the application is large) and will require automated tools of the type provided by InDiGO.

The main contribution of InDiGO is the development of an *extensible framework* to support the optimization process. The framework capabilities includes:

- Tools to extract application information from Cadena in a form amenable to analysis,
- Mechanisms for an algorithm designer to encode and expose design knowledge for potential optimizations,
- Tools to analyze an application to derive information necessary to customize the al-

gorithms.

In this work, we utilize Cadena tool to specify component-based distributed applications. Cadena provides component specification file and assembly specification file that describe application assembly. Cadena also generates JAVA skeleton files per each component type. Cadena CPS files describe internal behavior of components. Cadena also provides mechanisms to deploy a distributed system. The rest is my contribution to this work and includes:

- extension of the traditional methodology to develop component-based distributed systems in Cadena to include middleware distributed services through the use of annotations in the component code,
- development of a distributed application abstraction in the form of an application dependency graph that can be analyzed for possible optimizations,
- development of a tool to construct an application dependency graph from application specification information,
- design of a query interface to query application dependency graph for information of interest,
- development of basic queies,
- utilization of SPIN model checker to answer basic queries,
- development of a tool to convert application dependency graph into a Promela model used by SPIN,
- provision of mechanisms for an algorithm designer to encode and expose design knowledge for potential optimizations,
- development of customizable algorithms presented in this thesis,
- development of tools to analyze an application to derive information necessary to customize the algorithms, and
- implementation of applications presented in evaluation section.

1.3 Thesis organization

The rest of this thesis is organized as follows. Chapter 2, provides background information and describes distributed system model. Chapter 3 motivates the problem and discusses related work. Chapter 4 describes InDiGO framework in detail. The following chapter, Chapter 5, discusses types of optimization that we perform. Experimental results are presented in Chapter 6. We conclude and discuss future work in Chapter 7. Appendix describes a case study that utilizes the capabilities of our framework on a class of distributed applications. It also describes more distributed algorithms that we looked at during the years of working on InDiGO framework. We also provide supplemental information like grammars in the appendix.

Chapter 2

Model for distributed systems

In this chapter we present our distributed system model for message-passing systems with no failures. We consider an asynchronous timing model. Next we describe a component based approach to design and build distributed applications. In addition to describing formalism for the systems, we also define the main complexity measure - number of messages, and present the conventions we will use for describing algorithms in pseudocode.

2.1 System model

A *distributed system* is a collection of individual computing devices that can communicate with each other. We consider distributed systems where communication takes place through message passing. We assume the system to be without failures. We also consider asynchronous timing model. In asynchronous systems, there is no fixed upper bound on how long it takes a message to be delivered or how much time elapses between consecutive steps of a processor.

In a *message-passing system*, processors communicate by sending messages over communication links called *channels*. Each channel provides a bidirectional connection between two specific processors. The pattern of connections provided by the channels describes the *topology* of the system. The topology is represented by an undirected graph in which each node represents a processor. An edge is present between two nodes if and only if there is a channel between the corresponding processors. A node is a *neighbor* of another node if

and only if there is a direct link between the two. The collection of channels along with processors is also referred to as a *network*. An algorithm for a message-passing system with a specific topology consists of a local program for each processor in the system. Processor's local program provides the ability for the processor to perform local computation and to send messages to and receive messages from each of its neighbors in the given topology.

Formally, a distributed system consists of n processors $P_0, \ldots, P_{n-1}$, where i is the index of processor P_i and k channels $CH_1, \ldots, CH_k$, where j is the index of channel CH_j. Each processor P_i is modeled as a state machine SM_i.

Definition 2.1. We define a *state machine* SM_i for processor P_i as the following tuple $< Q_i, A_i, T_i, s_i, F_i >$, where Q_i is a set of states of P_i, A_i is a set of actions, T_i is a transition function $Q_i \times A_i \rightarrow Q_i$ that takes as input a state $q_i \in Q_i$ and an action $a \in A_i$ and produces as output a state $q_i^{'} \in Q_i$. We denote such a transition as $q_i \xrightarrow{a} q_i^{'}$. $s_i \subset Q_i$ is a distinguished subset of *initial states*, and $F_i \subset Q_i$ is a distinguished subset of *terminal states*.

Each processor is identified with a particular node in the topology graph.

Definition 2.2. A *configuration* is a vector $CF = (q_0, \ldots, q_{n-1}, ch_1, \ldots, ch_k)$ where q_i is a state of P_i and ch_j is a state of channel CH_j.

Definition 2.3. An *initial configuration* is a vector $CF_0 =(q_0, \ldots, q_{n-1}, ch_1, \ldots, ch_k)$ where each q_i is an initial state of P_i and each ch_j is empty.

Definition 2.4. An action $a \in A_i$ of processor P_i is either receipt of a message, send of a message, or a computation action.

Definition 2.5. Occurrences of an action that can take place in a system are modeled as *events*.

Notation 2.6. We will use notation a^x to denote the event of x^{th} occurrence of action a.

We consider three kinds of events.

Definition 2.7. A *computation event*, $c^x(i)$, represents a computation step of processor P_i in which P_i's transition function is applied to its current state. If the state machine for processor P_i has a transition $q_i \xrightarrow{c} q_i'$, which says that if processor P_i is currently in state q_i then it can go to state q_i' when computation event c^x happens, then the system can go from the configuration (q_0, ..., q_i, ..., q_{n-1}, ch_1, ..., ch_k) to the configuration (q_0, ..., q_i', ..., q_{n-1}, ch_1, ..., ch_k) when computation event c^x happens on processor P_i.

Definition 2.8. A *send event*, a^x, where $a = send(i, j, m)$, represents the sending of message m from processor P_i to processor P_j. If the state machine for processor P_i has a transition $q_i \xrightarrow{send(i,j,m)} q_i'$, which says that if processor P_i is currently in state q_i then it can go to state q_i' when send event a^x happens, where $a = send(i, j, m)$, then the system can go from configuration (q_0, ..., q_i, ..., q_{n-1}, ch_1, ..., ch_l, ...,ch_k) to the configuration (q_0, ..., q_i', ..., q_{n-1}, ch_1, ..., $ch_l \bullet m$, ...,ch_k), where ch_l is the state of channel CH_l from processor P_i to processor P_j, when send event a^x happens on processor P_i, where $a = send(i, j, m)$. Notation $ch_l \bullet m$ means that message m was added to channel CH_l.

Definition 2.9. A *receive event*, a^x, where $a = receive(i, j, m)$, represents the receiving of message m by processor P_i from processor P_j. If the state machine for processor P_i has a transition $q_i \xrightarrow{receive(i,j,m)} q_i'$, which says that if processor P_i is currently in state q_i then it can go to state q_i' when receive event a^x happens, where $a = receive(i, j, m)$, then the system can go from configuration (q_0, ..., q_i, ..., q_{n-1}, ch_1, ..., ch_l, ...,ch_k) to the configuration (q_0, ..., q_i', ..., q_{n-1}, ch_1, ..., ch_l', ...,ch_k), where ch_l is the state of channel CH_l from processor P_j to processor P_i with message m as the first message in it and ch_l' is the state of channel CH_l from processor P_j to processor P_i with message m removed, when receive event a^x happens on processor P_i, where $a = receive(i, j, m)$.

The behavior of a system over time is modeled as an execution.

Definition 2.10. An *execution* is a sequence of configurations alternating with events $CF_0, e_1, CF_1, e_2, CF_2, \ldots$, where $CF_i \xrightarrow{e_{i+1}} CF_{i+1}$ and CF_0 is an initial configuration.

With each execution we associate a schedule.

Definition 2.11. A *schedule* is the sequence of events in the execution.

Definition 2.12. An execution is *admissible* if each processor has an infinite number of computation events and every message sent is eventually delivered.

Remark 2.13. The requirement for infinite number of computation events models the fact that processors do not fail. It does not imply that the processor's local program must contain an infinite loop; the informal notion of termination of an algorithm can be accommodated by having the transition function not change the processor's state after a certain point, once the processor has completed its task. In other words, the processor takes "dummy steps" after that point.

Definition 2.14. A schedule is admissible if it is the schedule of an admissible execution.

2.2 Component-based distributed applications

In this section, we describe a component based approach to develop distributed applications. Cadena is an integrated modeling environment for modeling and building component based systems[21]. Here we also discuss the aspects of the development methodology in Cadena which are relevant to our approach. We are using the Corba Component Model (CCM) as the specification style. CCM framework aids application developers by providing services for common aspects such as distributed deployment, event notification, transactions, persistence, and security. Cadena, in particular, provides facilities for defining component types using CCM IDL, assembling systems from components and producing stubs and skeletons implemented in Java. Cadena is also providing basic event services to implement interactions between the components.

In the component architecture that we employ, a basic entity is a *component*. Distributed applications are assembled from components by specifying component instances and connections between them. Components reside on processors. Each processor might host any number of components. Each component owns one or more end points, called *ports*. The component where a port resides is called the *host* component of the port. Two components are connected by "wiring" their ports together. When a component sends data at one of its ports, the port relays the data to the port(s) that connect to it. When data arrives at a port, the component which owns the port invokes a handler for that port to process the data. The pattern of connections provided by the ports wiring along with components' internal connections describe the *application topology* of the system. The application topology is represented by a directed graph in which each port is represented by a node. An edge is present between two nodes representing ports if and only if there is a wiring between the corresponding ports. A node is a *neighbor* of another node if and only if there is a direct link between the two. Each component has a local program that provides the ability for the component to perform local computation and to send messages to and receive messages from each of its neighbors in the given application topology.

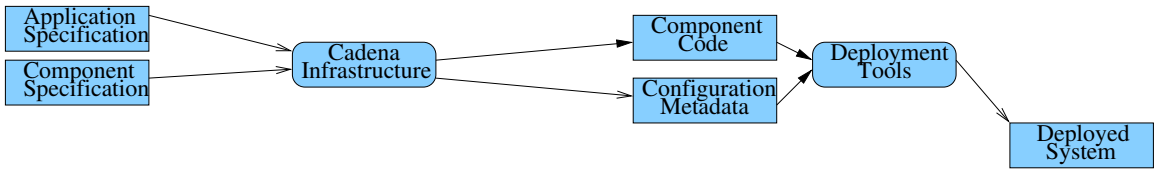

Figure 2.1: *Architecture of the traditional development process in Cadena*

Next, we formally define a component-based application and its structure. We will also use a simple example of Figure 2.3 to illustrate the various steps (shown in Figure 2.1) of the development process. In this example, discussed in more detail in the experimental results section, components are arranged in disjointed clusters, and are bidding for an item. Furthermore, components in each cluster bid for the item in a round-robin manner. Figure 2.3 shows the components for a single cluster.

Definition 2.15. We define a component C as the following tuple $< inp, outp, handlers >$, where inp is a set of input ports, $outp$ is a set of output ports and $handlers$ is a set of event handlers of component C.

Event handlers are associated with input ports and contain local actions and send message actions. Event handlers are triggered by receive message actions. So, computation events in the system model correspond to the local actions in the component-based application model. Send events in the system model correspond to sending a message actions in the component-based application model and receive events in the system model correspond to receiving a message actions in the component-based application model.

Notation 2.16. Let $C.inp$, $C.outp$ and $C.handlers$ denote the set of input ports, the set of output ports and the set of event handlers of component C respectively.

• The first step in traditional development of component based distributed systems is for the application developer to specify the components.

For example, a component `bidCompInit` shown in Figure 2.2 has three ports.

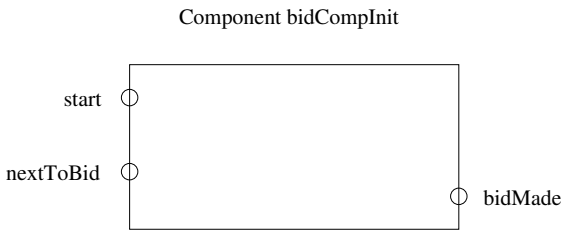

Figure 2.2: *Example of a component with 3 ports*

The component specification in Listing 2.1 defines `bidCompInit` as having two input ports and one output port. `bidCompInit` publishes events on port `bidMade` of type `bid`, and consumes events on port `nextToBid`. It also consumes events on port `start` of type `init`. Since we want to model asynchronous inter-process interactions via message passing, we will restrict ourselves to event ports (we do not consider ports for synchronous method calls such as those allowed in the Corba Component Model). In our framework, we allow a designer to tag a port as an `init` port; such a port is used to initialize and start the application. In Listing 2.1, the `init` port is tagged as type `init`.

```
1   :
2   eventtype init();
3   eventtype bid();
4
5   Enum Mode {continueBid, stopBid};
6   Component bidCompInit {
7    publishes bid bidMade;
8    consumes bid nextToBid;
9    consumes init start;
10   attribute Mode bidstate;
11 };
```

Listing 2.1: *Specification of component bidCompInit*

Listing 2.2 shows a Cadena component specification file. Component specification file describes component types and gives their ports information. For example, in Listing 2.2, lines 4-15 specify a component type bidCompInit. This type defines three ports specified in lines 6-8, 9-11, and 12-14.

```
1  ?xml version="1.0" encoding="ASCII"?>
2  <edu.ksu.cis.cadena.core.specification.module:Module xmi:version="2.0" xmlns:xmi=
3    <style href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_1MdskAMFEdqT6_lD
4    <components uuid="_OyaIQBbNEdypMp3lNrZ05w" name="bidCompInit" abstract="false">
5      <componentKind href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Nu9hoA
6      <ports uuid="_8_RL4BbOEdypMp3lNrZ05w" name="bidMade" interface="_sYVrUBbNEdyp
7        <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_AjRjcANsEdqTK
8      </ports>
9      <ports uuid="_JTsK4BbPEdypMp3lNrZ05w" name="nextToBid" interface="_sYVrUBbNEd
10       <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdqTK
11     </ports>
12     <ports uuid="_L196YBbPEdypMp3lNrZ05w" name="start" interface="_uhdfgcxsxagyaw
13       <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdqTK
14     </ports>
15   </components>
16 ...
17 </edu.ksu.cis.cadena.core.specification.module:Module>
```

Listing 2.2: *Example of a Cadena component specification file*

Definition 2.17. We define an application *App* as the following tuple $< components, connections >$, where *components* is a set of component instances in *App* and *connections* is a relation (x, y), where $x \in C.outp$, $y \in C'.inp$ and $C, C' \in components$ and describes interconnections between component instances in *App*.

Notation 2.18. Let *App.components* denote the set of component instances in *App*.

Notation 2.19. Let *App.connections* denote the set of connections in *App*.

• The next step in traditional development of component based distributed systems is to assemble a system by identifying the component instances and their interconnections. Cadena provides a graphical interface to specify the system assembly. Figure 2.3 shows the graphical representation of the scenario in Cadena. This scenario has one instance, C1, of component type `bidCompInit` and two instances, C2 and C3 of type `bidComp`. In this system, for example, output port `bidMade` of component instance C1 is connected to port `nexToBid` of component instance C2.

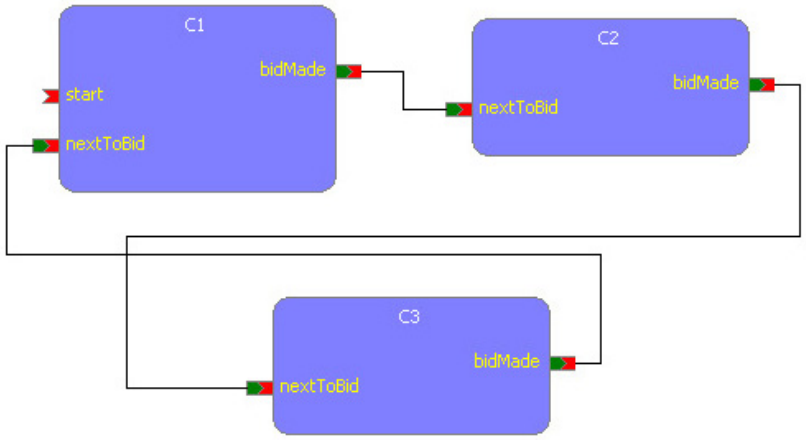

Figure 2.3: *Graphical representation of assembly specification in Cadena*

Listing 2.3 shows the assembly specification in Cadena.

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <edu.ksu.cis.cadena.core.specification.scenario:Scenario xmi:version="2.0" xmlns:
3    <connectors uuid="_qrVl8YjnEdyLreIOXUe8aA">
4      <portBindings uuid="_qrVl8ojnEdyLreIOXUe8aA">
5        <instanceRole uuid="_pln7EIjnEdyLreIOXUe8aA" instance="_eM-n0IjnEdyLreIOXUe
6          <port xsi:type="edu.ksu.cis.cadena.core.specification.module:ComponentPor
7        </instanceRole>
8        <portSpec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_U9ryIANsE
9      </portBindings>
10     <portBindings uuid="_qrVl84jnEdyLreIOXUe8aA">
11       <instanceRole uuid="_tOnbBYjnEdyLreIOXUe8aA" instance="_fwzLAIjnEdyLreIOXUe
12         <port xsi:type="edu.ksu.cis.cadena.core.specification.module:ComponentPor
13       </instanceRole>
14       <portSpec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_VJbvIANsE
15     </portBindings>
16     <kind href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_eUfwgANsEdqTKIG
17   </connectors>
18 ...
19   <componentInstances uuid="_eM-n0IjnEdyLreIOXUe8aA" name="C1">
20     <portProperties uuid="_eNIY0IjnEdyLreIOXUe8aA">
21       <key xsi:type="edu.ksu.cis.cadena.core.specification.module:ComponentPort"
22     </portProperties>
23     <portProperties uuid="_eNIY0YjnEdyLreIOXUe8aA">
24       <key xsi:type="edu.ksu.cis.cadena.core.specification.module:ComponentPort"
25     </portProperties>
26     <portProperties uuid="_eNIY0ojnEdyLreIOXUe8aA">
27       <key xsi:type="edu.ksu.cis.cadena.core.specification.module:ComponentPort"
28     </portProperties>
29     <type href="../module/intro.module#_L-_UwIjkEdyLreIOXUe8aA"/>
30   </componentInstances>
31   <style href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_1MdskAMFEdqT6_l
32   <imports href="../module/intro.module#_6P3noIjjEdyLreIOXUe8aA"/>
33 </edu.ksu.cis.cadena.core.specification.scenario:Scenario>
```

Listing 2.3: *Example of a Cadena assembly specification file*

For example, lines 19-30 specify a component instance C1. Lines 20-22, 23-25 and 26-28 specify three ports that component instance C1 has. Other component instances are not shown. Lines 3-17 specify a connection between two ports. More specifically, lines 4-9 specify an output port of one component and lines 10-15 specify an input port of another component. The two ports are connected.

• The next phase is the generation of code and the configuration metadata. Cadena uses the OpenCCM's IDL to Java compiler to generate the component and container code templates from the component IDL definitions. This produces an implementation (Java) file for each component into which the designer is supposed to fill the business logic. Java skeleton file corresponds to a state machine. An example of such a file is shown in Listing 2.4.

```
1  public class bidComp extends Component {
2
3     ...
4
5     public void process(Object data_, Port inPort_){
6
7        if (data_ instanceof Message){
8           if (data_ instanceof BidMadeMessage){
9
10          }
11       }
12    }
13 }
```

Listing 2.4: *Java skeleton file for bidCompInit*

The "wiring" or interaction between the components is realized in Cadena by providing basic event services, such as event notification. Public subscribe system is used where subscriber subscribes for events published by publisher. When publisher publishes an event, the subscriber is notified by the event notification service. Cadena tools also generate configuration code (in XML format) to deploy the system.

• The final step involves installing the code on each node in the network. The designer must provide a mapping, Map, to specify the node where each component instance is to be deployed. The deployment tools use this mapping to generate and install the code to be deployed on each node.

Definition 2.20. We define a mapping MAP as a function $C \rightarrow P$ that maps a component instance $c \in C$ to a processor $p \in P$ on which c is to be deployed.

The example in Figure 2.4 shows a possible mapping for components shown in Figure 2.3: the mapping function maps component C1 to processor P1, component C2 to processor P2, and component C3 to processor P3.

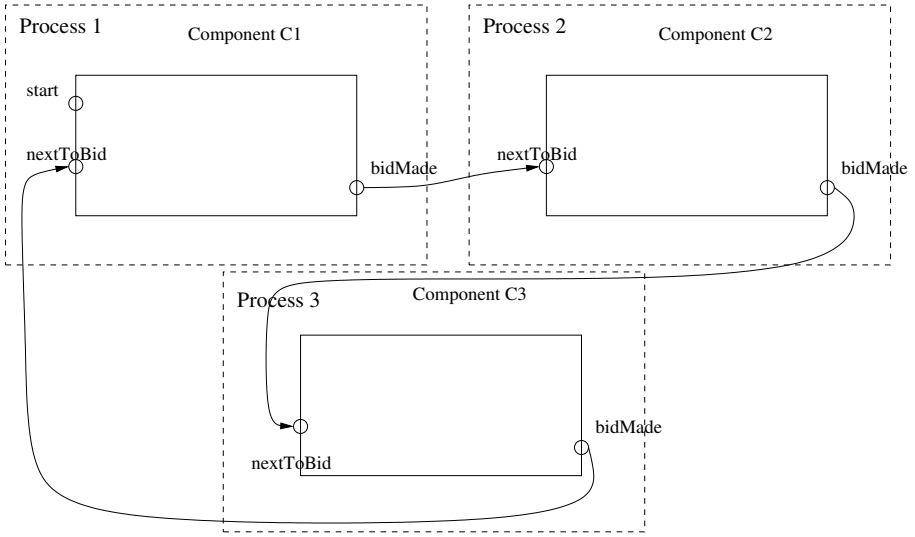

Figure 2.4: *Example of a system with 3 components*

2.3 Complexity measure

We will be interested in the number of messages (message complexity) as the complexity measure for analyzing our algorithms and optimizations.

To define this measure, we need a notion of the termination of algorithm. We assume that each processor's state set includes a subset of *terminated* states and each processor's transition function maps terminated states only to terminated states. We say that the system (algorithm) has terminated when all the processors are in terminated states and no messages are in transit. Note that an admissible execution must still be infinite, but once a processor has entered a terminated state, it stays in that state, taking "dummy" steps.

Definition 2.21. The *message complexity* of an algorithm is the maximum, over all admissible executions of the algorithm, of the total number of messages sent.

2.4 Pseudocode conventions

In the model just presented, an algorithm would be described in terms of state transitions. However, we will seldom do this, because state transitions tend to be more difficult for people to understand; in particular, flow of control must be coded in a rather contrived way in many cases.

Instead, we will present algorithms in pseudocode. Algorithms will be described in an interrupt-driven fashion for each processor. The effect of each message will be described individually. This is equivalent to the processor handling the pending messages one by one in some arbitrary order. It is also possible for the processor to take some action even if no message is received. The local computation done within a computation event will be described in a style consistent with typical pseudocode for sequential algorithms.

In the pseudocode, the local state variable of processor P_i will not be subscripted with i; in discussion and proof, subscripts will be added when necessary to avoid ambiguity. Comments will begin with //.

Chapter 3

Problem motivation and related work

In this chapter we point out the shortcomings of traditional approach in development of distributed applications and motivate the problem that we address in this thesis. We finish this chapter with the description of related work.

3.1 Problem motivation

In the traditional methodology for the development of distributed applications described in previous section, the application uses the basic event service to implement interactions between the components. In general, an application may require a richer set of distributed system services. For example, in bidding applications, we may want to constrain the various components to bid in a mutually exclusive manner. So, we might want to use a service that provides mutual exclusion functionality. We may also need a termination detection algorithm to detect when the bidding is over. To isolate the designer from the intricacies of a distributed system, one can provide a library of distributed algorithms implementing different types of services.

In implementing such an approach to provide a library of distributed algorithms, the designers of the library algorithms are faced with two opposing forces: One is to develop generic reusable algorithms that can be used in a wide variety of applications. On the other hand, applications may require algorithms to meet stringent performance constraints, which may force the designer to develop customized versions of the algorithms. The following

examples illustrate this tradeoff:

• Distributed algorithms often do not make any assumption regarding the application and are therefore conservative in nature. However, application components may follow a specific communication pattern or topology. For example, in a tele-teaching application, sessions or groups may be formed for different purposes with varying number of participants. Each such group may follow a specific communication pattern (for example, an answer message may be sent only in response to a question message) or a topology (for example, a ring or a star), which the underlying algorithms may be able to exploit. Thus, a straightforward use of a generic algorithm may not be efficient and this may force the designers to come up with their own implementations.

• To broaden the applicability, designers of mutual exclusion algorithms often work under the "pessimistic" assumption that the application components may request critical section entries in any order and include the communication necessary to ensure mutual exclusion. While this assumption may be true in general, in a specific application the components may issue requests in a specific order. For example, in a tele-conferencing application, there may be several operating phases, and in a particular phase, the participants may request access to a shared document in a cyclic manner. Or the application structure may impose a partial ordering on the entries itself. For example, it may be the case that the application components are divided into several clusters and the application may itself restrict at most one component in a cluster to access the shared resource (mutual exclusion is only required between clusters). In such cases, one might be able to take advantage of this application information to reduce the number of messages.

As an example, one way to implement mutual exclusion in a distributed environment is through the use of permissions. The process that wants to use a resource section issues a request to use it and waits until all other processes give it permission to do so. For example, if process 2 in Figure 3.1 wants to use a resource, it sends a request message to all other processes.

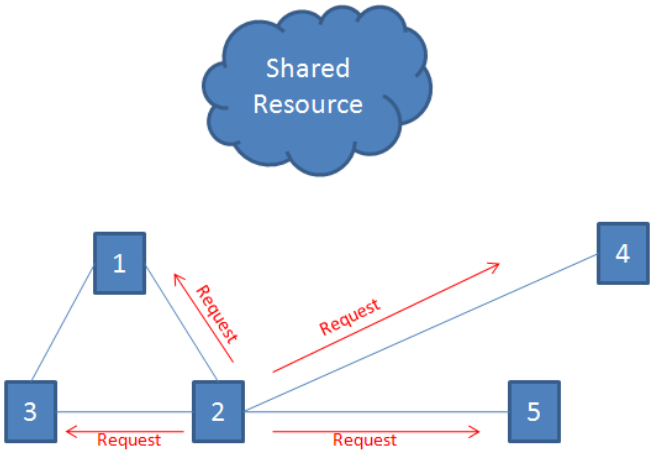

Figure 3.1: *Mutual exclusion example - sending of Request messages*

Other processes respond with an acknowledgement (see Figure 3.2). After receiving all the permissions, process 2 can use the resource.

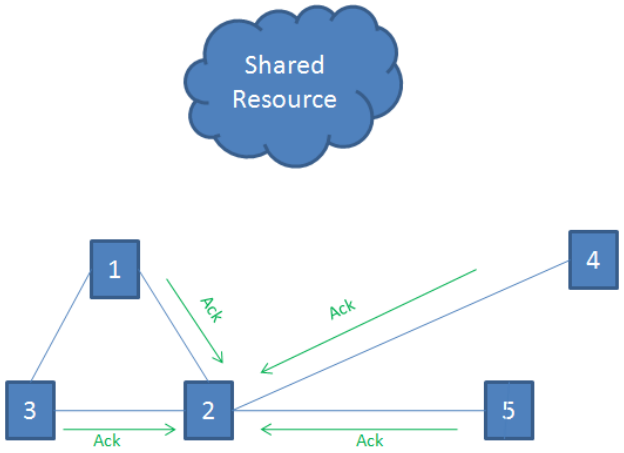

Figure 3.2: *Mutual exclusion example - receiving of Ack messages*

When the resource is no longer needed, process 2 notifies other processes that the resource has been released by it (see Figure 3.3).

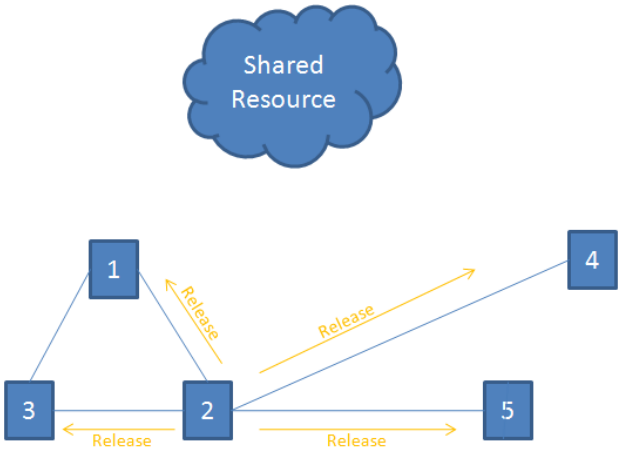

Figure 3.3: *Mutual exclusion example - sending of Release messages*

But application itself may impose a certain order on processes to use a resource. For example, it might be the case that processes 1, 2 and 3 access the resource in a cyclic manner: 1,2,3,1,2,3,... Then, if process 1 is requesting the resource use, processes 2 and 3 will not be interested in accessing the resource at the same time. So in generic algorithm, if process 2 wants to use the resource, it will still send requests messages to all the processes, but requests do not need to be sent to processes 1 and 3 (see Figure 3.4). So, generic algorithm is going to be inefficient.

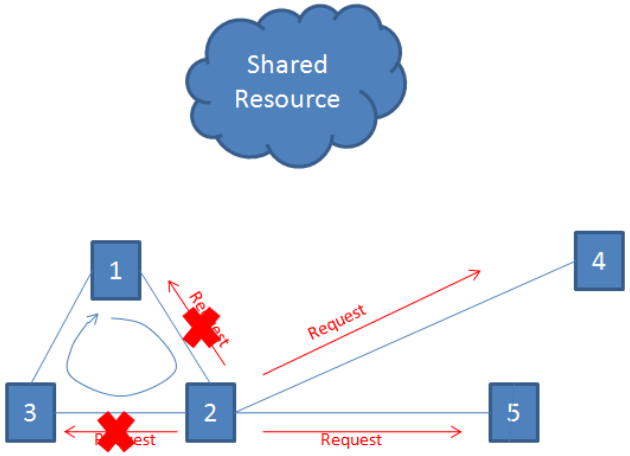

Figure 3.4: *Mutual exclusion example - taking ordering information into account*

• As another example, a number of algorithms have been proposed for termination detection. In general, an algorithm for termination detection has to determine that all components are passive and all channels are empty. In a particular application, however, the passive states of the components may be dependent on each other. As a simple example, if component A only communicates with B and performs tasks assigned by B only, then A will always be passive whenever B is passive. Such dependencies can be used to reduce the number of components to be polled for passive states.

For example, for star topology shown in Figure 3.5, if process 2 would like to determine termination, it would send a marker message to processes 1, 3 and 4.

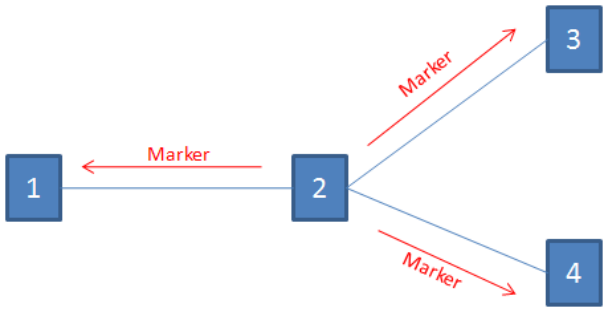

Figure 3.5: *Termination detection example - sending of Marker messages*

The processes respond with either Done message (if they are passive) or Continue message (if they are still active) as in Figure 3.6. If process 2 receives Done message from all the processes and it remained passive since it sent the marker messages out, termination is detected.

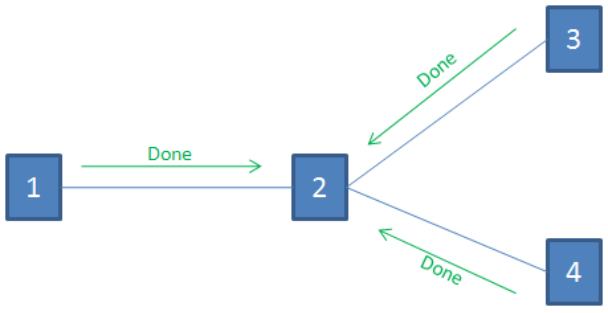

Figure 3.6: *Termination detection example - receiving of Done messages*

But, in a particular application, if component 1 only communicates with component 2 and performs tasks assigned by 2 only, then 1 will always be passive whenever 2 is passive.

For example, Answer message could only be sent in response to a Question message (see Figure 3.7). Then, if process 2 received all the answer messages from process 1 and is passive, then process 1 will be passive too (since it can only be activated by a message from process 2).

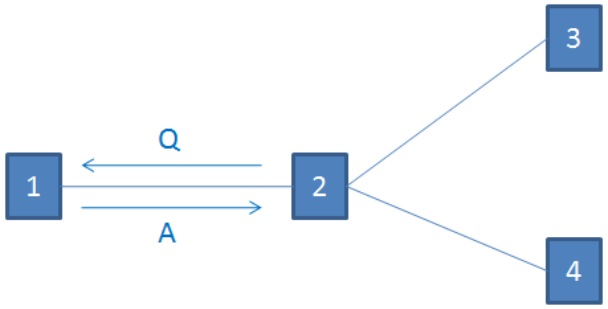

Figure 3.7: *Termination detection example - ordering on Q/A messages*

Then, in generic algorithm, process 2 will still send a marker message to all processes. But the message to process 1 is not needed (see Figure 3.8).

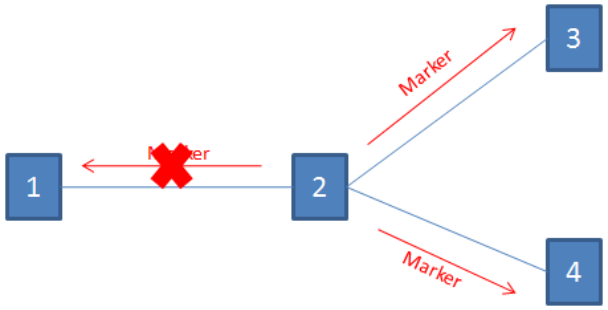

Figure 3.8: *Termination detection example - taking ordering information into account for Marker messages*

Message Done from process 1 is not needed either (see Figure 3.9). So, generic algorithm is going to be inefficient. Such dependencies as described above can be used to reduce the number of components to be polled for passive states.

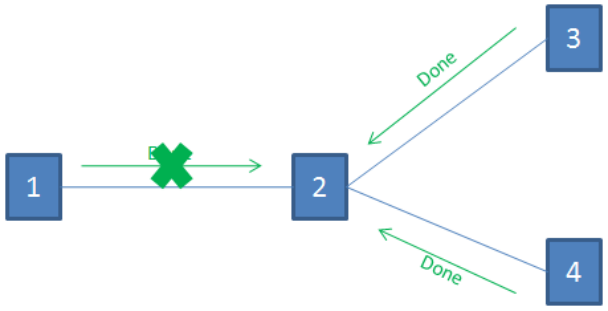

Figure 3.9: *Termination detection example - taking ordering information into account for Done messages*

In each of the cases discussed above, one can take advantage of the application structure to optimize the performance of the distributed algorithms. So, if the algorithms in the library are used as-is, the resulting implementations may be inefficient. In such cases, an application developer may be tempted to develop algorithms from scratch suited to the application. Such conflicts are especially problematic in product line software architectures wherein a fixed middleware infrastructure may have been developed for a class of applications (e.g., class of tele-teaching applications, or the class of avionics applications). In such cases, all applications in the product line (irrespective of their size and structure) may be forced to use the same set of underlying distributed algorithms to satisfy their requirements.

• In the Boeing Bold Stroke system, event service middleware is used to perform event notifications.[12,22] However, Bold Stroke developers have identified several places where specialized versions of the middleware are desirable. For instance, one such specialization replaces event-channel based notification by a direct method invocation, which is an order of magnitude more efficient. For small scenarios, such as shown in Figure 3.10, such optimizations can be identified and performed manually. However, it is tedious and error prone to do this manually for large systems, such as shown in Figure 3.11.

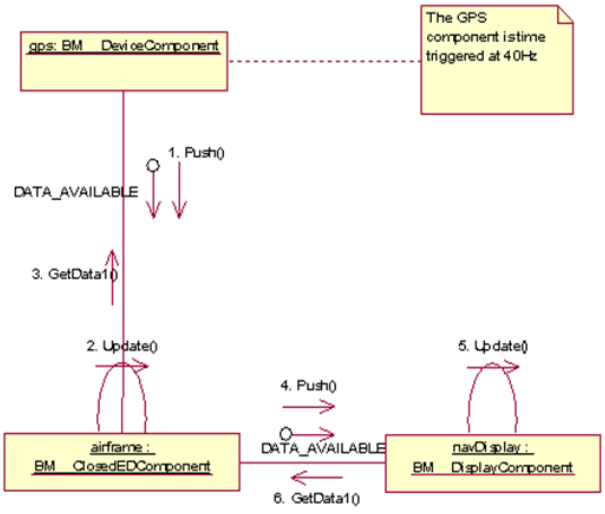

Figure 3.10: *Small size scenario*

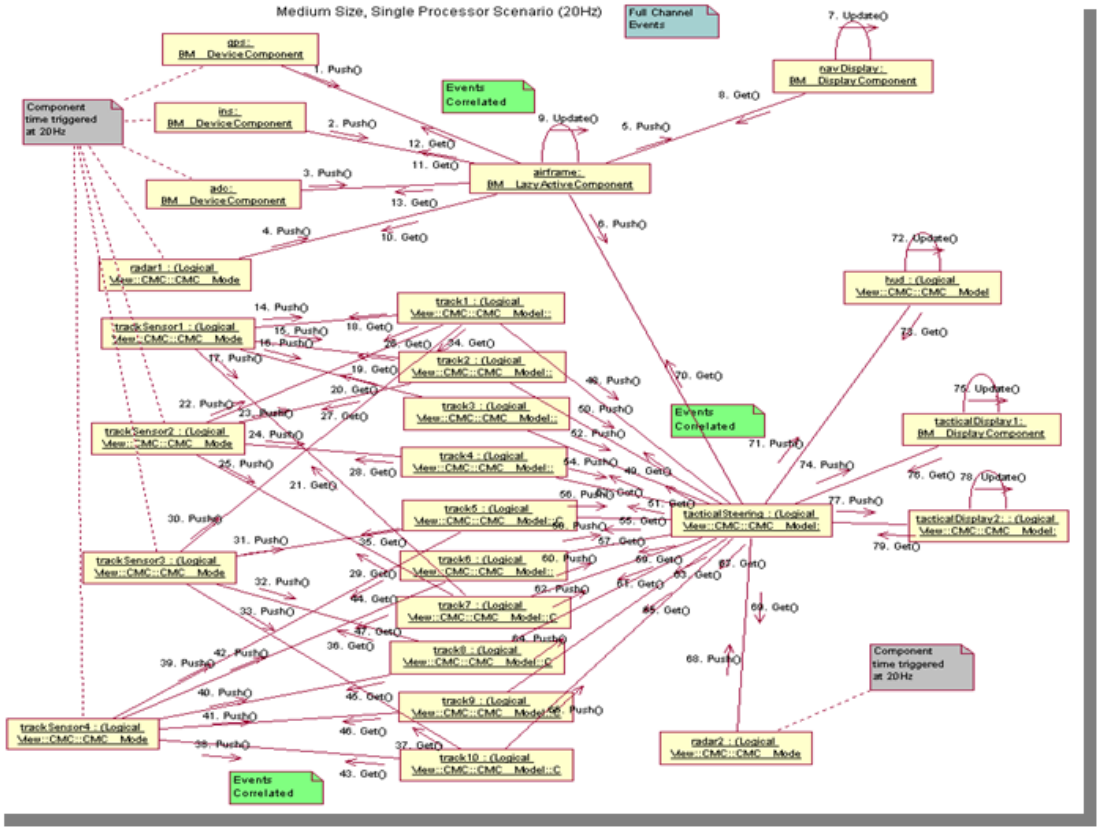

Figure 3.11: *Medium size scenario*

3.2 Related work

To ease the design and development of distributed systems, several frameworks have been proposed for component-based development[1–10,21,23]. The goal of such frameworks is to isolate the developers from lower-level details. For example, CORBA Component Model (CCM) Specification[24] defines component model, CCM implementation framework, and deployment mechanism. Cadena[21] is an integrated modeling environment for modeling and building CCM systems. It provides facilities for defining component types, assembling systems from component types, producing skeleton files implemented in Java, and generating deployment and configuration code. Cadena is also providing basic event services to implement interactions between the components. Other frameworks provide a middleware layer with distributed middleware services going beyond just providing a basic event notification service. However, even if a middleware layer is present in such frameworks to provide middleware services, the services provided are generic and might not be efficient for applications exhibiting, for example, certain patterns of resource usage or ordering of events. Our main interest, along with the ease of design and development, is to incorporate optimization mechanisms into a framework for distributed systems development.

The problem of optimization has been studied for specific services. Algorithms have been proposed for various problems for specific topologies and assumptions. For example, a number of optimized algorithms for mutual exclusion have been proposed for topologies such as ring, trees, and complete networks, as well as for hierarchically arranged application components[25–29].This approach, however, requires a new algorithm for every situation.

Researchers have also addressed the problem of optimization in specific domains. A number of concurrency control, transaction processing and multicasting algorithms which take advantage of *application semantics* (e.g., *conflict relation* among operations) have been proposed[3,7,16–19,30–33]. For example, if multiple replicas of a distributed database need to be updated, the updates need to be performed in the same order. But other unrelated operations could be done in any order. So, in an application, if messages of type $t1$ do

not conflict with those of type $t2$, then the message ordering algorithm does not have to order the delivery of messages of type $t2$ together with the messages of type $t1$. This approach, however, does not take into account ordering information that application itself may impose. So, the algorithm will implement this conflict relation irrespective of the order in which messages are issued by the application. Our work is complementary to this as it targets application semantics by analyzing the structure of the application.

Agbaria et al.[15] explored a similar approach for application-driven checkpointing wherein checkpoints are inserted at specific points in the application to eliminate channel state recording (thus, the checkpointing algorithm is essentially compiled away). But checkpoints need to be inserted manually and it is not automatic. Manual work is tedious and error prone especially for larger systems. We, on the other hand, would like to automate the optimization process.

Distributed algorithms often use notions defined on the state of the application. One interesting work in this regard is the HOPE optimistic programming system[34]. In optimistic programming, an algorithm makes an optimistic assumption about the application and verifies at a later point whether it is true or not. HOPE allows the programmer to explicitly assert such assumptions in the algorithm. These techniques, however, are used to increase concurrency. We are interested in optimization.

Some algorithms have been designed to dynamically monitor the application behavior and perform optimizations on-the-fly. COAL is an example of such an algorithm which keeps track of where the events are being sent and the events that have already been delivered when an event is being sent[35]. The idea of this monitoring is to dynamically guess the behavior on that particular execution. However, this approach does not take into consideration application knowledge. More optimizations can be done by utilizing application knowledge.

The problem of customizing programs has been studied extensively for sequential and parallel programs, wherein compiler techniques have been designed to analyze and optimize library routines statically. For example,[36] proposed the Broadway framework which is closely

related to our work. Broadway provides a framework for specifying optimization conditions for software libraries for parallel programs. The designer can define properties, their values of interest, how the library routines update these properties and the optimization conditions in terms of these properties values.

A number of high-level abstractions for broadcasting, multi-party synchronization, and atomicity have been proposed[37–41]. Using such primitives makes a program more declarative and the intent of the programmer clearer. Although these primitives may allow some optimizations, their main goal has been to simplify programming. We, on the other hand, want to explore abstractions whose primary purpose is to enable customizations.

Although there has been work in specific contexts, there is a lack of tools and methodologies to systematically attack this problem. In most cases, either the algorithms and middleware are not customizable or tools are not available to automatically configure them.

In our earlier preliminary work, we have studied model-driven techniques to customize event service middleware[42–44], event ordering algorithms[45,46] and synchronization algorithms[47–49] in which some of the issues listed above were addressed in a limited manner (e.g., application representation and analysis, design of configurable middleware).

The first case study dealt with an instance where a middleware service is made customizable by exposing a set of configurable options[42]. This work was motivated by optimization issues in Bold Stroke event communication middleware. In traditional implementations, an event connection, say from port p1 to p2, is implemented via event service middleware wherein p1 connects as a producer and p2 connects as a consumer to an event channel. In general, the push path for event notification from p1 to p2 may use several event channel features such as subscriber lists, correlation to allow subscription for composite events such as "e1 and e2", and distributed notifications. A framework, FRAMES, was developed which offers a number of alternatives to implement each of these features[42]. There have been performed extensive experimentations to evaluate these mechanisms and it was found that each option outperforms the others under certain conditions.

A second case study involved customization of distributed algorithms for event ordering[46]. In there, the causal ordering algorithm described by Prakash et al.[35] was studied which operates by propagating the immediate dependency relation. By analyzing the application's communication structure, it may be possible to eliminate the propagation of dependency information in some cases. To accommodate such optimizations, a customizable algorithm was designed which takes two tables as parameters, a generation_rule table and a propagation_rule table[46]. These tables determine the dependency information to be computed and propagated in the algorithm. An analysis algorithm was designed that computes these tables by analyzing the application's communication structure, thereby ensuring that only the required dependency information is computed and propagated. It was shown that this customization can capture the optimizations for causal ordering proposed by Quaireau et al.[33].

A third case study dealt with complete synthesis of algorithms customized to specific contexts. An aspect oriented technique for synthesis of synchronization code was developed[47,48,50]. The approach is to factor out synchronization as a separate aspect, synthesize synchronization code and then compose it with the functional code.

In the three case studies discussed above, several concepts and mechanisms crucial in enabling customizations have been identified: exposing configurable options, leveraging application's communication structure, code transformation and synthesis tools. However, since the mechanisms were developed for specific algorithms, most of the artifacts are hard-coded in the customization tools.

Chapter 4

InDiGO framework

In this chapter, we present our infrastructure for distributed algorithms optimization. We start with the brief description of the framework capabilities. The rest is presented from the point of view of the three participating entities: an application developer, an algorithm developer and optimization tools. We describe the responsibilities of each entity, the steps that application and algorithms developers need to take in their developments to utilize InDiGO infrastructure, and the interfaces that optimization tools provide. We then summarize by listing the key points of this chapter.

4.1 Description of framework capabilities

In this thesis, we propose InDiGO, an infrastructure to optimize distributed algorithms. The capabilities of InDiGO include:

- *Infrastructure to capture application information*: Exploiting application information will require that the application itself be in a form amenable to analysis. We use the internal representation of an application in Cadena to construct an *application dependency graph* (ADG), and provide a mechanism to analyze the ADG for relevant information.
- *Customizable Algorithms*: To customize the algorithms, they must be in a form amenable to customization. We have developed a mechanism which allows a designer

to expose design knowledge related to the communication structure of an algorithm. This involves identifying the interaction sets used for communication in an algorithm, and defining the semantics of these sets in terms of queries supported by the analysis infrastructure.

- *Optimization Tools*: We have developed tools to analyze the ADG for optimization information. Algorithms are optimized by removing communication redundant in the context of a specific application. We perform *static optimization* to initialize the interaction sets, *dynamic optimization* to update the interaction sets at run-time, and *physical topology-based optimization* for efficient mapping of algorithm message passing to the physical topology.

4.2 Application developer perspective

In this section, we describe our framework from an application developer perspective. An application developer is responsible for designing components and assembling systems from components using the underlying services. This development process is described in section 2.2.

4.2.1 Identifying the required services

Cadena provides basic event services to implement interaction between components. We would like to extend the development methodology described in section 2.2 to provide support for a richer set of distributed middleware services. We need to provide a mechanism for the designer to indicate his interest in a service. One possibility is for the designer to explicitly include calls to the interface offered by the service in the application code. The other possibility is to provide the designer with a domain-specific language to annotate code to specify the requirements. We have adopted the second approach. We are currently using annotations which are sufficient for services studied in this work (the more general problem of a domain specification language for such annotations may be needed and is beyond the scope of this thesis).

So, if the application developer is interested in using an underlying middleware service in his application, we require the developer to go through the following step:

• Application designer annotates the component code to specify the required library algorithms. For example, for the mutual exclusion service, we require the designer to annotate the code with regions which have to be executed exclusively. An example of such a file with annotations is shown in Listing 4.1.

```
1  public class bidComp extends Component {
2
3     ...
4
5     public void process(Object data_, Port inPort_){
6
7        if (data_ instanceof Message){
8           if (data_ instanceof BidMadeMessage){
9
10             /**@cs_request
11             ...
12             /**@cs_release
13
14          }
15       }
16    }
17 }
```

Listing 4.1: *Annotated Java skeleton file*

As shown in Listing 4.1, the annotations for mutual exclusion service are */**@cs_request* and */**@cs_release* and correspond to the beginning and the end of a critical section that needs to be accessed in a mutually exclusive manner. We allow multiple critical sections, but we do not currently support nested critical sections. If there is more than one critical section, we require the designer to add the critical section id to the annotations. For example, critical section 2 will have to be annotated with */**@cs_request_2* and */**@cs_release_2* annotations. Then, there will be a separate mutual exclusion service running per critical section.

The designer is provided with a list of available services. The interface of each service specifies the annotations to insert into code to use the specified service. As algorithm developers provide new services, the list of available services along with their interfaces is updated and provided to the application developers.

In this work we looked at three services, namely: mutual exclusion, termination detection and total ordering. The annotations to use for mutual exclusion service are */**@cs_request* and */**@cs_release* and must be inserted accordingly in the beginning and at the end of a critical section that needs to be accessed in a mutually exclusive manner. The annotations for termination detection service are */**@active* and */**@passive* and need to be inserted accordingly to mark the beginning and the end of the code region that can potentially activate another component through sending a message. Annotation for total ordering service is */**@total* and needs to be inserted on the line where a message is sent that need to be totally ordered.

Once the designer has annotated the component code, the deployment tools integrate it with the algorithm code. This integration involves adding the algorithm code to the code base to be deployed, generating the code for the application components to interact with the algorithm code, and code to initialize the services. For example, the annotations in Listing 4.1 are used to transform the component code to include calls to the mutual exclusion module to enter and exit the critical section.

4.3 Application dependency graph

In order to extract optimization information from application, we need to represent application in such a form that it would be possible to analyze it. We construct an *application dependency graph* (ADG) representing the structure of the application. The ADG is derived from two sources of information. First, Cadena component specification file describes component types and ports that each component type has. An example of such a file is shown in Listing 2.2. Application assembly information is stored in Cadena assembly specification file. This file describes application component instances and interconnections between them. An example of such a file is shown in Listing 2.3. Second, in Cadena, a component property specification (CPS) file is associated with each component to specify internal dependencies between the input and output ports of the component and behavior of each event handler. A fragment of the CPS file for component `bidComp` is shown in Figure 4.1 (the full grammar is given in Appendix A). The case statement in the CPS file specifies how an incoming event is processed. For example, the case statement in Figure 4.1 specifies that when an event on port `nextToBid` is received, if the variable `bidstate`'s value is `continueBid`, then the component executes actions `cs_request, bid, cs_release`, and emits an event on port `bidMade`. If the value is `stopBid`, then the component becomes `passive` and no event is emitted.

```
component bidComp {
    Mode bidstate;
    dependencies {
        nextToBid →
            case bidstate of {
                continueBid: cs_request; bid; cs_release; bidMade
                stopBid: passive
            }
    }
}
```

Figure 4.1: *Example of a CPS file for component bidComp*

As discussed earlier, the application developer annotates the component code to specify algorithm usage (e.g., annotations to request/release a critical section). One of the capabilities to be built is a tool to derive the CPS files from the annotated Java files for each component. Each annotation then would be represented by a corresponding action in the CPS file, which in turn will be represented by a node type in the ADG graph and the node type will reflect the original annotation. At present, we are able to derive this information for a restricted subset of the CPS grammar (for example, we do not handle nested case statements). Therefore, for some of the experimental studies, we had to manually specify the CPS files.

Let $C1, C2 \in App.components$. We say that $(p1, p2) \in App.connections$, where $p1 \in C1.outp$ and $p2 \in C2.inp$, if $p1$ is connected to $p2$ in App. We construct an ADG from the application specification and the CPS files as follows. First, we extract port id/port name information from the component specification file. Component specification file stores information in XML format. So, we use standard Java based parser to extract elements by tag name utilizing DocumentBuilderFactory, DocumentBuilder and Document classes from javax.xml.parsers library. Each component element describes a component type along with its ports. We extract port id/port name information for each component type specified in the component specification file and store it in memory. Next, we get component instance id/component instance name information from the assembly specification file. Assembly specification file also stores information in XML format. Again, we use standard Java based parser to extract elements by tag name. Each component instance element describes a component instance along with its ports. We extract component instance id/component instance name information for each component instance specified in the assembly specification file. Using the component instance id/component instance name information received from assembly specification file, we obtain $App.components$. Utilizing port id/port name information for each component type received from the component specification file, we obtain $C.inp$ and $C.outp$ for each $C \in App.components$. Each port bindings element in the

assembly specification file describes a connection between two ports. Using port binding information received from assembly specification file, we obtain *App.connections*. Each port in *C.inp* and *C.outp* for each $C \in$ *App.components* is a node in the ADG graph. In addition, each action and case statement in the CPS file of a component instance is a node (each case statement corresponds to a choice node). For component C, let *C.in_node*, *C.out_node* and *C.internal_node* denote the nodes corresponding to input ports, output ports and internal actions respectively. Internal actions are the actions listed in a CPS file. Each case statement in a CPS file corresponds to a choice action. Each dependency in a CPS file represents an event handler. Each dependency begins with specifying the port for which the event handler is listed next after the right arrow. Intuitively, each node represents an action (with nodes in *C.in_node*, *C.out_node* and *C.internal_node* representing receive, send and internal actions of C respectively). The edges, representing ordering between actions, are defined as follows: Let $C1$ and $C2$ be two components in the application and v^p to denote the port corresponding to node v. There is an edge $(v1, v2)$ in the ADG if

(a) $v_1 \in C1.out\_node$, and $v_2 \in C2.in\_node$, and $(v_1^p, v_2^p) \in$ *App.connections*.

(b) $v_1 \in C1.in\_node$ and $v_2 \in C1.internal\_node$ and event handler for v_1^p contains v_2 as the first action.

(c) $v_1, v_2 \in C1.internal\_node$ and v_1 immediately precedes v_2 in the same event handler.

(d) $v_1 \in C1.internal\_node$, $v_2 \in C1.out\_node$, and v_1 immediately precedes v_2.

Rule (a) defines the inter-component edges whereas the other rules define the intra-component edges.

Each node in the ADG corresponds to an action (e.g., the node representing an input port corresponds to a receive action). An action a in the system is either receipt of a message, send of a message, or an internal action. Occurrences of an action that can take place in a system are modeled as *events*. We use a^x to represent the x^{th} occurrence of a and *e.action* to denote the action corresponding to event e. We now define the set of possible executions of an ADG. An execution of ADG is a sequence, $s_0, e_0, s_1, e_1, \ldots$, where each s_i

is a state and e_i is an event. The state s_i is represented by the set of nodes in ADG which are enabled. Initially, s_0 contains the nodes representing the init ports (we use the Cadena's property specification mechanism to tag a port as an init port; such ports are used to trigger the start of the system). The outgoing edges for all nodes (except the choice node) have an AND-semantics; that is, control is transferred to all nodes reachable via the outgoing edges. Thus, when an event e_i representing a non-choice node a executes, s_{i+1} is obtained by removing a from s_i and adding nodes reachable from a via all outgoing edges. For a choice node, a node for one of the outgoing edges of a is added. The incoming edges for a node in an ADG have an OR-semantics, that is, the action for the node is enabled whenever control is transferred along any of its incoming edges. Once a node is added to a state, it is enabled and can be executed.

Example

As an example, let's consider a component shown in Figure 4.2.

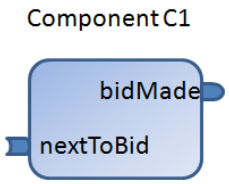

Figure 4.2: *Example of a component with two ports.*

The component in Figure 4.2 has two ports: an input port *nextToBid* and an output port *bidMade*. So, there will be two nodes in the application graph which will correspond to the two ports as in Figure 4.3.

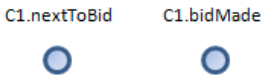

Figure 4.3: *Part of an application dependency graph that corresponds to the component in Figure 4.2.*

Next, consider two components and connections between them as shown in Figure 4.4.

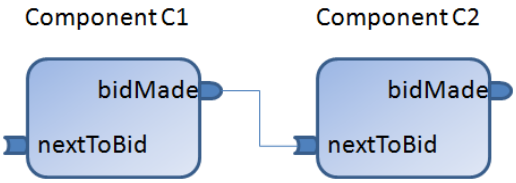

Figure 4.4: *Example of two connected components.*

In Figure 4.4, the output port *bidMade* of Component C1 is connected to the input port *nextToBid* of Component C2. So, the application graph shown in Figure 4.5 will correspond to the scenario of Figure 4.4 with four ports and a connection between two of them.

Figure 4.5: *Part of an application dependency graph that corresponds to the component in Figure 4.4.*

Next, let the handler for events on input port *nextToBid* of component type C look like in Listing 4.2.

```
1  if (data_ instanceof BidMadeMessage){
2
3     ...
4     /**@cs_request
5     ...
6     // bid
7     ...
8     /**@cs_release
9     ...
10 }
```

Listing 4.2: *Handler sketch for port* nextToBid.

Let C1 component be an instance of component type C. When an event on input port *nextToBid* of C1 is received, the handler shown in Listing 4.2 processes it. Critical section access is requested next. Critical section is relased after critical section code ends. According to the rules specified for the creation of an application graph, two more nodes *C1.cs_request* and *C1.cs_release* corresponding to *request to enter critical section* and *critical section release* actions will be added to the graph. Also, edges from the input port *C1.nextToBid* to the *C1.cs_request* node and from the *C1.cs_request* node to the *C1.cs_release* node will be added. The portion of the application graph will then look like in Figure 4.6.

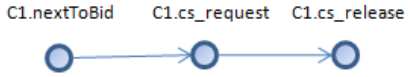

Figure 4.6: *Part of an application dependency graph that corresponds to the component in Figure 4.2 with internal connections.*

4.3.1 Query interface for application dependency graph

Now, that we have an application dependency graph, we need a mechanism to derive application specific information from it. We achieve that by running queries on the application dependency graph.

4.3.1.1 Basic queries

We have identified a set of basic queries useful in a number of algorithms that provide us with the information regarding the structure of the application (ordering of events). This set includes the following queries:

- *precede(a,b)* is true iff $\forall\ x,\ a^x$ occurs before b^x in all executions of the ADG.
- *alternate(a, b)* is true iff $\forall\ x,\ a^x$ occurs before b^x and b^x occurs before a^{x+1} in all executions of the ADG.
- *exclusive(a,b)* is true iff both a and b are not simultaneously enabled in all reachable states.
- *absence(a,b,c)* is true iff in all executions of the ADG, $\forall\ x$, $(e_x.action = b) \Rightarrow (\exists y < x, e_y.action = a$ and $\forall z,\ y < z < x,\ e_z.action \neq c)$. Informally, it states that whenever b occurs, a has already occurred and c does not occur between these occurrences of a and b.
- *exclusive(a,b,cond)* is true iff both a and b are not simultaneously enabled in all reachable states in which *cond* is true. $precede(a, b, cond)$, $alternate(a, b, cond)$ and $absence(a, b, c, cond)$ are defined similarly.
- *exclusive(cond1,cond2)* is true iff both conditions *cond*1 and *cond*2 are not simultaneously true in all reachable states.

To answer these queries, our first approach was to develop a separate algorithm per query. An algorithm would analyze the ADG and get a yes or no answer. A better approach is to use model checking to answer the queries precisely. Both of our approaches are described in the following sections.

4.3.1.2 Arguments to queries

Our framework currently supports several argument types that can be used in queries. One type is the arguments that represent actions. Actions correspond to nodes in ADG graph. The framework also supports arguments with conditions that are based on the state of the system at run time. A condition describes that some component is in a certain state (some node in ADG is enabled). The condition is first translated into which node c in ADG is enabled and then the query is run taking into account that c is enabled. For instance, we might want to know if actions a and b are exclusive when component C_i is in active state. The condition will be translated into $C_i.active$ is enabled. So, the query to answer is whether a and b are exclusive when $C_i.active$ is enabled.

Other types of arguments to basic quieries are conditions on counters. Developers can specify counters, rules to update the counters and use conditions on these counters as arguments to basic queries. The counters then will be associated with each component in application. We denote *App.counters* a set of counters per application *App*. Then, every counter $counter \in App.counters$ will be associated with each component $C \in App.components$ so that every component C has a counter *counter* associated with it. Counters are specified in a counter file. Counter file lists counter names one name per line. An example of such a counter file is shown in Listing 4.3 and specifies one counter in_cs. *in_cs* counter represents how many times a component entered and exited a critical section.

```
1 in_cs
```

Listing 4.3: *Counter file that specifies one counter in_cs.*

The rules to update this counter are shown in Listing 4.4.

```
1 if cs_request.enabled then in_cs.increment
2 if cs_release.enabled then in_cs.decrement
```

Listing 4.4: *Update rules for in_cs counter.*

These rules state that when a cs_request node of the ADG graph becomes enabled, then the counter needs to be incremented. Also, when corresponding cs_release node of the ADG graph becomes enabled, then the counter needs to be decremented.

The grammar for the counter update rules is given in Figure 4.7.

```
rules          ::=    ( rule )*
rule           ::=    <IF> <IDENTIFIER> "." nodeState <THEN> <IDENTIFIER> "." counterAction
nodeState      ::=    <ENABLED> | <DISABLED>
counterAction  ::=    <INCREMENT> | <DECREMENT>
```

Figure 4.7: *Grammar for counter update rules.*

The meaning of each rule is that if a node type, represented by the first identifier in the rule line, is in a certain state, enabled or disabled, then the counter, represented by the second identifier in the rule line, must perform an action, i.e. be incremented or decremented.

4.3.1.3 Development of algorithms to answer queries

Our first approach to answer a query was to develop a separate algorithm per query. The algorithm would analyze the ADG and get a yes or no answer. The algorithms we developed are based on depth first traversal of the graph. For example, the pseudo code for the algorithm used to answer *precede(a,b)* query is shown in Listing 4.5. We do not present the complete algorithm here, nor the correctness proof, because we decided to pursue the second approach. However, we describe it here for completness of presentation.

```
- reverse edges of the graph
- do depth-first traversal of the graph
  starting from node b
- stop following a path when the path reaches a,
  b, already visited node on this path, or init
  node and check for the following conditions:
    - if the path reaches b, halt and say NO
    - if the path reaches an already visited
      node on this path, halt and say NO
    - if the path reaches init node,
      halt and say NO
    - if the path reaches a, backtrack
      and continue the traversal
- if every path from b leads to a,
  halt and say  YES
```

Listing 4.5: *Pseudo code for the algorithm to answer precede(a,b) query.*

Although we have developed separate algorithms for the queries, these algorithms are conservative in nature. The responses to these queries do not yield any false positives and we were going to use only positive responses for optimizations. This has been done because the ADG is itself an abstraction of the application. One of the longer term goals in Cadena is to derive more accurate models from the component code (e.g., derivation of more detailed CPS files for Java files) which can yield more precise results.

4.3.1.4 Using model checking to answer queries

The second approach (the approach we adopted) to answer the queries is the use of a model checking tool on the ADG. It allows to answer the queries precisely as well as to verify more general properties.

For this, we use Spin model checker. This verification system was developed in the eighties and nineties and is freely available from the Web. Spin is one of the most widely used logic model checkers in the world.

The idea behind a model checking approach is that when the software itself cannot be verified exhaustively, we can build a simplified model of the underlying design that preserves its essential characteristics but that avoids known sources of complexity. The design model can often be verified, while the full-scale implementation cannot.

Our ADG graph represents a simplified model of a distributed system and captures the necessary application information that can be utilized for optimization.

The specification language that Spin accepts is called Promela. So, ADG needs to be translated into a Promela model that Spin can work on. This translation is achieved with the ADG-to-Promela conversion tool described in detail in section 4.5.2. Next, Optimizer tool, described in section 4.5.3, runs queries on the Promela model utilizing SPIN model checker to get a precise answer to each query.

4.4 Algorithm developer perspective

In this section, we describe our framework from an algorithm developer perspective. The task of a middleware developer is to provide a library of customizable distributed algorithms for common tasks such as mutual exclusion and termination detection.

4.4.1 Development of customizable algorithms

In this section, we discuss the design of customizable algorithmic base. One can follow several complimentary approaches to build a customizable distributed algorithms library. One approach is to develop a *set of algorithms* for the same problem, with each algorithm offering advantages over its alternatives in specific operational contexts. For example, this approach was followed in[42] to design a set of mechanisms for event communication whose relative performance are dependent on factors such as number and location of producers and consumers and publication rates. Tools were developed to analyze the application to select most appropriate mechanism for each event type. In this thesis, we follow a complimentary approach wherein we want to customize specific algorithms themselves (rather than selecting between algorithms). To enable customization, an algorithm developer must expose design knowledge pertaining to an algorithm in a form which can be leveraged for analysis. Algorithms have been designed with parameters such as maximum number of possible node failures or conflict relations to adapt their behavior. While parameters such as conflict relation exploit application semantics, they do not directly analyze the application structure for optimization.

In this thesis, we study mechanisms to expose knowledge related to the communication structure of an algorithm for possible optimizations.

4.4.1.1 Interaction sets

In our framework, we require the algorithm designers to adopt the following approach:

• For each algorithm *alg*, the designer first identifies the *interaction sets*, denoted by *alg.interaction_set*, which characterize its communication structure and specify the processes participating in each interaction. The designer then writes *alg* in terms of these sets. As seen later, this involves a simple transformation of the existing algorithms.

For example, the communication structure of Lamport's algorithm for mutual exclusion can be characterized by the following three interaction sets: *send_request_to* (SRT) is the set of processes to whom a request message has to be sent to enter critical section, *wait_for_ack* (WFA) is the set of processes from whom ack must be obtained prior to entering critical section, and *send_release_to* (SRelT) is the set of processes to whom a Release message must be sent. After the interaction sets are identified, algorithm developer designs an algorithm in terms of these interaction sets.

Alternatively, one can define sets with well defined meanings for a class of algorithms. The sets could could be general enough to be used in a number of distributed algorithms or could be very specific to a particular type of algorithms or even a particular algorithm.

In later sections of this chapter (section 4.4.2, section 4.4.3 and section 4.4.4) we look at several distributed algorithms for which we define algorithm specific interaction sets that describe the communication structure of those particular algorithms.

4.4.1.2 Membership criteria for interaction sets

• In the next step, for each interaction set *interaction_set*, the algorithm designer defines the *membership criterion*, denoted by *interaction_set.membership_criterion*, which specifies the criterion for a process to be involved in an interaction and so defines if a process is a part of the interaction set. The membership criteria must be defined in terms of the queries supported by the analysis infrastructure. This criterion is a problem-specific property a process must satisfy to participate in the interaction. As shown in the examples, this allows the sets to be defined in a problem-specific manner (rather than only in terms of physical topology).

For example, in the Lamport's algorithm for mutual exclusion, the membership criterion for SRT interaction set can be expressed in terms of *exclusive* basic query. Let *C.cs_request* denote the action of component C requesting access to enter the critical section. Let *in_cs* denote the "in critical section" counter and *C.in_cs = 1* denote a component C being in critical section. For simplicity, we have assumed that at most one application component is mapped to each site and will use C_i to denote the component mapped to site i. Let S denote the set of all components. SRT set is then defined in the following manner:

$SRT_i = \{j : C_j \in S \wedge \neg exclusive(C_i.in\_cs = 1, C_j.in\_cs = 1)\}$.

That is, processor j belongs to the SRT set of processor i if processor j could potentially request access to critical section concurrently with processor i.

The membership criteria need to be expressed in such a form that tools are able to parse it. For that reason, we require the algorithm developer to specify membership criteria for interaction sets in a standardized form. An example of such an input file for membership criterion of SRT interaction set is shown in Listing 4.6. It specifies the name of the interaction set, the name of the query to run and the arguments for the query in a comma delimited format. The full grammar for specifying membership criteria is given in Appendix B.

```
1 SRT.i, ALL, not exclusive, i.in_cs = 1, j.in_cs = 1
2 ...
```

Listing 4.6: *Sample of membership criteria input to optimizer.*

InDiGo framework supports a number of basic queries that can be used in defining membership criteria.

It is the responsibility of the algorithm designer to ensure the correctness of the algorithm written in terms of interaction sests with respect to membership criteria. Just like in traditinal algorithm development, algorithm designers are to provide correctness proofs with respect to algorithm properties.

4.4.1.3 Rules for dynamic updates to the interaction sets

• Finally, we allow the algorithm designer to further leverage the design knowledge and provide information for *dynamic updates* to the interaction sets.

In an algorithm, as a result of message passing, a process may obtain knowledge of the states of the application entities at other processes. For example, when process i receives a request message from process j in a mutual exclusion algorithm, it knows that an application entity at process j is in the requesting state. This information can be used to further constrain the interaction sets via dynamic update.

The designer exposes this information by identifying a set of assertions, $alg.app\_assert$, and two sets of control points, α_pos and α_neg, for each assertion $\alpha \in alg.app\_assert$. α_pos is the set of control points where α is known to be true and α_neg represents the control points where α may no longer be true. For each such assertion α, we declare a boolean variable $cond_\alpha$, and insert "$cond_\alpha$:= true" at each control point in α_pos and "$cond_\alpha$:= false" at each control point in α_neg. We will illustrate this concept using examples in the following sections.

As with the membership criteria for interaction sets, information on dynamic updates to the interaction sets needs to be expressed in such a form that tools are able to parse it. For that reason, we also require the algorithm developer to specify information on dynamic updates to the interaction sets in a standardized form. We require the algorithm developer to add a condition to the input file. This condition will describe which node in the abstraction model of our system will be enabled when a process is in a certain state. An example of such an input file for information on dynamic updates to the SRT interaction set is shown in Listing 4.7.

```
1 SRT.i, ALL, not exclusive, i.in_cs = 1, j.in_cs = 1, if k.state = requesting then k.cs_request = enabled
2 ...
```

Listing 4.7: *Sample of dynamic information input to optimizer.*

This example specifies the name of the set for which dynamic optimization information is

needed, the name of the query to run and the arguments for the query. It also specifies that if a process is in the requesting state, a cs_request node for this process will be enabled in the abstraction model for our system. This, in turn, will be translated into which node is enabled and the specified query will be run with an additional argument representing which node is enabled.

These dynamic rules are used during the system execution to further constrain interaction sets based on the information received through message passing of the executing system.

4.4.2 Mutual exclusion algorithm example

The Mutual Exclusion Problem

The mutual exclusion problem concerns a group of processors which occasionally need access to some resource that cannot be used simultaneously by more than a single processor, for example, some output device. Each processor may need to execute a code segment called a critical section, such that at any time, at most one processor is in the critical section (mutual exclusion), and if one or more processors try to enter the critical section , then one of them eventually succeeds as long as no processor stays in the critical section forever (no deadlock).

The above properties do not provide any guarantee on an individual basis since a processor may try to enter the critical section and yet fail, since it is always bypassed by other processors. A stronger property, which implies no deadlock, is no lockout: if a processor wishes to enter the critical section, then it will eventually succeed as long as no processor stays in the critical section forever.

More formally, an algorithm solves the mutual exclusion problem with no deadlock if the following hold:

- Mutual exclusion: In every configuration of every execution, at most one processor is in the critical section.
- No deadlock: In every admissible execution, if some processor is in the requesting state in a configuration, then there is a later configuration in which some processor is in the critical section.
- No lockout: In every admissible execution, if some processor is in the requesting state in a configuration, then there is a later configuration in which that same processor is in the critical section.

Description of Lamport's Mutual Exclusion Algorithm

One way to implement mutual exclusion in a distributed environment is through the use of permissions. The process that wants to enter a critical section issues a request to enter it and waits until all other processes give it permission to do so.

Consider the Lamport's permission based mutual exclusion algorithm[51] shown in Figure 4.8. In this algorithm n represents the number of sites. We will make no distinction between the concept of a process and the concept of a site in the distributed architecture. The meaning of different types of messages that the algorithm is using is as follows:

- When a process is attempting to enter its critical section, it sends a message of the *request* type to other processes.
- When it leaves its critical section, it broadcasts a message of *release* type.
- When a process P_i receives a message of the *request* type from a process P_j, it acknowledges the receipt with an *ack*.

Every process has a local clock and transmits messages that consist of three fields: (*message type, local clock, site number*). Therefore, each message carries its meaning along with timing information that will be used to ensure that the timing mechanism remains coherent. Each process also maintains a sorted queue of such received messages.

For each process we have the following local declarations:

- *ack* is an array of n elements of type *bool* and is used to keep track of permissions received from all other sites per request message sent;
- *request_sent* is a *boolean* and keeps track if a request to enter a critical section has been set;
- *my_seq_num* is a local clock (my sequence number); it is reset on receiving a new message in such a way as to ensure that every transmission date is earlier than any receipt date;
- *RQ* is an ordered request queue; the order is defined on a pair (*clock, site number*) as following: $(a, b) < (c, d) \equiv (a < c) \vee ((a = c) \wedge (b < d))$;

- *in_CS* is a *boolean* and specifies if a process currently is in critical section.

The algorithm is shown in Figure 4.8 and works as following:

- When a process wants to enter a critical section, it sends a request message to every process
- On receipt of a message (*request, k, j*) or (*release, k, j*) request queue is updated accordingly.
- A process P_i enters the critical section when its request is at the head of the queue and its timestamp is the oldest.

The algorithm is assumed to use logical clocks (clock update instructions are not shown).

Code for process P_i:

local variables:

```
01  bool ack[n]
02  bool request_sent ← false
03  int my_seq_num ← 0
04  queue RQ ← empty
05  bool in_cs ← false

06  :: (want_to_enter_CS ∧ !request_sent)
07       my_seq_num = local clock value;
08       broadcast(request, my_seq_num, i)
09       RQ.enqueue(<request, my_seq_num, i>)
10       request_sent ← true
11       ∀ j≠i ack[j] ← false; ack[i] ← true
12  :: receive(request, k, j)
13       RQ.enqueue(<request, k, j>)
14       if (k, j) > (my_seq_num, i) ack[j] ← true
15       send(ack) to j
16  :: receive(ack)
17       ack[j] ← true
18  :: receive(release, k, j)
19       RQ.dequeue()
20  :: (request_sent ∧ ∀ j ack[j] = true ∧ head() = (request, i))
21       in_CS ← true
22  :: (in_CS ∧ want_to_exit_CS)
23       broadcast(release, my_seq_num, i)
24       RQ.dequeue()
25       request_sent ← false
26       in_CS ← false
```

Figure 4.8: *Lamport's permission based mutual exclusion algorithm*

Customized Version

The communication structure of Lamport's algorithm can be characterized by the following three interaction sets:

- *send_request_to* (SRT) is the set of processes to whom a request message has to be sent to enter critical section.
- *wait_for_ack* (WFA) is the set of processes from whom ack must be obtained prior to entering critical section.
- *send_release_to* (SRelT) is the set of processes to whom a release message must be sent.

The algorithm written using these sets is shown in Figure 4.9. The algorithm is assumed to use logical clocks (clock update instructions are not shown). As can be seen, the transformation is simple.

Next, we define the *membership criteria* for these sets. From the interface offered by the algorithm, the designer knows the actions of the application pertaining to mutual exclusion. The criteria must be defined in terms of these actions only (as each algorithm is designed independently, the designer may not know of other events in the application). Let *C.cs_request* denote the action of component C requesting access to enter the critical section. Let *in_cs* denote the "in critical section" counter and *C.in_cs = 1* denote a component C being in critical section. For simplicity, we have assumed that at most one application component is mapped to each site and will use C_i to denote the component mapped to site i. Let S denote the set of all components. SRT set is then defined in the following manner:

$SRT_i = \{j : C_j \in S \wedge \neg exclusive(C_i.in\_cs = 1, C_j.in\_cs = 1)\}$.

That is, processor j belongs to the SRT set of processor i if processor j could potentially request access to critical section concurrently with processor i.

Both WFA_i and $SRelT_i$ are defined to be the same as SRT_i. It is the responsibility of the designer to ensure correctness of the algorithm with respect to these criteria. That is, any values assigned to these sets satisfying the specified criteria must ensure mutual

exclusion. For dynamic membership, we identify assertion "$enabled(C_j.cs\_request)$" (stating that component C_j is ready to enter critical section), and line 14 as the control point where it becomes true and line 20 when it becomes false. A call to procedure $update\_SRT$ is added (line 7) before the set is used. The code for this procedure is synthesized by the dynamic optimization rules.

Code for process P_i:

local variables:

```
01  bool ack[n]
02  bool request_sent ← false
03  int my_seq_num ← 0
04  queue RQ ← empty
05  bool in_cs ← false
06  :: (want_to_enter_CS ∧ !request_sent)
07       update_SRT()
08       my_seq_num = local clock value;
09       send(request, my_seq_num, i) to SRT
10       RQ.enqueue(<request, my_seq_num, i>)
11       request_sent ← true
12       ∀ j≠i ack[j] ← false; ack[i] ← true
13  :: receive(request, k, j)
14       RQ.enqueue(<request, k, j>)
15       if (k, j) > (my_req_num, i) ack[j] ← true
16       send(ack) to j
17  :: receive(ack)
18       ack[j] ← true
19  :: receive(release, k, j)
20       RQ.dequeue()
21  :: (request_sent ∧ (∀j | j ∈ WFA ∧ ack[j] = true) ∧ head() = (request, i))
22       in_CS ← true
23  :: (in_CS ∧ want_to_exit_CS)
24       send(release, my_seq_num, i) to SRelT
25       RQ.dequeue()
26       request_sent ← false
27       in_CS ← false
```

Figure 4.9: *Customized version of mutual exclusion algorithm*

4.4.3 Termination detection algorithm example

The Termination Detection Problem

The problem of detecting that a distributed algorithm has terminated is both important and non-trivial. Even if observation has shown that all the constituent processes of the algorithm are in a passive state - that is, are not active - this cannot be taken as a proof that the algorithm as a whole has terminated: for a process observed to be passive maybe reactivated by a message from a process that has not yet been observed and which then becomes passive. The problem would be simple if knowledge were available, at any instant, of a global state that took into account both the processes and the communication channels. Designing an algorithm for the problem thus comes down to designing a distributed control mechanism that will recognize a particular state of global stability, that of termination.

A process is said to be active if it is executing the text of its program and is passive if it is in any other state. A passive process can be either terminated, having completed its task, or waiting for messages from other processes. If all the processes are passive and no messages are in transit, the complete distributed algorithm is said to be terminated.

Description of Distributed Termination Detection Algorithm

We study the distributed termination detection algorithm that was first described in[52]. The algorithm is shown in Figure 4.10 and works as following. Processes are labeled $P_i, 0 \leq i \leq n$. We employ a *token* to transmit the values $quiet_p$. The token cycles through the processes visiting $P_{(i+1)mod\ n}$ after departing from P_i. A cycle is initiated by a process P_{init}, called the initiator. If the token completes the cycle (returns to P_{init}) after visiting all the processes and if all processes P return a value $quiet_P$ of *true* in this cycle, then the inititator detects termination, i.e. it sets *claim* to *true*. If any process q returns a value $quiet_q$ of *false* in a cycle, then the current cycle is terminated and a new cycle is initiated with q as the initiator. A process ends one observation period and immediately starts the next observation period when it sends the token. The algorithm, described next in detail, shows how $quiet_P$ is set.

There are no shared variables in a distributed system. However, for purposes of exposition we assume that *claim* is a shared global variable which has an initial value of *false* and which may be set *true* by any process. Such a global variable can be simulated by message transmissions; for instance, the process that sets *claim* to *true* may send messages to all other processes notifying them.

Two types of messages are employed in the termination detection algorithm:

- *(marker)*
- *(token, initiator)*

Each process has the following constants and variables. These will be subscripted, by i, when referring to a specific process i.

- *ic*: number of incoming channels to the process, a constant
- *idle*: process is idle
- *quiet*: process has been continuously idle since the token was last sent by the process; *false* if the token has never been sent by this process
- *have_token*: process holds the token
- *init*: the value of the initiator in the *(token, initiator)* message last sent or recieved;

undefined if the process has never received such a message

- m: number of markers received, since the token was last sent by the process

Initial conditions

- The token is at P_k
- m_i = the number of channels from processes with indices greater than i, for all i, i.e., the cardinality of the set, $\{c \mid c \text{ is a channel from } P_j \text{ to } P_i \wedge j > i\}$
  (This initial condition is required because otherwise, the token will permanently stay at one process)
- $quiet_i = false$, for all i
- $have\_token_i = true$ for $i = 0$; $false$ otherwise
- $init_i$ is arbitrary, for all i

Code for process P_i:

```
01  :: receive(marker)
02      m ← m + 1
03  :: receive(app)
04      if (quiet)
05          quiet ← false
06  :: receive(token, initiator)
07      init ← initiator
08      have_token ← true
09  :: (have_token ∧ (ic = m) ∧ idle)
10      if (quiet ∧ (init = i))
11          claim ← true // termination detected
12      else if (quiet ∧ (init ≠ i)) // continue old cycle
13          m ← 0
14          send marker to all neighbors
15          have_token ← false
16          send(token, init) to P_(i+1)mod n
17      else if (!quiet) // initiate new cycle
18          m ← 0
19          quiet ← true
20          init ← i
21          send marker to all neighbors
22          have_token ← false
23          send(token, init) to P_(i+1)mod n
```

Figure 4.10: *Distributed termination detection algorithm for an arbitrary topology*

Customized Version

The communication structure of the algorithm shown in Figure 4.10 can be characterized by the following three interaction sets:

- *send_marker_to* (SMT) is the set of all neighbor processors to which a marker message has to be sent.
- *wait_response_from* (WRF) is the set of all neighbor processors from whom a marker message is to be received.
- *send_token_to* (STT) is a singleton set consisting of the id of the next processor to send token to.

The algorithm written using these sets is shown in Figure 4.11. As in the case of Lamport's mutual exclusion algorithm, the transformation is simple.

Next, we define the *membership criteria* for these sets. For simplicity, we have assumed that at most one application component is mapped to each site and will use C_i to denote the component mapped to site i. Let Nbr_i denote the set of processes such that a component at processor i communicates with a component at processor j in *App*. We define SMT_i as following: $SMT_i = \{j : j \in Nbr_i)\}$. SMT set for processor i specifies neighbours of i. WRF_i is defined the same as SMT_i.

We define $STT(i)$ as following:

$$\begin{aligned} STT(i) = \ & \text{if } i < n-1 \\ & \qquad j = i+1 \\ & \text{else} \\ & \qquad j = 0 \end{aligned}$$

This specifies that i must send the token to $P_{(i+1) mod\ n}$ processor next. For dynamic membership, we identify assertion "$enabled(C_i.passive\_noact)$", stating that component C_i is passive and has not sent any messages to activate other components since its own last activation. Calls to procedures *update_SMT* (lines 14 and 24) and *update_STT* (lines 17 and 26) are added and called before the sets are used. The code for these procedures is synthesized by the dynamic optimization rules.

Code for process P_i:

```
01  :: receive(marker)
02       m ← m + 1
03  :: receive(app)
04       if (quiet)
05            quiet ← false
06  :: receive(token, initiator)
07       init ← initiator
08       have_token ← true
09  :: (have_token ∧ (WRF.size = m) ∧ idle)
10       if (quiet ∧ (init = i))
11            claim ← true // termination detected
12       else if (quiet ∧ (init ≠ i)) // continue old cycle
13            m ← 0
14            update_SMT()
15            send marker to SMT
16            have_token ← false
17            update_STT()
18            send(token, init) to STT
19       else if (!quiet) // initiate new cycle
20            m ← 0
21            quiet ← true
22            init ← i
23            update_SMT()
24            send marker to SMT
25            have_token ← false
26            update_STT()
27            send(token, init) to STT
```

Figure 4.11: *Customized version of distributed termination detection algorithm*

4.4.4 Total order algorithm example

The Total Order Problem

We have also studied an algorithm for total ordering of messages. The total order (TO) problem is about seeing the order of messages by different processes being the same across the distributed system.

Consider a group of processes multicasting messages to each other. Each message is always timestamped with the current (logical) time of its sender. When a message is multicast, it is conceptually also sent to the sender. In addition, we assume that messages from the same sender are received in the order they were sent, and that no messages are lost. The first assumption is equivalent to the assumptions of having a fully connected graph of links that have FIFO property.

Description of Total Order Algorithm Based on Lamport Timestamps

The algorithm is shown in Figure 4.12 and works as following. When a process receives a message, it is put into a local queue, ordered according to its timestamp. The receiver multicasts an acknowledgement to the other processes. Note, that if we follow Lamport's algorithm for adjusting local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgement.

The interesting aspect of this approach, is that all processes will eventually have the same copy of the local queue. Each message is multicast to all processes, including acknowledgements, and is assumed to be received by all processes. Recall also that we assume that messages are delivered in the order that they are sent. Each process puts a received message in its local queue according to the timestamp in that message. Lamport's clocks ensure that no two messages have the same timestamp, but also that the timestamps reflect a consistent global ordering of events.

A process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process. At

that point, the message is removed from the queue and handed over to the application; the associated acknowledgements can simply be removed. Because each process has the same copy of the queue, all messages are delivered in the same order everywhere. In other words, we have established totally-ordered multicasting.

```
    Code for process P_i:

    local variables:

01  time ← 0
02  queue ← empty

03  :: receive(mes, source, dest) from APP component
04      time ← time + 1
05      queue.add(time, mes, source, dest)
06      broadcast(mes, source, time)
07  :: receive(mes, source, t)
08      if (dest = i)
09          queue.add(t, mes, source, dest)
10      else
11          queue.add(t, ACK, source, i)
12      if (t > time)
13          time ← t
14      time ← time + 1
15      broadcast(ACK, source, time)
16  :: receive(ACK, source, t)
17      queue.add(t, ACK, source, i)
18      if (t > time)
19          time ← t
20  :: ∃ entry : queueEntry | queue.has(entry) ∧ entry.messageType = APP_MESSAGE ∧
21      ∀ p : process - i - entry.source | ∃ entryP : queueEntry | queue.has(entryP) ∧
22      entry.source = p ∧ entryP.t > entry.t
23          send(app, source) to APP component
24          ∀ e : queueEntry |
25              if (e.messageType = ACK ∧ e.t < entry.t)
26                  queue.remove(e)
27          queue.remove(entry)
```

Figure 4.12: *Total order algorithm based on Lamport timestamps*

Customized Version

The communication structure of the algorithm shown in Figure 4.12 can be characterized by the following three interaction sets:

- *send_app_message_to* (SAMT) is the set of processors to whom application message needs to be sent.
- *wait_ack_from* (WAKF) is the set of processors from whom acks must be obtained prior to delivering the message to application.
- *send_ack_to* (SAT) is the set of processors to whom acks must be sent when application message is received.

The algorithm written using these sets is shown in Figure 4.13.

Next, we define the *membership criteria* for these sets. For simplicity, we have assumed that at most one application component is mapped to each site and will use C_i to denote the component mapped to site i. Let S denote the set of all components. We define $SAMT_i$ as $SAMT_i = \{j : C_j \in S\}$. We define $WAKF_i$ as following:

$$WAKF_i = \{j : C_j \in S \wedge \neg exclusive(C_i.total, C_j.total)\}$$

SAT_i is defined to be the same as $WAKF_i$. It is the responsibility of the designer to ensure correctness of the algorithm with respect to these criteria. That is, any values assigned to these sets satisfying the specified criteria must ensure total ordering.

Code for process P_i:

local variables:

```
01  time ← 0
02  queue ← empty
03  :: receive(mes, source, dest) from APP component
04      time ← time + 1
05      queue.add(time, mes, source, dest)
06      update_SAMT()
07      send(mes, source, time) to SAMT
08  :: receive(mes, source, t)
09      if (dest = i)
10          queue.add(t, mes, source, dest)
11      else
12          queue.add(t, ACK, source, i)
13      if (t > time)
14          time ← t
15      time ← time + 1
16      update_SAT()
17      send(ACK, source, time) to SAT
18  :: receive(ACK, source, t)
19      queue.add(t, ACK, source, i)
20      if (t > time)
21          time ← t
22  :: ∃ entry : queueEntry | queue.has(entry) ∧ entry.messageType = APP_MESSAGE ∧
23      ∀ p | p ∈ WAKF | ∃ entryP : queueEntry | queue.has(entryP) ∧
24      entry.source = p ∧ entryP.t > entry.t
25          send(app, source) to APP component
26          ∀ e : queueEntry |
27              if (e.messageType = ACK ∧ e.t < entry.t)
28                  queue.remove(e)
29          queue.remove(entry)
```

Figure 4.13: *Customized version of total order algorithm based on Lamport timestamps*

4.4.5 Proofs for customizable algorithms

When algorithm developers design algorithms, they need to prove that their algorithms solve the problem at hand correctly. In our framework, we require algorithm developers to design distributed algorithms in such a way that communication structure of an algorithm is exposed for possible optimizations. That is done through the specification of interaction sets. Algorithm developers also provide membership criteria for interaction sets expressed in terms of basic queries that InDiGO infrastructure supports. Algorithm developers still need to provide a proof that their algorithms written in terms of interaction sets are correct and satisfy algorithm properties.

In this section we show that the task of algorithm developers does not become more difficult because of the fact that they need to follow a certain path in designing their algorithms. We will provide proofs for traditional algorithms as well as for algorithms written to be used in our framework and show that the difficulty level in providing proofs for customized algorithms does not increase. Specifically, we will take a look at the mutual exclusion algorithm presented earlier and show that mutual exclusion, no deadlock and no lockout properties of mutual exclusion algorithms are preserved. We will prove these properties for traditional and customized algorithms.

Formally, an algorithm solves the mutual exclusion problem with no deadlock if the following properties hold:

Property P 1. Mutual exclusion: In every configuration of every execution, at most one processor is in the critical section.

Property P 2. No deadlock: In every admissible execution, if some processor is in the requesting state in a configuration, then there is a later configuration in which some processor is in the critical section.

Property P 3. No lockout: In every admissible execution, if some processor is in the requesting state in a configuration, then there is a later configuration in which that same

processor is in the critical section.

The traditional version of mutual exclusion algorithm based on Lamport's timestamps is shown in Figure 4.8.

Theorem 4.1. *Mutual exclusion algorithm in Figure 4.8 satisfies mutual exclusion property P 1.*

Proof. We proceed by contradiction. Suppose that two processes P_i and P_j are in critical section CS at the same time. This means that both processes P_i and P_j broadcasted their request message to all other processes (line 8) and put their own requests in their local request queues RQ (line 9). Assume, without loss of generality, that the logical time t_i of P_i's latest request message is smaller than the logical time t_j of P_j's latest request message. Then, in order to enter critical section CS, P_j had to see in its local request queue RQ a message from P_i with logical time greater than t_j and hence greater than t_i (line 20). Then, FIFO property implies that P_j must have seen P_i's current request message when it was in CS. But then, in order for P_j to have its request on top of its local request queue RQ, P_j must have seen a subsequent release message from P_i so that the request message from P_i is removed from RQ (lines 18-19). This implies that P_i must have already left CS (lines 23-26) at the time P_j was in CS. We arrive at contradiction. □

Customizable version of mutual exclusion algorithm based on Lamport's timestamps is shown in Figure 4.9.

Theorem 4.2. *Mutual exclusion algorithm in Figure 4.9 satisfies mutual exclusion property P 1.*

Proof. We proceed by contradiction. Suppose that two processes P_i and P_j are in critical section CS at the same time. SRT set is defined as follows:

$SRT_i = \{j : C_j \in S \wedge \neg exclusive(C_i.in\_cs = 1, C_j.in\_cs = 1)\}$ (for simplicity, we have assumed that at most one application component is mapped to each site and we use C_i to denote the component mapped to site i; S denotes the set of all components.). Since processes P_i and P_j could be in critical section at the same time then, by definition of SRT set, it will mean that process P_j is in the SRT set of process P_i and that process P_i is in the SRT set of process P_j. Since process P_i is in the critical section, it had to send a request message to the processes in its SRT set (line 9), and that includes process P_j. Since process P_j is in the critical section, it had to send a request message to the processes in its SRT set (line 9), and that includes process P_i. Then both processes P_i and P_j put their own requests in their local request queues RQ (line 10). Assume, without loss of generality, that the logical time t_i of P_i's latest request message is smaller than the logical time t_j of P_j's latest request message. Then, in order to enter critical section CS, P_j had to receive an acknowledgement message from all processes in its WFA set (WFA set is defined the same as SRT set) and see in its local request queue RQ a message from P_i with logical time greater than t_j and hence greater than t_i (line 21). Then, FIFO property implies that P_j must have seen P_i's current request message when it was in CS. But then, in order for P_j to have its request on top of its local request queue RQ, P_j must have seen a subsequent release message from P_i so that the request message from P_i is removed from RQ (lines 19-20). This implies that P_i must have already left CS (lines 24-27) at the time P_j was in CS. We arrive at contradiction. □

Next, we will prove no lockout property P 3. No lockout property, in turn, implies no deadlock property P 2.

Theorem 4.3. *Mutual exclusion algorithm in Figure 4.8 satisfies no lockout property P 3.*

Proof. No lockout property follows from the fact that requests are serviced in the order of the logical times of their request messages. We argue that a request message with the smallest logical time among those for current requests eventually gets served. Since there

are only finitely many requests messages that are assigned logical times smaller than that of any particular request messages, an inductive argument then can be used to show that all requests are granted.

Suppose that process P_i has broadcasted a request message (line 8) and this message has the smallest logical time, t_i, among those for current requests. We argue that eventually the conditions for P_i to enter critical section CS (line 20) must become satisfied. First, P_i eventually will receive its own request message and put it in its local request queue RQ (line 9). Also, since request messages receive corresponding acknowledgments and the clock variables are managed using Lamport's timestamp mechanism, eventually P_i obtains a message from each of the other processes with a logical time greater than t_i. Finally, since P_i's request is the current request with the smallest logical time, any request with a smaller logical time must have already been served. That implies that eventually P_i receives release messages for the requests served and requests with smaller timestamps are removed from the queue (lines 18-19). In this way, all the conditions for P_i to enter CS (line 20) must eventually become satisfied. □

Theorem 4.4. *Mutual exclusion algorithm in Figure 4.9 satisfies no lockout property P 3.*

Proof. No lockout property follows from the fact that requests are serviced in the order of the logical times of their request messages. We argue that a request message with the smallest logical time among those for current requests eventually gets served. Since there are only finitely many requests messages that are assigned logical times smaller than that of any particular request messages, an inductive argument then can be used to show that all requests are granted.

Suppose that only a subset n of all processes N can request an access to critical section at some point. Suppose that process $P_i \in n$. Then, by definition of SRT set, SRT_i will contain all processes in $n - P_i$. Suppose that process P_i has sent a request message to all processes in its SRT set (line 9) and this message has the smallest logical time, t_i,

among those for current requests. We argue that eventually the conditions for P_i to enter critical section CS (line 21) must become satisfied. First, P_i eventually will receive its own request message and put it in its local request queue RQ (line 10). Also, since request messages receive corresponding acknowledgments and the clock variables are managed using Lamport's timestamp mechanism, eventually P_i obtains a message from each of the processes in its WFA set with a logical time greater than t_i. Finally, since P_i's request is the current request with the smallest logical time, any request with a smaller logical time must have already been served. That implies that eventually P_i receives release messages for the requests served and requests with smaller timestamps are removed from the queue (lines 19-20). In this way, all the conditions for P_i to enter CS (line 21) must eventually become satisfied. □

Theorem 4.5. *Mutual exclusion algorithm in Figure 4.8 satisfies no deadlock property P 2.*

Proof. Follows directly from Theorem 4.3. □

Theorem 4.6. *Mutual exclusion algorithm in Figure 4.9 satisfies no deadlock property P 2.*

Proof. Follows directly from Theorem 4.4. □

As can be seen, the proofs for customizable algorithms needed for our framework have the same level of difficulty as the proofs for traditional algorithms. Therefore, algorithm developers are not burdened with extra work when they design and prove customizable algorithms suited for InDiGO infrastructure.

4.4.6 Discussion

There are several points to note here:

• As can be seen, it is relatively easy to transform existing algorithms into the format required by the scheme described above (interaction sets) as these interaction sets are already being implicitly used in the algorithm design.

• Explicitly defining the membership criteria allows designers to capture the intended participants in each interaction in a problem-specific manner. For some algorithms, the interaction sets may coincide with the neighbor sets typically used in distributed algorithms. Explicit definition allows us to compute the interaction set values in a more meaningful way, including dynamically varying them based on application state. This becomes especially important when an application employs several algorithms with different interaction sets.

• The interaction sets can be viewed as defining a logical topology for an algorithm. In fact, in the description of many algorithms in the literature, the underlying graph is defined in a problem specific manner (e.g., neighbors in the graph are defined as those with which a process may actually communicate in the application rather than those in the physical topology). This logical topology is mapped to the physical topology during deployment. Our framework requires the definition of the logical topology to be made explicit using the interaction sets. As an application may require several algorithms, the logical topologies for each of the algorithms may be different, and furthermore, this may be different from the communication topology of the application and the physcial topology. Defining the interaction sets allows our optimization engine to further refine these sets before they are mapped to the physical topology.

• The interaction sets can be specific to an algorithm or to a class of algorithms. For example, we could also have followed an alternative approach wherein we define sets with well defined meanings for a class of algorithms, and ask developers to use these predefined sets to program the algorithms. For example, the set SRT could be common to mutual exclusion algorithms.

4.5 Optimization tools perspective

In this section, we describe our framework from the framework tools perspective. The infrastructure to automatically derive a representation of the application structure from the application specification, the use of model checking tools to answer the queries of interest on this representation precisely and optimization engine provide us with the tools necessary to analyze the application to obtain information useful in optimizing the algorithms.

4.5.1 ADG construction tool

In our framework, we specify an application as an ADG graph. We developed a tool to automatically construct an ADG graph. ADG construction is described in details in Section 4.3 on page 41. Inputs to the ADG construction tool are a Cadena module file that describes component types and provides information about ports that each component type has, a Cadena assembly specification file that describes component instances and connections between them in a specific application, and a component specification file that describes internal dependencies between the input and ouput ports of the component and behavior of each event handler.

ADG construction tool follows the rules for the construction of an ADG graph described in Section 4.3 and generates output file containing information about ADG nodes and connections between them in XML format. A sample of such an output for the application in Figure 2.3 is shown in Listing 4.8. A pictorial view of the graph generated by the ADG construction tool is shown in Figure 4.14 for the application in Figure 2.3.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<adgGraph>
<nodes>
<node nodeId="1" nodeName="C0.start" nodeStatusInit="true"/>
<node nodeId="2" nodeName="C0.nextToBid" nodeStatusInit="false"/>
<node nodeId="3" nodeName="C0.case" nodeStatusInit="false"/>
<node nodeId="4" nodeName="C0.cs_request" nodeStatusInit="false"/>
<node nodeId="5" nodeName="C0.bid" nodeStatusInit="false"/>
<node nodeId="6" nodeName="C0.cs_release" nodeStatusInit="false"/>
<node nodeId="7" nodeName="C0.bidMade" nodeStatusInit="false"/>
<node nodeId="8" nodeName="C0.passive" nodeStatusInit="false"/>
...
</nodes>
<edges>
<edge edgeId="137" edgeName="C0.start-C0.case">
<nodeFrom nodeId="1" nodeName="C0.start" statusOutcoming="AND"/>
<nodeTo nodeId="3" nodeName="C0.case" statusIncoming="OR"/>
<edge/>
<edge edgeId="138" edgeName="C0.nextToBid-C0.case">
<nodeFrom nodeId="2" nodeName="C0.nextToBid" statusOutcoming="AND"/>
<nodeTo nodeId="3" nodeName="C0.case" statusIncoming="OR"/>
<edge/>
<edge edgeId="139" edgeName="C0.case-C0.cs_request">
<nodeFrom nodeId="3" nodeName="C0.case" statusOutcoming="OR"/>
<nodeTo nodeId="4" nodeName="C0.cs_request" statusIncoming="OR"/>
<edge/>
<edge edgeId="140" edgeName="C0.case-C0.passive">
<nodeFrom nodeId="3" nodeName="C0.case" statusOutcoming="OR"/>
<nodeTo nodeId="8" nodeName="C0.passive" statusIncoming="OR"/>
<edge/>
<edge edgeId="141" edgeName="C0.cs_request-C0.bid">
<nodeFrom nodeId="4" nodeName="C0.cs_request" statusOutcoming="AND"/>
<nodeTo nodeId="5" nodeName="C0.bid" statusIncoming="OR"/>
<edge/>
<edge edgeId="142" edgeName="C0.bid-C0.cs_release">
<nodeFrom nodeId="5" nodeName="C0.bid" statusOutcoming="AND"/>
<nodeTo nodeId="6" nodeName="C0.cs_release" statusIncoming="OR"/>
<edge/>
<edge edgeId="143" edgeName="C0.cs_release-C0.bidMade">
<nodeFrom nodeId="6" nodeName="C0.cs_release" statusOutcoming="AND"/>
<nodeTo nodeId="7" nodeName="C0.bidMade" statusIncoming="OR"/>
<edge/>
...
</edges>
...
```

Listing 4.8: *Sample of ADG construction tool output.*

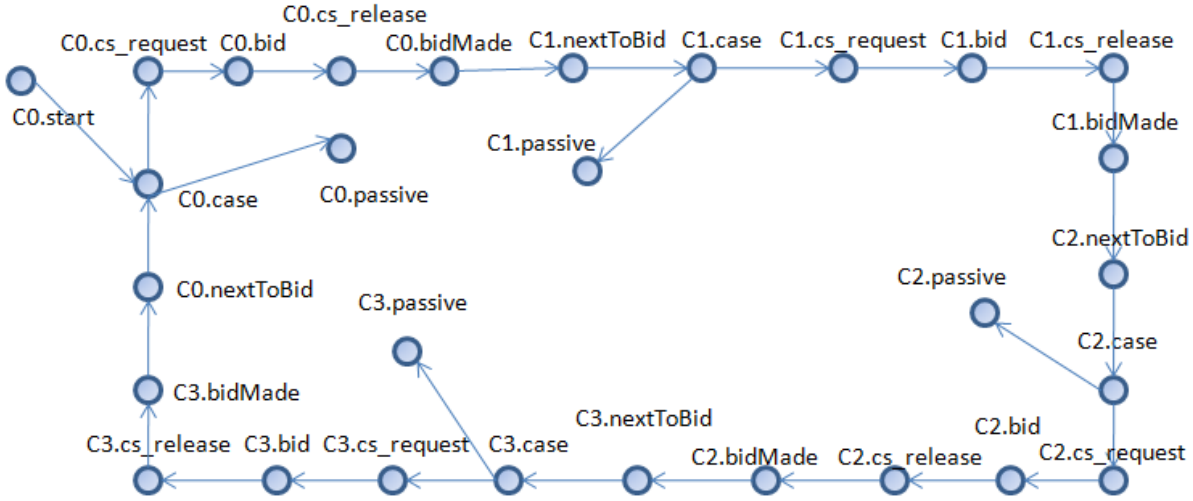

Figure 4.14: *A pictorial view of ADG graph*

4.5.2 Promela model construction tool

Spin models are written in Promela specification language. So, in order to utilize Spin model checker, we need to convert ADG graph into Promela model. Here, we describe the Promela model construction tool that we have developed.

ADG graph is stored in an XML file and the file describes the nodes of the graph and its edges. This XML file serves as an input to our translation tool. Counters file and counter update rules file are also inputs to the Promela model costruction tool. The output of the tool is the model file written in Promela specification language.

The translation and output to a Promela file is done as follows. First, we extract ADG node information from the ADG graph XML file. We use standard Java based parser to extract elements by tag name utilizing DocumentBuilderFactory, DocumentBuilder and Document classes from javax.xml.parsers library. Each node element describes an ADG node. In the declaration part of the Promela file, we declare a variable of boolean type for each node of the ADG graph, one declaration per line. The false value of the variable specifies that the node the variable represents is currently disabled, while true value of the variable specifies that the node the variable represents is currently enabled at execution time. All init nodes of the ADG graph are initialized to true (they are enabled when the system starts up). All other nodes are initialized to false. We also declare a variable for each counter from counters file per each component and initialize them to 0. We then output a blank line. Next, we output proctype declaration, call our proctype test, and output an open curly brace on the next line. The body of the proctype starts after the open brace. On the next line, we output the beginning of a Promela loop statement. Next, we output the beginning of a Promela selection statement. Statements inside a selection statement are chosen non-deterministically. Then, we extract ADG edge information from the ADG graph XML file. We use standard Java based parser to extract elements by tag name utilizing DocumentBuilderFactory, DocumentBuilder and Document classes from javax.xml.parsers library. Each edge element describes an ADG edge. Edges of the ADG graph are translated

into guarded commands. A guard is a condition that describes the node that the edge is coming out from as enabled. These guarded commands are placed inside the Promela if structure. The body of each guarded command represents the result of a transition in the ADG graph. Some nodes become enabled (these are those nodes that the edge goes into) and other nodes become disabled (these are those nodes that the edge goes out from). If a node has outcoming OR edges, then there is a guarded command per each outcoming edge. If a node has outcoming AND edges, then all the nodes that the edges go into need to be combined in the body part of one guarded command and get enabled. If a node has incoming AND edges, all the nodes that these edges are coming out from need to be included in the guard part of the guard statement, since all of them need to be enabled for the destination node to become enabled. Counter update rules are embedded in guarded commands as following. When an action in the body of a guarded command matches the if part of some counter update rule, then the corresponding action of the rule is included in the body of the guarded command as well. After all edges are translated into guarded commands and all guarded commands are included into the body of the Promela if structure, we output the end of the Promela selection structure on the next line. Next, we output the end of the Promela repetition structure on the next line. Lastly, we ouput a closing brace that signifies the end of the proctype.

Let's look at an example. ADG graph depicted in Figure 4.15 is represented by XML file in Listing 4.9. The graph has 5 nodes two of which are init nodes. This XML file is given to the translation tool as input. The output from the translation tool is the Promela model shown in Listing 4.10.

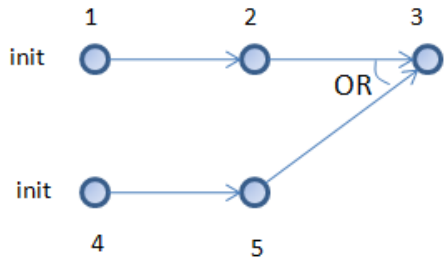

Figure 4.15: *An example of ADG graph with five nodes*

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<adgGraph>
<nodes>
<node nodeId="1" nodeName="1" nodeStatusInit="true"/>
<node nodeId="2" nodeName="2" nodeStatusInit="false"/>
<node nodeId="3" nodeName="3" nodeStatusInit="false"/>
<node nodeId="4" nodeName="4" nodeStatusInit="true"/>
<node nodeId="5" nodeName="5" nodeStatusInit="false"/>
</nodes>
<edges>
<edge edgeId="6" edgeName="1-2">
<nodeFrom nodeId="1" nodeName="1" statusOutcoming="AND"/>
<nodeTo nodeId="2" nodeName="2" statusIncoming="OR"/>
<edge/>
<edge edgeId="7" edgeName="2-3">
<nodeFrom nodeId="2" nodeName="2" statusOutcoming="AND"/>
<nodeTo nodeId="3" nodeName="3" statusIncoming="OR"/>
<edge/>
<edge edgeId="8" edgeName="4-5">
<nodeFrom nodeId="4" nodeName="4" statusOutcoming="AND"/>
<nodeTo nodeId="5" nodeName="5" statusIncoming="OR"/>
<edge/>
<edge edgeId="9" edgeName="5-3">
<nodeFrom nodeId="5" nodeName="5" statusOutcoming="AND"/>
<nodeTo nodeId="3" nodeName="3" statusIncoming="OR"/>
<edge/>
</edges>
</adg>
```

Listing 4.9: *ADG XML input file.*

```
1  bool e1 = true;
2  bool e2 = false;
3  bool e3 = false;
4  bool e4 = true;
5  bool e5 = false;
6
7  active proctype test()
8  {
9    end:do
10     :: if
11        :: (e1 == true) -> atomic {e1 = false; e2 = true}
12        :: (e2 == true) -> atomic {e2 = false; e3 = true}
13        :: (e3 == true) -> atomic {e3 = false}
14        :: (e4 == true) -> atomic {e4 = false; e5 = true}
15        :: (e5 == true) -> atomic {e5 = false; e3 = true}
16     fi
17   od
18 }
```

Listing 4.10: *Promela file generated for ADG graph in Figure 4.15.*

For example, boolean variable e1 is declared in line 1 and represents node 1. It is initialized to true since node 1 is an init node. The guard statement in line 11 represents the edge from node 1 to node 2. When node 1 is enabled (the guard), the system can go to a different state where e1 becomes disabled and e2 becomes enabled.

4.5.3 Optimizer

The optimization engine derives information necessary for application and physical topology based optimizations. Here, we describe our optimizer in detail.

The following are the inputs to the optimization engine:

a) An application *App* specified as a Promela model.

b) The membership criteria for the interaction sets used in the algorithms *alg* that application *App* requires.

c) Information on dynamic updates to the interaction sets used in the algorithms *alg* that application *App* requires.

d) Network model specified as a *physical topology graph (PTG)* that describes the underlying physical topology on top of which the application is being deployed.

e) A mapping *Map* identifying the location of each component in the physical topology.

The optimizer outputs information for application-based static optimization, application-based dynamic optimization and physical topology-based optimization in XML format.

Next we describe the inputs to the optimizer in detail.

Application *App*

An application *App* is specified as a Promela model. This model is an abstraction of the application *App*. The Promela model is produced by the Promela model construction tool. An example of such a file is shown in Listing 4.11.

```
bool e1 = true;
bool e2 = false;
bool e3 = false;
bool e4 = true;
bool e5 = false;

active proctype test()
{
  end:do
    :: if
       :: (e1 == true) -> atomic {e1 = false; e2 = true}
       :: (e2 == true) -> atomic {e2 = false; e3 = true}
       :: (e3 == true) -> atomic {e3 = false}
       :: (e4 == true) -> atomic {e4 = false; e5 = true}
       :: (e5 == true) -> atomic {e5 = false; e3 = true}
    fi
  od
}
```

Listing 4.11: *Example of Promela file generated for ADG graph in Figure 4.15.*

Membership criteria for the interaction sets

Another input to the optimization engine is the membership criteria for the interaction sets used in the algorithms *alg* that application *App* requires. The membership criteria for interaction sets need to be supplied to optimizer in a form that the optimizer is able to parse it. An example of such an input file for membership criterion of SRT interaction set is shown in Listing 4.12. It specifies the name of the interaction set, the name of the query to run and the arguments for the query in a comma delimited format.

```
SRT.i, ALL, not exclusive, i.in_cs = 1, j.in_cs = 1
...
```

Listing 4.12: *Sample of membership criteria input to optimizer.*

Information on dynamic updates to the interaction sets

Information on dynamic updates to the interaction sets used in the algorithms *alg* that application *App* requires is also supplied to optimizer as an input. This information is specified in a standardized form. An example of such an input file for information on dynamic updates to the SRT interaction set is shown in Listing 4.13.

```
SRT.i, ALL, not exclusive, i.in_cs = 1, j.in_cs = 1, if k.state = requesting then k.cs_request = enabled
...
```

Listing 4.13: *Sample of dynamic information input to optimizer.*

This example specifies the name of the set for which dynamic optimization information is needed, the name of the query to run and the arguments for the query. It also specifies that if a process is in the requesting state, a cs_request node for this process will be enabled in the abstraction model for our system. Optimizer will translate this information into which node is enabled and will run the specified query with an additional argument representing which node is enabled.

Network model

The network model is specified as a *physical topology graph (PTG)* that describes the underlying physical topology on top of which the application is being deployed. An example of such a file is shown in Listing 4.14.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<Topology>
<Nodes>
<Node id="1" nodeName="0"/>
<Node id="2" nodeName="1"/>
<Node id="3" nodeName="2"/>
<Node id="4" nodeName="3"/>
<Node id="5" nodeName="4"/>
<Node id="6" nodeName="5"/>
...
</Nodes>
<Links>
<Link id="13" node1="0" node2="1"/>
<Link id="14" node1="1" node2="2"/>
<Link id="15" node1="2" node2="3"/>
<Link id="16" node1="3" node2="0"/>
...
</Links>
</Topology>
```

Listing 4.14: *Sample of PTG XML file provided to optimizer as an input.*

Component to physical topology mapping

A mapping *Map* identifying the location of each component in the physical topology is also provided to the optimizer as input by the application developer. An example of such a file is shown in Listing 4.15.

```
C1, 1
C2, 2
...
```

Listing 4.15: *Sample of a mapping file provided to optimizer as an input.*

This example specifies that component C1 is mapped to processor 1 and component C2 is mapped to processor 2.

The optimization engine takes Promela model as an input. Next, the optimizer processes membership criteria information file. The optimizer parses each membership criterion at a time. Each membership criterion has a name of the interaction set that it defines, the name of the query to run, and arguments to the query. These parts are parsed and the name of the interaction set is temporarily stored in memory. The arguments to the query are either ADG nodes that represent events or counters. If the arguments to the query are counters, then one of the arguments is constant and corresponds to the same processor that the interaction set is for. The other argument is variable and corresponds to a counter for another processor and the query needs to be run for every possible processor in the variable part. Therefore, we run the query number of processors minus one times to populate one interaction set. If the arguments to the query are ADG nodes, then again each query contains a part that stays constant (it corresponds to a node of ADG that is related to a process associated with the interaction set) and a variable part (that will be associated with another node of ADG). The variable part could be any node of the ADG so the query has to be run against every possible node. Each query is translated into a never claim supported by Spin according to a template that is specific to the query. To run a query, we run Promela model with the never claim that represents the query to run. For each run, query arguments are updated appropriately in the never claim.

For example, query exclusive(e2,e5) will be translated to the never claim shown in Listing 4.18.

```
1 never {
2   do
3     :: ((e2 == true) && (e5 == true)) -> break
4     :: else
5   od
6 }
```

Listing 4.16: *Never claim example.*

The never claim is verified for every possible state of the system. If we run the verification on the model shown in Listing 4.10 with never claim in Listing 4.18, Spin would output

"claim violated" response. What it means is that there is a state of the system represented by the model when both e2 and e5 are true (or enabled). Therefore, exclusive(e2,e5) query would return false.

If we look at another example, say exclusive(e1,e2), the corresponding never claim will look like in Listing 4.17.

```
1 never {
2   do
3     :: ((e1 == true) && (e2 == true)) -> break
4     :: else
5   od
6 }
```

Listing 4.17: *Never claim example.*

If we run the verification on the model shown in Listing 4.10 with never claim in Listing 4.17, Spin verification will not detect a violation. What it means is that there is no state of the system represented by the model when both e1 and e2 are true (or enabled). Therefore, exclusive(e1,e2) query would return true.

We developed a script that runs Promela model with each never claim at a time. Appropriate arguments are supplied for each run. The result of each run is used to populate interaction sets specified in membership criterion. Physical topology graph along with the mapping *Map* are utilized here as well and are used to see if several components are mapped to one processor.

Next, the optimizer processes the information on dynamic updates to the interaction sets. The procedure is similar to the one described for membership criteria. Optimizer translates the specified conditions into wich nodes in the model are enabled and the query is run taking that information into consideration. The result of each run is translated into optimization information in the form of rules.

We completed implementation of optimizer parser to process membership criteria and dynamic updates information for mutual exclusion service. We plan to finish implementation of the parser for other services in the future work.

Finally, the optimizer caclulates the shortest path information for each pair of nodes in the physical topology graph for physical topology based optimization and outputs that

information in XML format.

4.5.3.1 Discussion on optimizer complexity

In this section we will discuss the computational complexity of our tools. We will concentrate on the optimizer because the optimizer is doing the bulk of the work. Other tools process information by performing necessary conversions.

The main job of the optimizer is to provide optimization information. The optimization information is in the form of sets of processes that participate in a certain interaction. These sets are used to specify what is in our algorithms' interaction sets described by membership criterion per interaction set. Each algorithm is written in terms of interaction sets. Since algorithms (services) are deployed on processors, we can talk of membership sets related to a process. Each membership criterion is written in terms of basic queries that need to be executed on ADG. Each basic query contains a part that stays constant (it corresponds to a node of ADG that is related to a process associated with the interaction set) and a variable part (that will be associated with another node of ADG). The variable part could be any node of the ADG so the query has to be run against every possible node. Therefore, the query needs to be run n-1 times in the worst case, with n being the number of nodes in the ADG graph. Total complexity of running the query will then be (n-1) x (complexity of running one query instance).

We utilize Spin model checker to answer the queries. First, ADG is translated into Promela model. Next, a query is translated into a never claim supported by Spin. Then, a model is verified with the never claim provided. Each Spin component (Promela model or never claim) is a finite state automaton A. We will refer to state set S of A as A.S. The computational complexity for the depth-first search algorithm utilized by Spin is linear in the number of reachable states in A.S. But since A.S is computed from two asynchronous components, Promela model and never claim, the size of this state set is equal to the size of the Cartesian product of these two component state sets. Since the never claim is always represented by a constant number of instructions (see Listing 4.18), the computational complexity is still linear in the number of reachable states R in promela model. Thus, total

computational complexity of running one query is in the order of n x R.

```
never {
  do
    :: ((e2 == true) && (e5 == true)) -> break
    :: else
  od
}
```

Listing 4.18: *Never claim example.*

So, the complexity of populating an interaction set for one component is in the order of $n \times R$. In the worst case, we might have one component per prosessor and one ADG node per component. So, to populate an interaction set for all components will require $n \times n \times R$ in the worst case.

We might have m different interaction sets. To populate all of them then will require $m \times n \times n \times R$ in the worst case. This is the total complexity of our optimizer.

In practice, we can lower the computational complexity tremendously. First, we note that a basic query might contain only a certain type of nodes or update rules for counters might depend on only certain types of nodes. For example, SRT set is defined as following:

$SRT_i = \{j : C_j \in S \wedge \neg exclusive(C_i.in\_cs = 1, C_j.in\_cs = 1)\}$.

Therefore, to answer exclusive query with conditions on in_cs counter, we can project ADG graph to another graph that will have only cs_request and cs_release type nodes. For example, if we need to run the above mentioned query on graph shown in Figure 4.16, we can first project the graph to the one shown in Figure 4.17. As the result, the number of nodes in the graph is much lower than in the original graph. When the graph is translated into Promela model, the number of reachable states is tremendously lower than in the original model. The total complexity then will be in the order of $m \times n_{projected} \times n_{projected} \times R$ with $n_{projected}$ significantly smaller than original n.

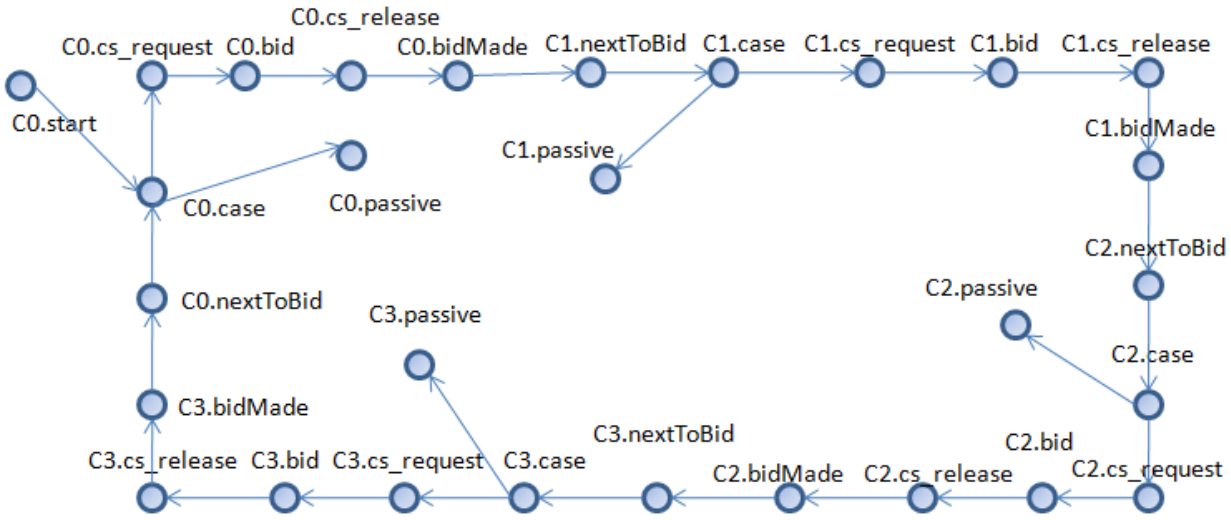

Figure 4.16: *An example of ADG graph before projection*

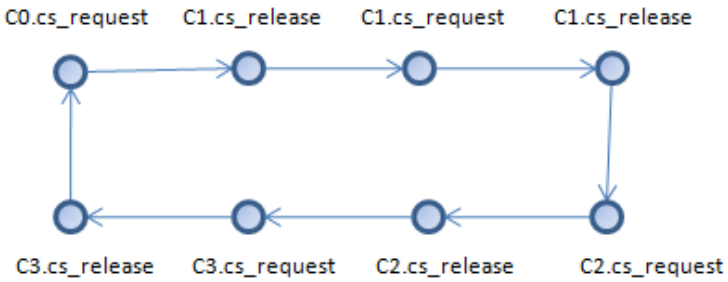

Figure 4.17: *An example of ADG graph after projection*

We can do even better. Suppose, nodes C1.cs_request and C2.cs_request correspond to the components that are mapped onto the same processor. Then, the two cs_request nodes could be combined into one and two cs_release nodes could be combined into one as well as in Figure 4.18.

A few other observations. Some interaction sets could be defined the same. Then, there is no need to run same set of queries on ADG multiple times. The information can just be copied over. This results in a smaller m in our complexity formula.

We utilize only boolean variables in our Promela model. There are no channels and message buffers are not needed. The two most important sources of complexity in Spin models, the number of asynchronously executing processes and the size of message buffers,

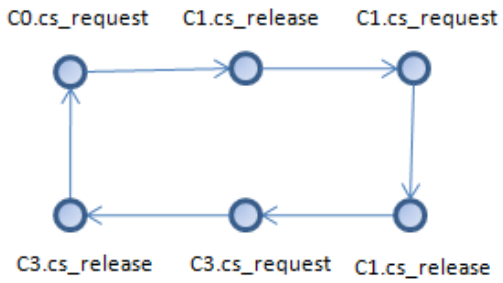

Figure 4.18: *An example of a projected ADG graph*

are not an issue in our approach. That results in high scalability of our approach, which is the key to the success of our customization technique.

The main advantages of our approach is that analysis is performed off-line and at the modeling level where the state space is much smaller.

4.6 Summary

Here, we will summarize the steps that application and algorithms developers need to take in their developments to utilize InDiGO infrastructure.

In our framework, the application developer specifies applications in Cadena, an integrated modeling and development environment for component-based systems. The developer designs components first and then develops an application by identifying the component instances and specifying their port interconnections (assembly specification).

For each component instance, Cadena generates a skeleton Java file. The application designer fills in the business logic in the Java file and annotates the component code to specify the required library algorithms. The designer is provided with a list of available services and annotations per service interface. As algorithm developers provide new services, the list of available services along with their interfaces is updated and provided to the application developers.

A component property specification (CPS) file is also generated by Cadena for each component and contains information relevant to the internal structure of the component.

To derive application specific information from application specification, we construct an *application dependency graph* (ADG) representing the structure of the application. This process is automated - we use ADG construction tool.

The optimization engine uses the ADG to derive information necessary to optimize the algorithms by running queries on the ADG. We have identified a set of basic queries useful in a number of algorithms.

The application designer must also provide a mapping, Map, to the optimizer to specify the nodes where each component instances are to be deployed.

The task of a algorithm developer is to provide a library of distributed algorithms for common tasks such as mutual exclusion and termination detection. To enable customization, an algorithm developer must expose design knowledge pertaining to an algorithm in a form which can be leveraged for analysis. To achieve this, we require a designer to first identify

the interaction sets, which characterize algorithm's communication structure and specify the processes participating in each interaction. The designer then writes the algorithm in terms of these sets. Next, the designer defines the *membership criterion* for each set; that is, the criterion for a process to be involved in an interaction. This criterion is a problem-specific property a process must satisfy to participate in the interaction. The criteria must be defined in terms of the queries supported by the analysis infrastructure.

Next, the algorithm designer supplies information for *dynamic updates to the interaction sets*. This information can be utilized by our framework to further constrain interaction sets at run time and to achieve even better optimization.

The optimization engine takes as input a) an application *App* specified as an ADG graph; b) the membership criteria for the interaction sets used in the algorithms *alg* that application *App* requires; c) information on dynamic updates to the interaction sets used in the algorithms *alg* that application *App* requires; d) network model specified as a *physical topology graph (PTG)* that describes the underlying physical topology on top of which the application is being deployed; and e) a mapping *Map* identifying the location of each component in the physical topology and proceeds with translating the ADG graph into Promela model. Next, the optimizer processes membership criteria information to produce application based static optimization information. Next, the optimizer processes the information on dynamic updates to the interaction sets. This results in application based dynamic optimization information in the form of rules. Finally, the optimizer caclulates the shortest path information for each pair of nodes in the physical topology graph for physical topology based optimization and outputs that information in XML format.

The final step involves installing the code on each node in the network. The application designer must provide a mapping, Map, to specify the nodes where each component instances are to be deployed. Cadena provides deployment tools that use this mapping to generate and install the code to be deployed on each node.

Chapter 5

Optimizations

This chapter describes optimizations that are possible through the utilization of our framework. We perform application-based static optimizations, application-based dynamic optimizations and physical topology-based optimizations. Application-based optimization comes in the form of reducing the number of control messages that an algorithm has to use to achieve its task. In our framework, customizable algorithms are designed by algorithm developers and written in terms of interaction sets. These interaction sets capture the communication structure of an algorithm for possible optimizations in a given context. Application-based optimization is achieved through constraining interaction sets based on application information. Physical topology-based optimizations are realized through eliminating redundant messages and are based on shortest paths information. Each type of optimization is describe in a separate section of this chapter. We summarize in section 5.4.

5.1 Application-based static optimizations

Static application-based optimization is performed by computing the initial values of the interaction sets. These values are known to hold throughout the execution of the application. Optimization engine provides this information in the form of XML file.

For each algorithm used by *App*, the optimization engine computes the values of the interaction sets by issuing queries on the ADG. Note that the optimization engine does not need to know how these sets are used in the algorithm. It merely uses the membership

criteria for interaction sets, provided by an algorithm developer to the optimizer, to query the ADG. So, these queries are essentially those corresponding to the membership criteria for each interaction set. Based on the responses, the optimization engine produces a file in XML format describing the set membership. An excerpt of the file for the SRT sets used in the mutual exclusion algorithm for bidding application described in evaluation section 6.1.1 on page 109 is shown in Listing 5.1. This set is computed with respect to each component, and the elements belonging to one of the bidders are shown.

```
1  <?xml version="1.0" encoding="ASCII" standalone="no"?>
2  <components>
3  <component componentName="bidder0">
4  <SRT>
5  <SRTelement SRTelementName="bidder8"/>
6  <SRTelement SRTelementName="bidder9"/>
7  <SRTelement SRTelementName="bidder6"/>
8  <SRTelement SRTelementName="bidder7"/>
9  <SRTelement SRTelementName="bidder4"/>
10 <SRTelement SRTelementName="bidder5"/>
11 <SRTelement SRTelementName="bidder11"/>
12 <SRTelement SRTelementName="bidder10"/>
13 </SRT>
14 </component>
15 ...
```

Listing 5.1: *Example of static optimization information.*

For example, if bidder0 wants to access critical section, it needs to send a request only to components listed as elements of its SRT set, namely: bidder4 through bidder11. Notice, that bidder1 through bidder3 are excluded.

5.2 Application-based dynamic optimizations

We also perform dynamic optimization of the interaction sets. During the execution of the application, it might be the case that in specific states, the set membership can be further constrained. During the execution of the algorithm, a process may be able to gain knowledge of the state of the application entities at other nodes via incoming messages. For example, when process i receives a request message from j in a mutal exclusion algorithm, it knows that an application component at j is requesting critical section entry. Again, since the algorithm designer has knowledge of the algorithm, it can provide such information by identifying the assertions that hold true about the application state at certain points in the algorithm. The optimization engine then computes a set of dynamic optimization rules to update the set membership dynamically.

The algorithm developer is required to specify information on dynamic updates to the interaction sets in a standardized form and provide it to optimizer as input. The optimization engine evaluates queries provided in the dynamic updates information and, based on the responses to the queries, generates a set of dynamic optimization rules in XML format. For example, ADG analysis may reveal that when C_j is requesting entry into critical section, C_k cannot be concurrently requesting entry. Hence if i has already received a request from j, then SRT_i is updated to exclude k.

An excerpt of the file for the SRT sets used in the mutual exclusion algorithm for bidding application described in evaluation section 6.1.1 on page 109 is shown in Listing 5.2. As shown, for each condition, it specifies the process ids that can be removed from the SRT interaction set. The call to procedure *update_SRT* is added to the algorithm (line 07 in Figure 4.9). This procedure updates the SRT interaction set at run time.

```
1 <?xml version="1.0" encoding="ASCII" standalone="no"?>
2 <dynamicRules>
3 <RULE componentName="bidder0">
4 <currentlyRequesting componentName="bidder11"/>
5 <doesNotBelongToSRT componentName="bidder9"/>
6 <doesNotBelongToSRT componentName="bidder10"/>
7 </RULE>
8 ...
```

Listing 5.2: *Example of dynamic optimization information.*

From this listing, for example, for component bidder0, if it wants to access critical section, and it knows that bidder11 is currently requesting access to critical section too, bidder0 does not need to send a request message to bidder9 and bidder10 and excludes them from its SRT set when update_ SRT() is called.

5.3 Physical topology-based optimizations

Network model is represented by the *physical topology graph PTG* that describes the underlying physical topology on top of which the application is being deployed. PTG is an undirected graph in which each node represents a physical processor. An edge is present between two nodes if and only if there is a physical connection between the corresponding physical processors. We will call edges links.

The Core Service Layer (CSL) in J-Sim simulator that we use to evaluate our framework provides the basic communication services[20] and requires that the routing tables be initialized so that messages can be routed properly. Since many algorithms involve sending the same message to multiple destinations in an interaction set, we have extended CSL to perform multi-destination routing. Given a message and a set of destinations, we compute the set of links on which to forward the message so that duplicate messages are eliminated. For example, if a message has to be sent from i to both j and k, and j is on the path from i to k, then a single copy of the message with both j and k as destinations is first sent to j. This extension requires CSL to be initialized with shortest path information.

The optimizer caclulates the shortest path information for each pair of nodes in the physical topology graph for physical topology based optimization and outputs that information in XML format. An example of such information is shown in Listing 5.3. CSL layer in J-Sim is initialized with this shortest path information and then uses it to perform multi-destination routing.

```
1  <?xml version="1.0" encoding="ASCII" standalone="no"?>
2  <TopologyStatic maxPath="5">
3  <Nodes>
4  <Node nodeName="0">
5  <Path toNode="1" pathLenght="1">
6  <PathElement hopCount="1" nodeName="1"/>
7  </Path>
8  <Path toNode="2" pathLenght="2">
9  <PathElement hopCount="1" nodeName="1"/>
10 <PathElement hopCount="2" nodeName="2"/>
11 </Path>
12 <Path toNode="3" pathLenght="1">
13 <PathElement hopCount="1" nodeName="3"/>
14 </Path>
15 <Path toNode="4" pathLenght="2">
16 <PathElement hopCount="1" nodeName="1"/>
17 <PathElement hopCount="2" nodeName="4"/>
18 </Path>
19 ...
```

Listing 5.3: *Example of physical topology shortest path information.*

5.4 Discussion

We have provided an infrastructure which consists of a tool-chain to perform algorithm optimizations. Although we have focused on specific types of optimizations in this thesis, the infrastructure is extensible to allow a richer set of optimizations. For example, the algorithm designer can enable optimizations by exposing more information about the algorithms. Any algorithm information defined in terms of queries on the ADG can be leveraged by the optimization engines for possible customization. The assertion set, $alg.app\_assert$, is an example of one such type of information which we have exploited to perform additional optimizations. Similarly, one can develop tools to capture more information about the application in the ADG, which can reveal more optimization opportunities. Indeed, one of the goals in Cadena is to derive more accurate models from the component code (e.g., derivation of more detailed CPS files for Java files). Finally, more sophisticated analysis algorithms can also be plugged into the tool-chain to analyze the ADG for aggressive optimizations. As an example, we decided to utilize model checking the ADG that can be used to verify more general properties.

We will end this chapter with the discussion on optimality of optimization. In computing, optimization is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. For instance, in networking environment it is desirable to reduce the number of messages flowing in a network. If we are able to do that, we say that we are able to optimize the system with respect to number of messages.

Optimization might mean different things to different disciplines. For example, in operations research, optimization is the problem of determining the input to a function that minimizes or maximizes its value. In computer programming, optimization usually means producing more efficient software.

Although we would usually talk about optimality when it comes to optimization, it is rare for the process of optimization to produce a truly optimal system. For example, optimizing compilers are not optimal. There is no way that a compiler can guarantee that,

for all program source code, the fastest or smallest possible equivalent compiled program is output; such a compiler is fundamentally impossible because it would solve the halting problem.

Optimization can be automated by tools or performed by developers. Optimizing a whole system is usually done by humans because the system is too complex for an automated optimizer. Developers explicitly change code so that the system performs better. Although it can produce better efficiency, it is far more expensive than automated optimizations.

Currently, automated optimizations are almost exclusively limited to compiler optimization. We go beyond that in this thesis and provide a framework to analyze the system for optimization opportunities at design level.

Chapter 6

Evaluation

In this chapter, we evaluate our infrastructure with respect to optimization in terms of the number of messages that algorithms use to perform their tasks. We want to see if our framework can utilize application ordering information for possible optimizations. We also want to see if our framework can recognize when more constraints on ordering are present in an application and translate that into higher optimization level. We implemented several distributed applications to perform experiments that would help us evaluate the optimizations performed. All applications were implemented on J-Sim simulator.[20] J-Sim is a component-based, compositional simulation environment.

In the following section, we describe our experiments for a class of bidding applications. Bids are required to be made in a mutually exclusive manner, and all bids must be delivered in a total order to all components. We also have to determine when the bidding has finished. Each application in this class requires mutual exclusion, termination detection and total ordering algorithms. We implemented each of these algorithms in J-Sim. We describe two applications used as case studies, each with different application-level constraints.

Next, we look at a class of teleteaching applications. In a question/answer session of a teleteaching application, students ask questions and instructors respond to them. Both questions and answers are required to be made in a mutually exclusive manner and be delivered in a total order to all components. We also have to determine when the session has finished. Each application in this class requires mutual exclusion, termination detection

and total ordering algorithms. For a teleteaching application, we analize the application structure and derive application and physical topology based optimization information in the form of XML files. We do not implement this application but show that interaction sets are constrained both statically and dynamically which will result in application based optimization. We also analyze physical topology for shortest path information and show that the information will result in physical topology based optimization. Whereas class of bidding applications might include a specific ordering within clusters of bidders, class of teleteaching applications in addition to that might exhibit specific ordering between the clusters.

In the last section of this chapter we evaluate the effectiveness of the customization techniques by comparing the performance of the customized algorithms to those designed for specific operational contexts. We designed optimized algorithms for one of the bidding applications described earlier. We will compare the performance of these optimized algorithms to that of our customized algorithms. We end this chapter with the discussion on the effectiveness of our customization techniques.

6.1 Bidding applications

In this section, we look at a class of bidding applications. In bidding applications, bids are required to be made in a mutually exclusive manner, and all bids must be delivered in a total order to all components. We also have to determine when the bidding has finished. Each application in this class requires mutual exclusion, termination detection and total ordering algorithms. We implemented each of these algorithms in J-Sim. We describe two bidding applications, each with different application level constraints.

6.1.1 Bidding application 1

Description

This application involves twelve players making bids. Each player is located on a separate physical machine and the machines are connected as shown in Figure 6.1.

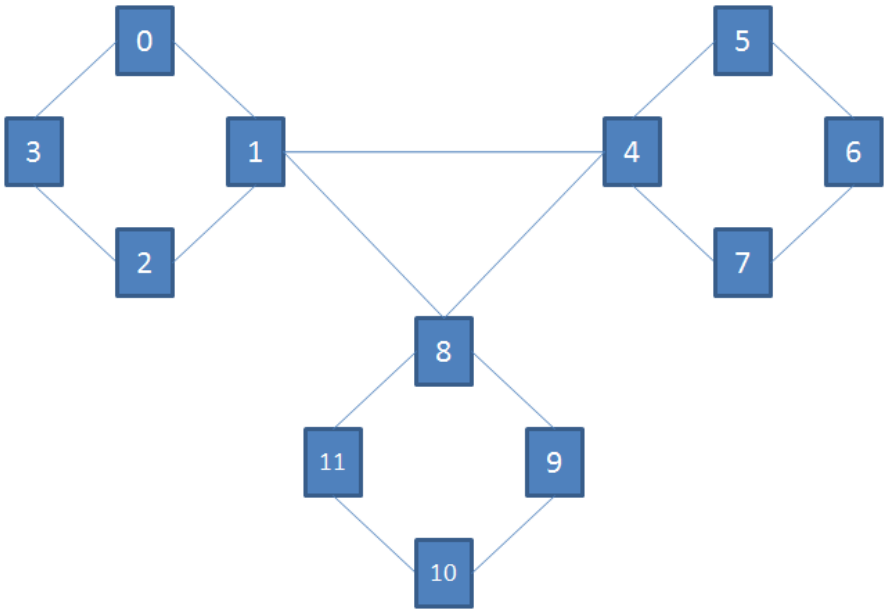

Figure 6.1: *Application 1 physical topology*

The bids that each player receives from other players need to be totally ordered, and only one player can bid at a time. Players are logically organized into three groups as shown in Figure 6.2. Players in each group make bids in a round-robin fashion (e.g., in group 0, players bid in the order 0,1,2,3,0,1,2,3....). This order is enforced by the application itself. Each player's bid is based on their current group bidding probability, which decreases with each bid made. Once a player in a group decides not to bid, no other player in the group can make any more bids. We need to know when bidding stops.

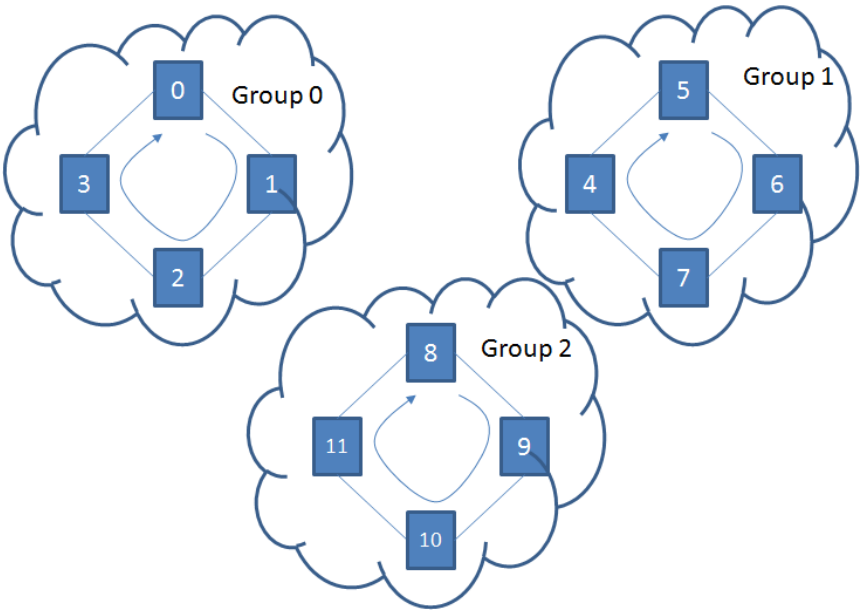

Figure 6.2: *Application 1 logical topology*

Capturing application information using Cadena

To specify this application in Cadena, we first defined two component types, bidCompInit and bidComp. bidComp component type is shown in Figure 6.3.

Figure 6.3: *Application 1 bidComp component type*

Next, we created twelve component instances and specified the port connections via the graphical interface of Cadena. The graphical representation of the scenario is shown in Figure 6.4.

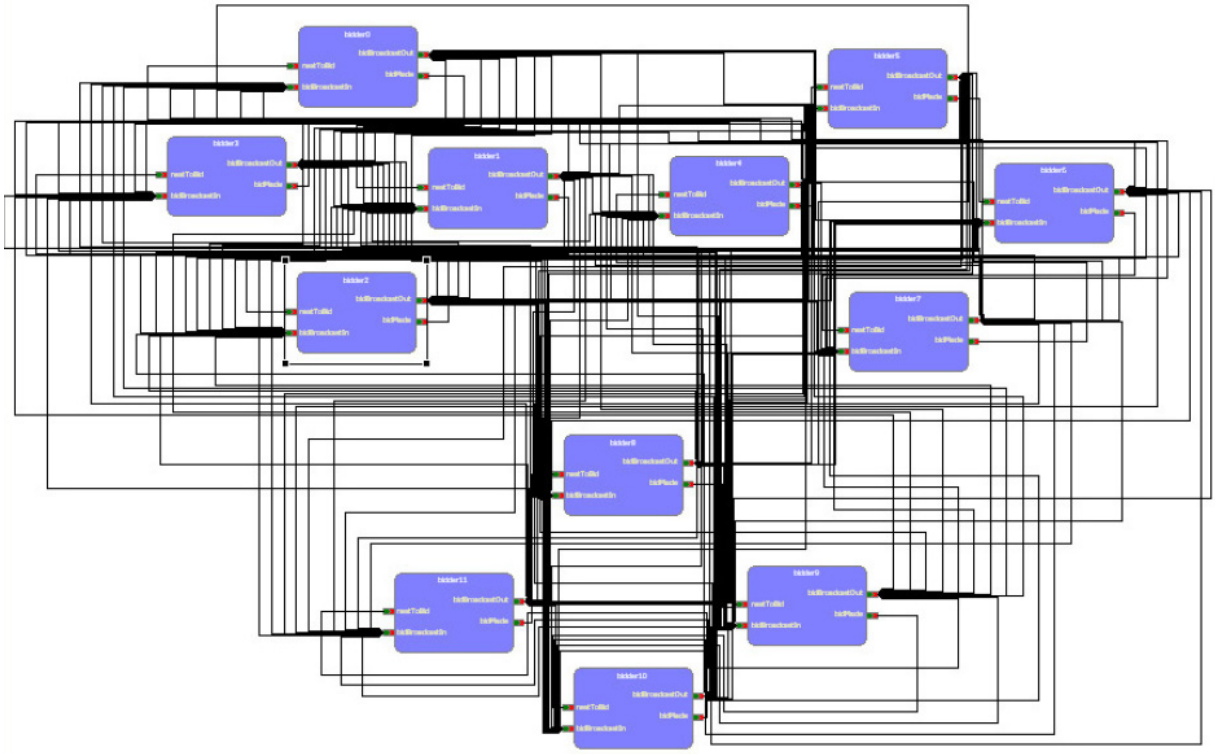

Figure 6.4: *Graphical representation of application 1 scenario*

Component specification file generated by Cadena is shown in Listing 6.1 and assembly specification file is shown in Listing 6.2.

```
1  ?xml version="1.0" encoding="ASCII"?>
2  <edu.ksu.cis.cadena.core.specification.module:Module xmi:version="2.0" xmlns:xmi=
3    <style href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_1MdskAMFEdqT6_lD
4    <components uuid="_OyaIQBbNEdypMp3lNrZ05w" name="bidCompInit" abstract="false">
5      <componentKind href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Nu9hoA
6      <ports uuid="_8_RL4BbOEdypMp3lNrZ05w" name="bidMade" interface="_sYVrUBbNEdyp
7        <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_AjRjcANsEdqTK
8      </ports>
9      <ports uuid="_JTsK4BbPEdypMp3lNrZ05w" name="nextToBid" interface="_sYVrUBbNEd
10       <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdqTK
11     </ports>
12     <ports uuid="_L196YBbPEdypMp3lNrZ05w" name="start" interface="_uhdfgcxsxagyaw
13       <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdqTK
14     </ports>
15   </components>
16 ...
17 </edu.ksu.cis.cadena.core.specification.module:Module>
```

Listing 6.1: *Part of component specification file generated by Cadena for Application 1*

```
1  </xmi:XMI>
2  <?xml version="1.0" encoding="ASCII"?>
3  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://w
4    <edu.ksu.cis.cadena.core.specification.module:Module uuid="_6P3noIjjEdyLreIOXUe
5      <style href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_1MdskAMFEdqT6_
6      <components uuid="_L-_UwIjkEdyLreIOXUe8aA" name="C" abstract="false">
7        <componentKind href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Nu9h
8        <ports uuid="_Z8Q2AIjmEdyLreIOXUe8aA" name="port1" interface="_YJnCEIjlEdyL
9          <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdq
10       </ports>
11       <ports uuid="_b3o9kIjmEdyLreIOXUe8aA" name="port2" interface="_mW4ZEIjlEdyL
12         <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdq
13       </ports>
14       <ports uuid="_hDn6QIjmEdyLreIOXUe8aA" name="port3" interface="_mW4ZEIjlEdyL
15         <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_AjRjcANsEdq
16       </ports>
17     </components>
18     <components uuid="_360UcIjmEdyLreIOXUe8aA" name="CC" abstract="false">
19       <componentKind href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Nu9h
20       <ports uuid="_7S0yEIjmEdyLreIOXUe8aA" name="port1" interface="_mW4ZEIjlEdyL
21         <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_Ehsg0ANsEdq
22       </ports>
23       <ports uuid="_83r3EIjmEdyLreIOXUe8aA" name="port2" interface="_mW4ZEIjlEdyL
24         <spec href="http://cadena.projects.cis.ksu.edu/ccm/CCM.style#_AjRjcANsEdq
25       </ports>
26     </components>
27   </edu.ksu.cis.cadena.core.specification.module:Module>
28 </xmi:XMI>
```

Listing 6.2: *Part of assembly specification file generated by Cadena for Application 1*

Cadena also generated skeleton files per component type. Skeleton file for bidComp component is shown in Listing 6.3).

```
1  public class bidComp extends Component {
2
3     ...
4
5     public void process(Object data_, Port inPort_){
6
7        if (data_ instanceof Message){
8           if (data_ instanceof BidMadeMessage){
9
10          }
11       }
12    }
13 }
```

Listing 6.3: *J-Sim Java skeleton file for bidCompInit*

We then added appropriate algorithm specific annotations to the .java files generated by the Cadena tools. An excerpt of the annotated file with annotations to enter critical section is shown in Listing 6.4.

```
1  public class bidComp extends Component {
2
3     ...
4
5     public void process(Object data_, Port inPort_){
6
7        if (data_ instanceof Message){
8           if (data_ instanceof BidMadeMessage){
9
10             /**@cs_request
11             ...
12             /**@cs_release
13
14          }
15       }
16    }
17 }
```

Listing 6.4: *Annotated J-Sim Java skeleton file*

Next, we specified the CPS files (the CPS file for bidComp is similar to one in Figure 4.1 with additional dependency for bidBroadcastIn).

Optimizations

We first constructed the ADG using our ADG contruction tool. The optimization engine then used the query interface of the ADG to initialize the interaction sets. It also produces dynamic optimization rules along with physical platform optimization information. An excerpt of the file describing static optimization rules for the SRT sets used in the mutual exclusion algorithm is shown in Listing 6.5.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<components>
<component componentName="bidder0">
<SRT>
<SRTelement SRTelementName="bidder8"/>
<SRTelement SRTelementName="bidder9"/>
<SRTelement SRTelementName="bidder6"/>
<SRTelement SRTelementName="bidder7"/>
<SRTelement SRTelementName="bidder4"/>
<SRTelement SRTelementName="bidder5"/>
<SRTelement SRTelementName="bidder11"/>
<SRTelement SRTelementName="bidder10"/>
</SRT>
</component>
...
```

Listing 6.5: *Sample of static optimization information produced for mutual exclusion service for Application 1.*

For component bidder0, if it wants to access critical section, it needs to send a request only to components listed as elements of its SRT set, namely: bidder4 through bidder11. Notice, that bidder1 through bidder3 are excluded because the ADG analysis shows that bidder1, bidder2 and bidder3 cannot make bids concurrently with bidder0.

An excerpt from dynamic optimization rules for mutual exclusion algorithm is shown in Listing 6.6.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<dynamicRules>
<RULE componentName="bidder0">
<currentlyRequesting componentName="bidder11"/>
<doesNotBelongToSRT componentName="bidder9"/>
<doesNotBelongToSRT componentName="bidder10"/>
</RULE>
...
```

Listing 6.6: *Sample of dynamic optimization information produced for mutual exclusion service for Application 1.*

The ADG analysis, for instance, reveals that if bidder0 knows that bidder11 is currently requesting (that is, bidder0 has received a request message from bidder11), then bidder9 and

bidder10 cannot be requesting concurrently with bidder0 (this is due to the cyclic nature of requests in each group). This rule is shown in Listing 6.6. Note that such optimizations are difficult to arrive at by manual inspection. Optimization information is also derived for termination detection and total ordering services.

Information for physical topology based optimization comes in the form of shortest path information for each pair of processors in a given physical topology. An excerpt from the shortest path information file produced by optimizer for this application is shown in Listing 6.7.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<TopologyStatic maxPath="5">
<Nodes>
<Node nodeName="0">
<Path toNode="1" pathLenght="1">
<PathElement hopCount="1" nodeName="1"/>
</Path>
<Path toNode="2" pathLenght="2">
<PathElement hopCount="1" nodeName="1"/>
<PathElement hopCount="2" nodeName="2"/>
</Path>
<Path toNode="3" pathLenght="1">
<PathElement hopCount="1" nodeName="3"/>
</Path>
<Path toNode="4" pathLenght="2">
<PathElement hopCount="1" nodeName="1"/>
<PathElement hopCount="2" nodeName="4"/>
</Path>
...
```

Listing 6.7: *Sample of shortest path information produced for Application 1.*

The csl subcomponent of a J-Sim component will use this information for physical topology based optimization. For instance, if the same message needs to be sent by processor 0 to processors 1 and 2, then only one message will be sent to processor 2. Since processor 1 is on the shortest path from processor 0 to processor 2, it will receive a message from processor 0 to processor 2. A separate message from processor 0 to processor 1 is then not needed.

Simulation results

The results of a typical run of the simulation are shown in Figure 6.5.

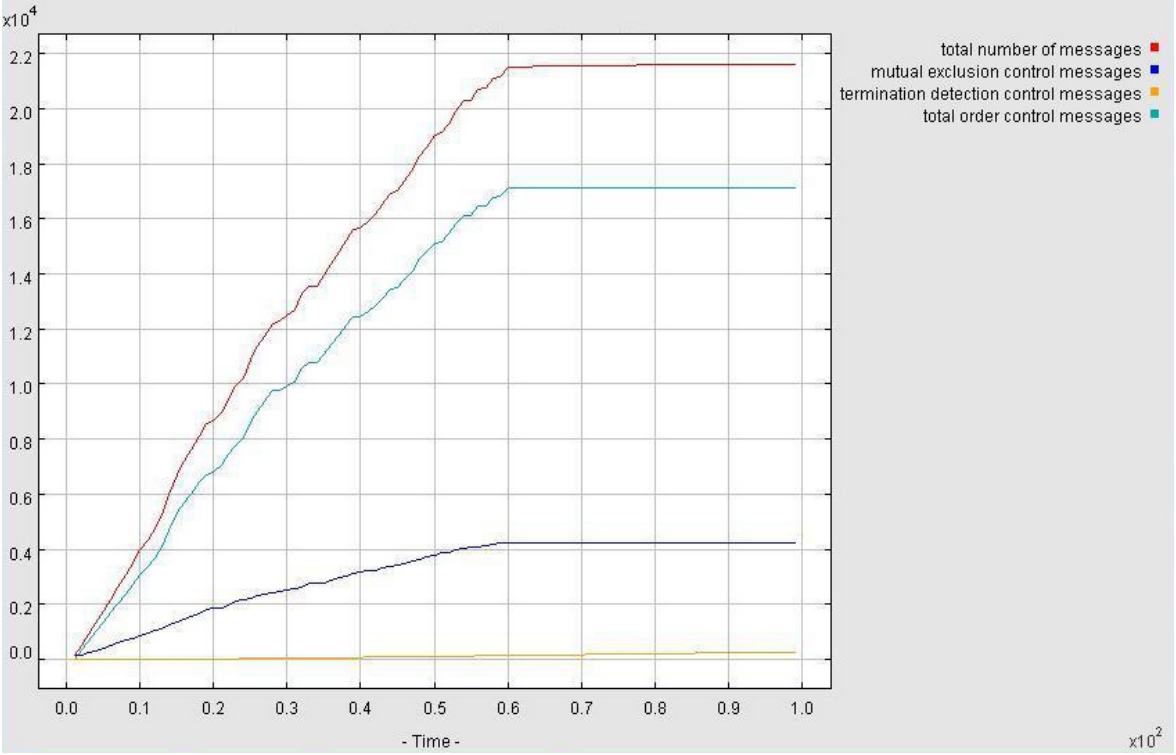

Figure 6.5: *Typical run of an application 1*

Table 6.1 and Figures 6.6 - 6.9 show the average number of messages per bid for five runs of our system. The averages are shown for mutual exclusion (ME), termination detection (TD), and total ordering (TO) algorithms as well as for total number of messages (note that the total result contains some application messages in addition to the algorithm messages). We also varied the level of optimization: No optimization (No_Opt), Static Optimization (S_Opt), Static and Dynamic Optimization (SD_Opt) and Static, Dynamic and Path Optimization (SDP_Opt).

	ME	TD	TO	Total
No_Opt	83	47	336	469
S_Opt	71	47	293	400
SD_Opt	45	17	293	355
SDP_Opt	31	17	170	219

Table 6.1: *Application 1 - average number of messages per bid*

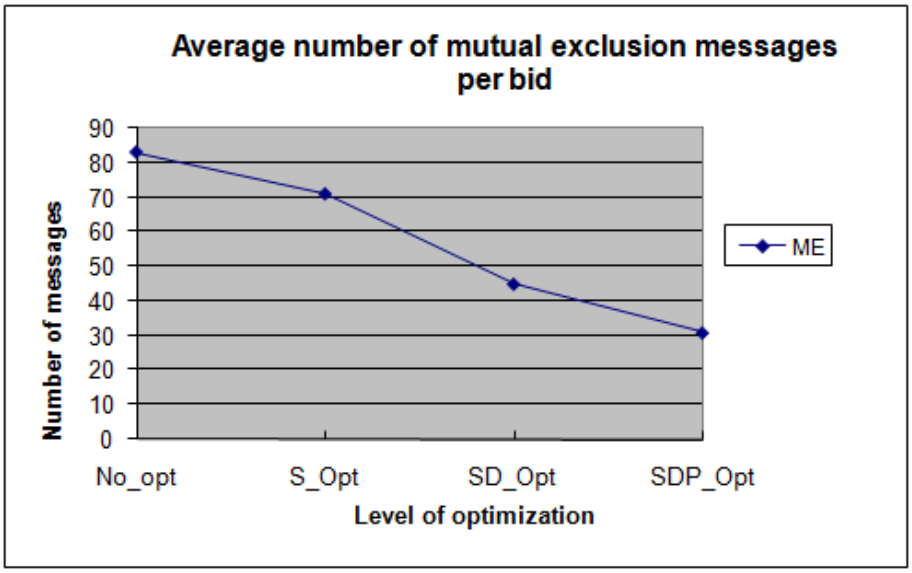

Figure 6.6: *Application 1 - average number of mutual exclusion messages per bid*

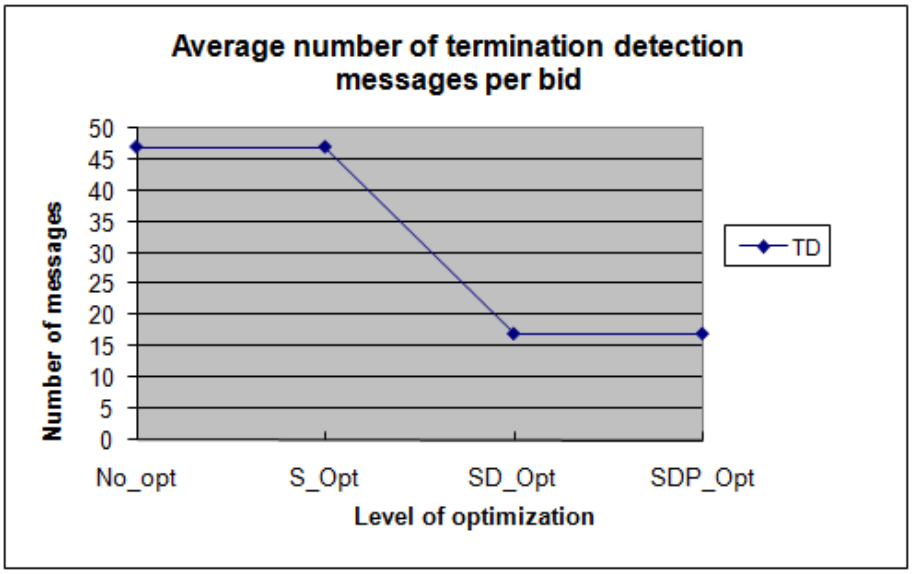

Figure 6.7: *Application 1 - average number of termination detection messages for last round*

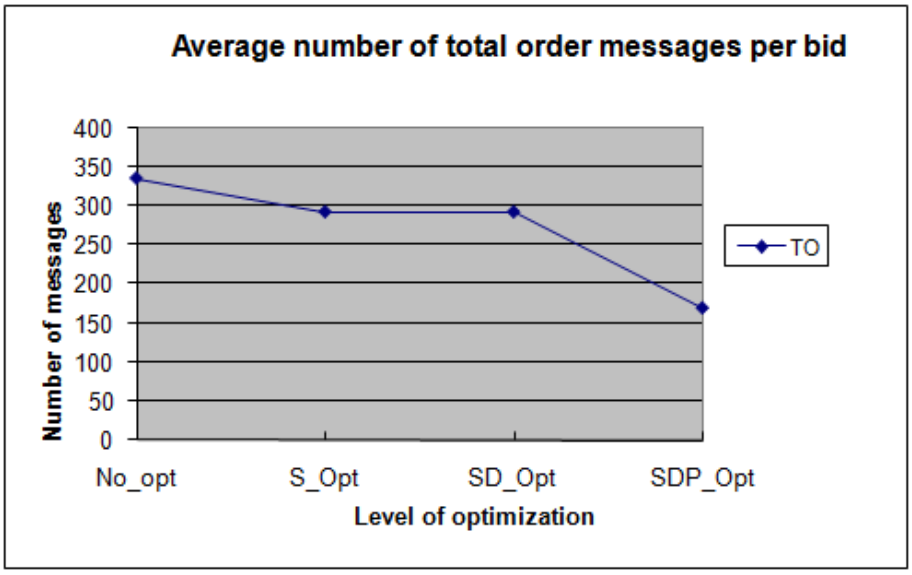

Figure 6.8: *Application 1 - average number of total order messages per bid*

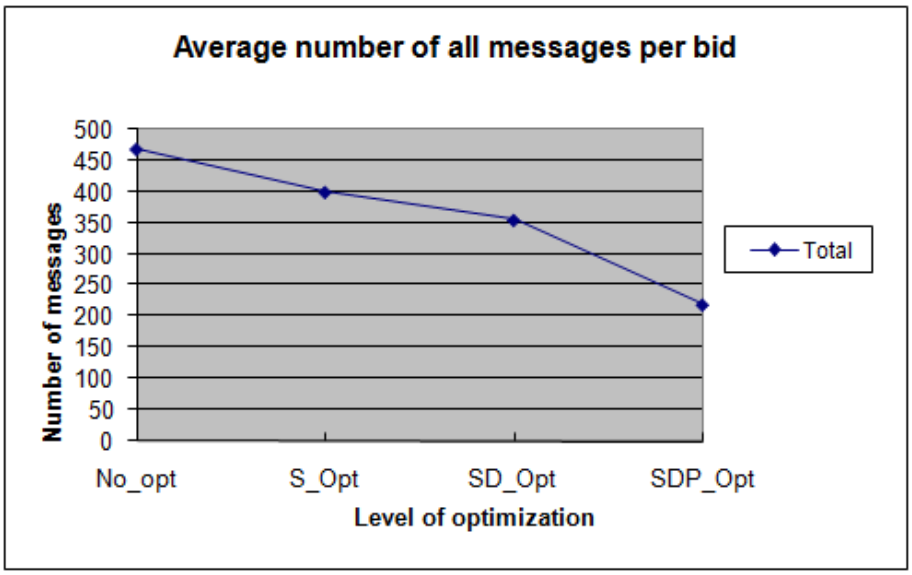

Figure 6.9: *Application 1 - average number of all messages per bid*

The results in Table 6.10 and Figures 6.11 - 6.14 show improvement in the number of messages when optimizations are performed. In Table 6.1, the row corresponding to No_Opt shows the number of messages with no optimizations (even though we have 12 components, the number of messages for ME is 83 as we are counting each hop in the physical network as a separate message). As can be seen in Table 6.10 or in Figure 6.11, for the mutual exclusion algorithm, static application optimization results in 14 percent improvement as compared to the case with no optimization. Static and dynamic application optimization results in 46 percent improvement, and addition of platform optimization results in 63 percent improvement. Similar improvement are observed for other algoritms as well. Since the termination detection algorithm may be initiated several times, the results correspond to the final initiation.

	ME	TD	TO	Total
No_Opt	0	0	0	0
S_Opt	14	0	13	15
SD_Opt	46	64	13	24
SDP_Opt	63	64	49	53

Figure 6.10: *Application 1 - % improvement in the number of messages over No_Opt case*

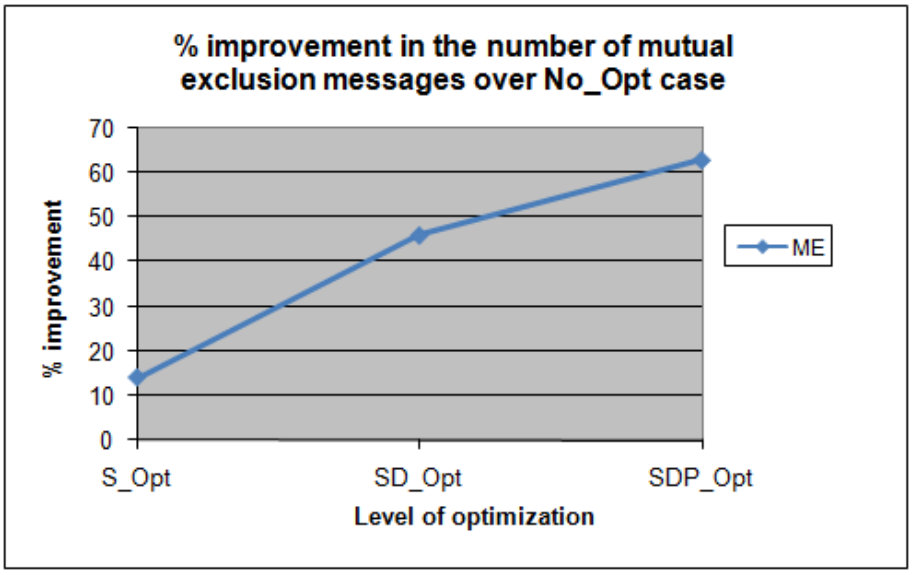

Figure 6.11: *Application 1 - % improvement in the number of mutual exclusion messages over No_Opt case*

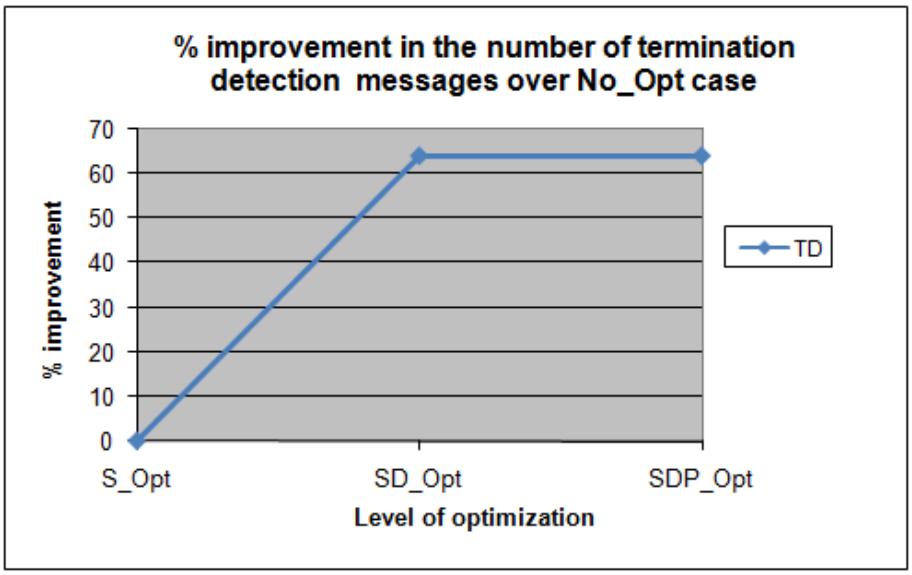

Figure 6.12: *Application 1 - % improvement in the number of termination detection messages over No_Opt case*

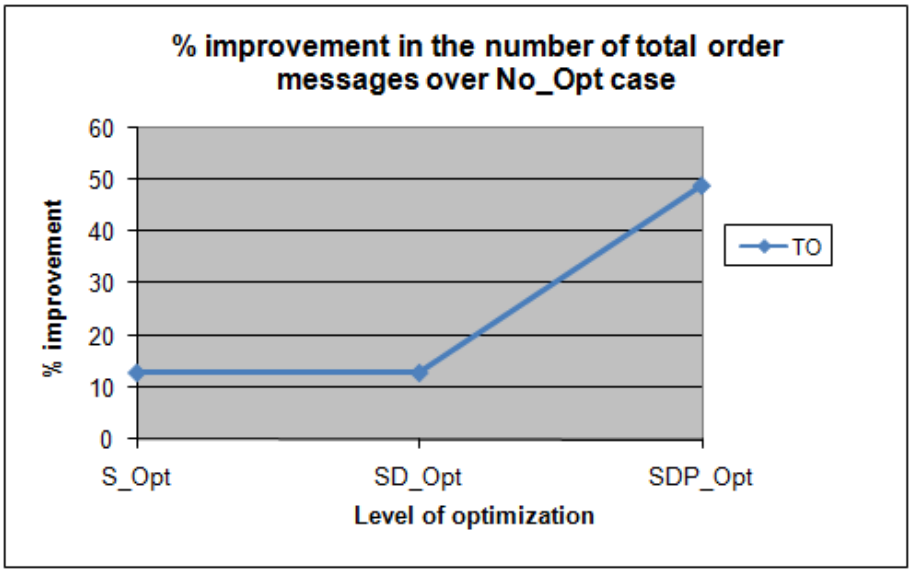

Figure 6.13: *Application 1 - % improvement in the number of total order messages over No_Opt case*

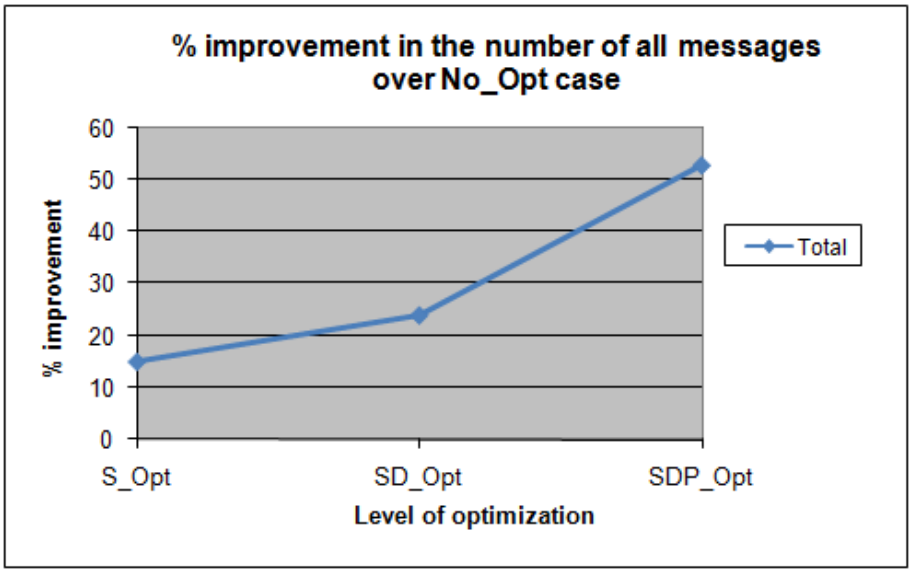

Figure 6.14: *Application 1 - % improvement in the total number of messages over No_Opt case*

6.1.2 Bidding application 2 with fewer constraints

Description

The application context of our next application is the same as in Application 1. However, Application 2 imposes the round-robin ordering of bids only in group 0 (others can request in any order) as shown in Figure 6.15. Thus, this application imposes fewer constrains on the components. Hence, one would expect fewer optimization opportunities.

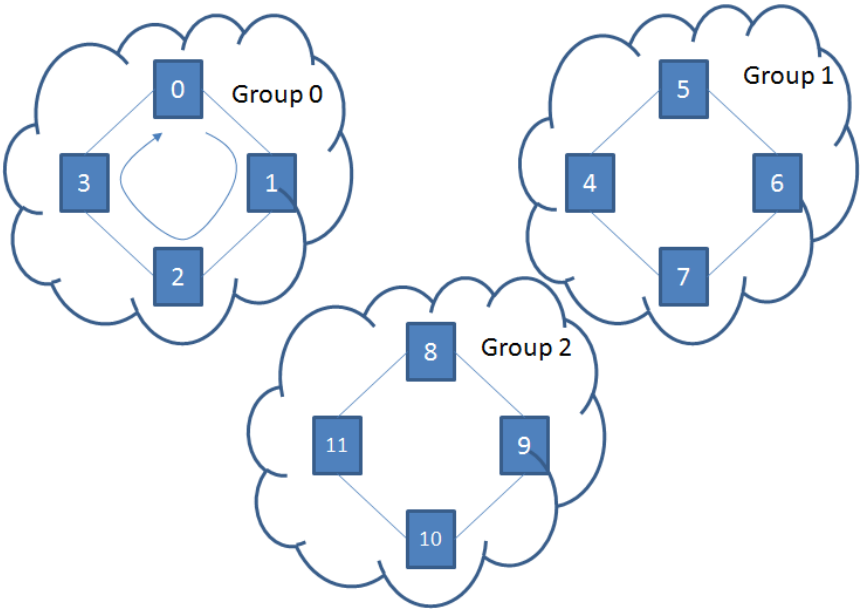

Figure 6.15: *Application 2 logical topology*

Components and the application are specified in Cadena as in application 1.

Optimizations

ADG is constructed first using our ADG contruction tool. The optimization engine then used the query interface of the ADG to initialize the interaction sets. It also produces dynamic optimization rules along with physical platform optimization information. An excerpt of the file describing static optimization rules for the SRT sets used in the mutual exclusion algorithm is shown in Listing 6.8.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<components>
<component componentName="bidder4">
<SRT>
<SRTelement SRTelementName="bidder0"/>
<SRTelement SRTelementName="bidder1"/>
<SRTelement SRTelementName="bidder2"/>
<SRTelement SRTelementName="bidder3"/>
<SRTelement SRTelementName="bidder8"/>
<SRTelement SRTelementName="bidder9"/>
<SRTelement SRTelementName="bidder6"/>
<SRTelement SRTelementName="bidder7"/>
<SRTelement SRTelementName="bidder5"/>
<SRTelement SRTelementName="bidder11"/>
<SRTelement SRTelementName="bidder10"/>
</SRT>
</component>
...
```

Listing 6.8: *Sample of static optimization information produced for mutual exclusion service for Application 2.*

For component bidder4, if it wants to access critical section, it needs to send a request to components listed as elements of its SRT set, namely: bidder0 through bidder11 excluding bidder4. Notice, that all processes are included this time, whereas in application 1 this set was constrained.

An excerpt from dynamic optimization rules for mutual exclusion algorithm is shown in Listing 6.9.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<dynamicRules>
<RULE componentName="bidder0">
<currentlyRequesting componentName="bidder11"/>
</RULE>
...
```

Listing 6.9: *Sample of dynamic optimization information produced for mutual exclusion service for Application 2.*

This time, if bidder0 knows that bidder11 is currently requesting (that is, bidder0 has received a request message from bidder11), bidder9 and bidder10 cannot be removed from bidder0's SRT set as in application 1.

Information for physical topology based optimization is the same as in application 1 and is shown in Listing 6.10.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<TopologyStatic maxPath="5">
<Nodes>
<Node nodeName="0">
<Path toNode="1" pathLenght="1">
<PathElement hopCount="1" nodeName="1"/>
</Path>
<Path toNode="2" pathLenght="2">
<PathElement hopCount="1" nodeName="1"/>
<PathElement hopCount="2" nodeName="2"/>
</Path>
<Path toNode="3" pathLenght="1">
<PathElement hopCount="1" nodeName="3"/>
</Path>
<Path toNode="4" pathLenght="2">
<PathElement hopCount="1" nodeName="1"/>
<PathElement hopCount="2" nodeName="4"/>
</Path>
...
```

Listing 6.10: *Sample of shortest path information produced for Application 2.*

Simulation results

The simulation results are shown in Table 6.2.

	ME	TD	TO	total
No_Opt	83	47	336	469
S_Opt	80	47	321	445
SD_Opt	71	36	322	432
SDP_Opt	51	36	187	275

Table 6.2: *Application 2 - average number of messages per bid*

The results in Table 6.16 show improvement in the number of messages when optimizations are performed.

	ME	TD	TO	Total
No_opt	0	0	0	0
S_Opt	4	0	4	5
SD_Opt	14	23	4	8
SDP_Opt	39	23	44	41

Figure 6.16: *Application 2 - % improvement in the number of messages over No_Opt case*

Figures 6.11 - 6.14 compare applications 1 and 2 in terms of improvement in the number of messages when optimizations are performed. As can be seen, the performance improvements in application 2 are less as compared to Application 1. For example, for mutual exclusion algorithm, the improvement for S_Opt over No_Opt is 4 percent as compared to 14 percent in Application 1. Similarly, the improvement between static and dynamic application optimization over non-optimized case is 14 percent compared to 46 percent in Application 1. These results show that, if more optimization opportunities exist, our approach will correctly recognize and use that.

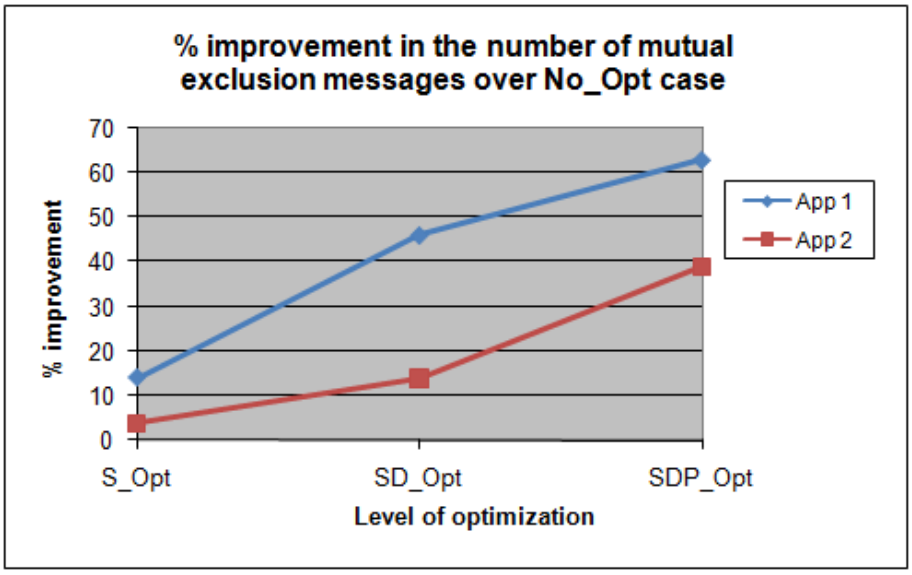

Figure 6.17: *Comparison of % improvement in the number of mutual exclusion messages over No_Opt case for Applications 1 and 2*

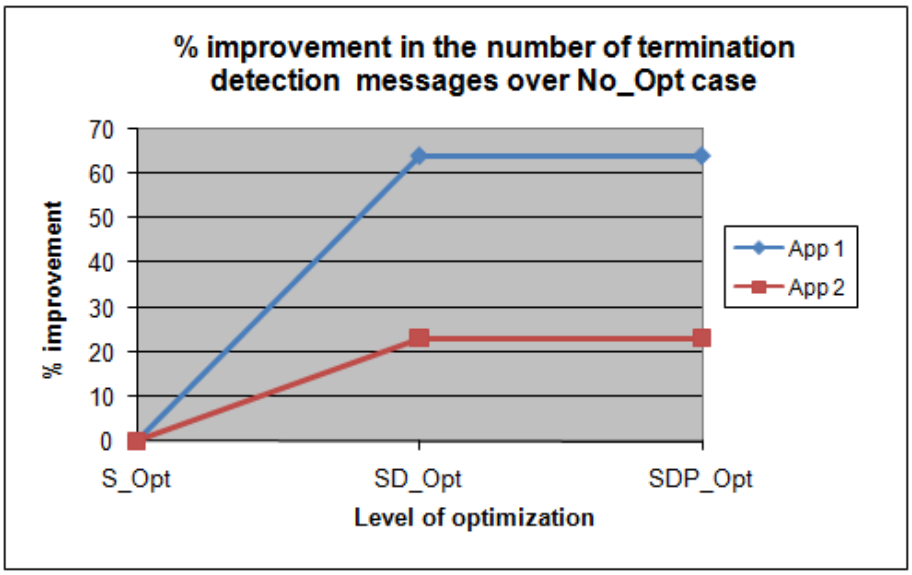

Figure 6.18: *Comparison of % improvement in the number of termination detection messages over No_Opt case for Applications 1 and 2*

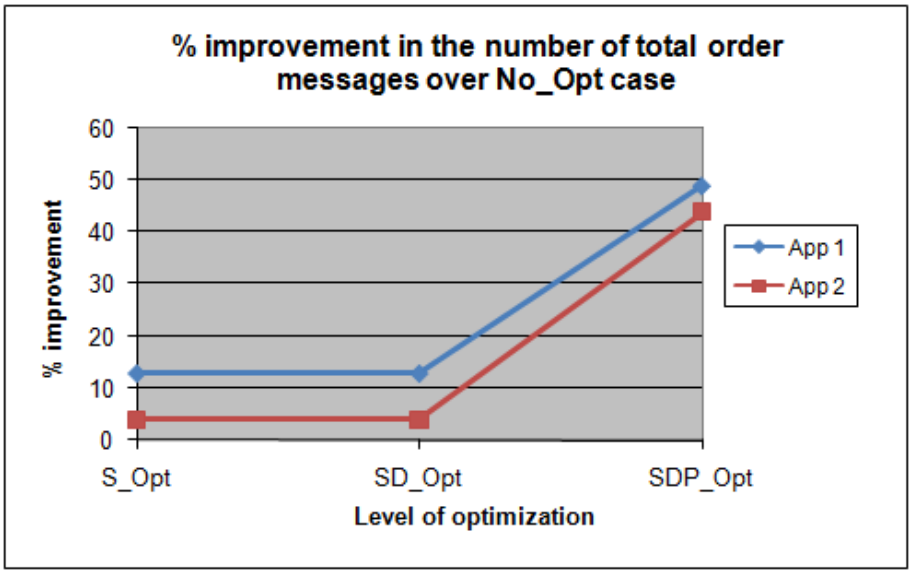

Figure 6.19: *Comparison of % improvement in the number of total order messages over No_Opt case for Applications 1 and 2*

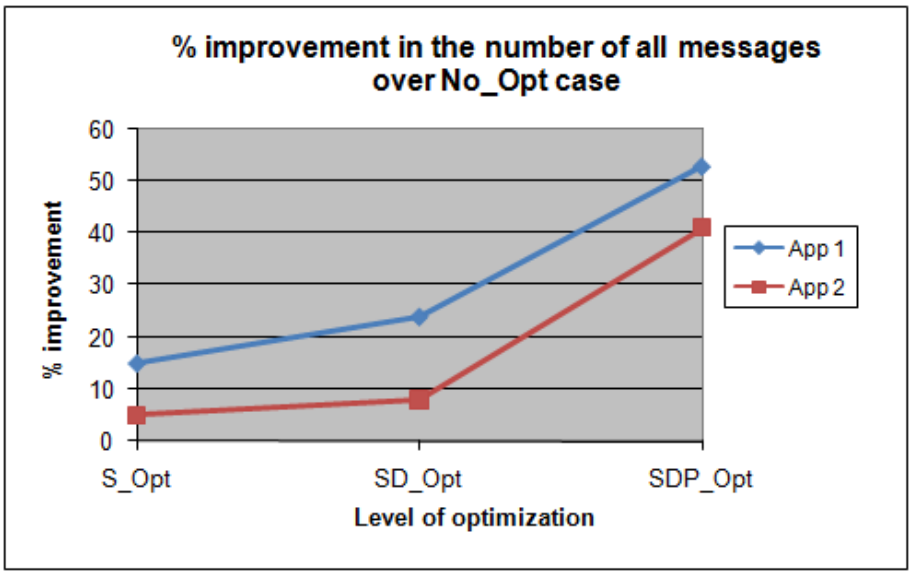

Figure 6.20: *Comparison of % improvement in the total number of messages over No_Opt case for Applications 1 and 2*

6.2 Teleteaching applications

In this section, we look at a class of teleteaching applications. In a question/answer session of a teleteaching application, students ask questions and instructors respond to them. Both questions and answers are required to be made in a mutually exclusive manner and be delivered in a total order to all components. We also have to determine when the session has finished. Each application in this class requires mutual exclusion, termination detection and total ordering algorithms. For a teleteaching application, we analize the application structure and derive application and physical topology based optimization information in the form of XML files. We do not implement this application but show that interaction sets are constrained both statically and dynamically which will result in application based optimization. We also analyze physical topology for shortest path information and show that the information will result in physical topology based optimization. Whereas class of bidding applications might include a specific ordering within clusters of bidders, class of teleteaching applications in addition to that might exhibit specific ordering between the clusters.

6.2.1 Teleteaching application 3

Description

This application involves one instructor and two students in a question/answer session. Students ask questions and instructor responds to each question with an answer. The answer is broadcast to all students. Instructor and each of the students is located on a separate physical machine and the machines are connected as shown in Figure 6.21.

Figure 6.21: *Teleteaching application 3 physical topology*

Questions/answers need to be totally ordered, and only one student or instructor can ask/answer a question at a time. Instructor and students are logically organized into two groups as shown in Figure 6.22. Students ask questions in a round-robin fashion (e.g., the order is s1,s2,s1,s2....). This order is enforced by the application itself and represents the ordering in a group.

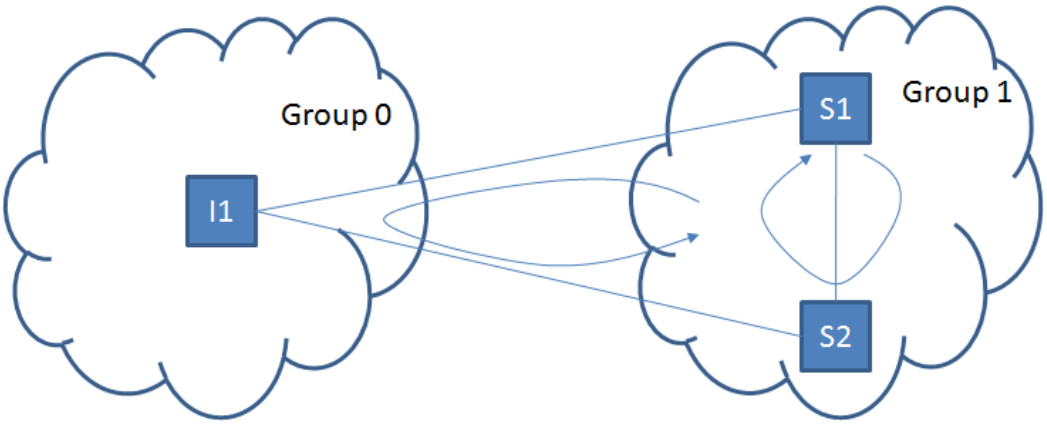

Figure 6.22: *Teleteaching application 3 logical topology*

Instructor answers a question only in response to a question received from one of the students. This order is enforced by the application and represents the ordering between the groups.

Each student's desire to ask a question is based on their current group probability to ask a question, which decreases with each question asked. Once a student in a group decides not to ask a question, no other student in the group can ask any more questions. We need to know when the session is over.

Capturing application information using Cadena

To specify this application in Cadena, we first defined two component types, Instructor and Student. Instructor component type is shown in Figure 6.23 and Student component type is shown in Figure 6.24.

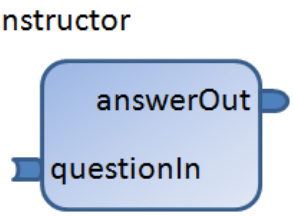

Figure 6.23: *Application 3 - Instructor component type*

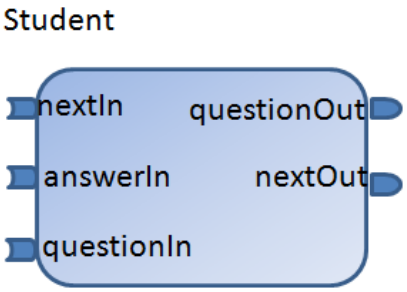

Figure 6.24: *Application 3 - Student component type*

Next, we created one component instance I1 of Instructor type and two component instances S1 and S2 of Student type. We then specified the port connections via the graphical interface of Cadena. The graphical representation of the scenario is shown in Figure 6.25.

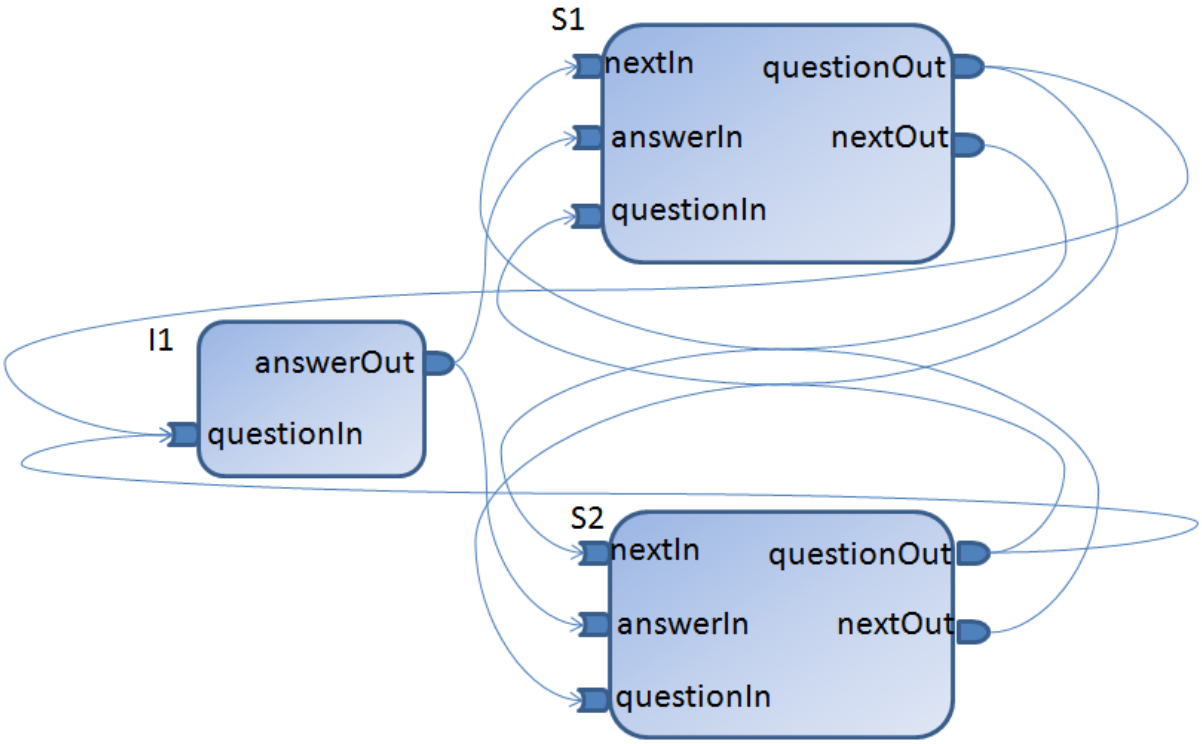

Figure 6.25: *Graphical representation of teleteaching application 3 scenario*

Cadena generated skeleton files per component type. Skeleton file for bidComp component is shown in Listing 6.11).

```
1  public class Instructor extends Component {
2
3     ...
4
5     public void process(Object data_, Port inPort_){
6
7        if (data_ instanceof Message){
8           if (data_ instanceof QuestionInMessage){
9
10          }
11       }
12    }
13 }
```

Listing 6.11: *J-Sim Java skeleton file for Instructor component*

We then added appropriate algorithm specific annotations to the .java files generated by the Cadena tools. An excerpt of the annotated file with annotations to enter critical section is shown in Listing 6.12.

```
1  public class Instructor extends Component {
2
3     ...
4
5     public void process(Object data_, Port inPort_){
6
7        if (data_ instanceof Message){
8           if (data_ instanceof QuestionInMessage){
9
10             /**@cs_request
11             ...
12             /**@cs_release
13
14          }
15       }
16    }
17 }
```

Listing 6.12: *Annotated J-Sim Java skeleton file*

Next, we specified the CPS files.

Optimizations

We first constructed the ADG using our ADG contruction tool. The optimization engine then used the query interface of the ADG to initialize the interaction sets. It also produces dynamic optimization rules along with physical platform optimization information. An excerpt of the file describing static optimization rules for the SRT sets used in the mutual exclusion algorithm is shown in Listing 6.13.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<components>
<component componentName="S1">
<SRT>
</SRT>
</component>
...
```

Listing 6.13: *Sample of static optimization information produced for mutual exclusion service for Application 3.*

For component I1, if it wants to access critical section, it needs to send a request only to components listed as elements of its SRT set. In this case, SRT set is empty. Therefore, when I1 wants to access critical section, it does not need to send request messages at all. Application constraints will guarantee that no other component will be requesting critical section when I1 will. This information might be difficult to arrive at manually. But when the process is automated, the use of customizable generic algorithms is very attractive.

An excerpt from dynamic optimization rules for mutual exclusion algorithm is shown in Listing 6.14.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<dynamicRules>
<RULE componentName="I1">
<currentlyRequesting componentName="S1"/>
</RULE>
...
```

Listing 6.14: *Sample of dynamic optimization information produced for mutual exclusion service for Application 3.*

In this particular case, no dynamic optimization information is generated for mutual exclusion service for component I1 because it is a small scenario. Note, again, that manual inspection is not necessary and saves development time. Optimization information is also derived for termination detection and total ordering services.

Information for physical topology based optimization comes in the form of shortest path information for each pair of processors in a given physical topology. An excerpt from the shortest path information file produced by optimizer for this application is shown in Listing 6.15.

```
<?xml version="1.0" encoding="ASCII" standalone="no"?>
<TopologyStatic maxPath="2">
<Nodes>
<Node nodeName="1">
<Path toNode="2" pathLenght="1">
<PathElement hopCount="1" nodeName="2"/>
</Path>
<Path toNode="3" pathLenght="2">
<PathElement hopCount="1" nodeName="2"/>
<PathElement hopCount="2" nodeName="3"/>
</Path>
...
```

Listing 6.15: *Sample of shortest path information produced for Application 3.*

The csl subcomponent of a J-Sim component will use this information for physical topology based optimization. For instance, if the same message needs to be sent by processor 1 to processors 2 and 3, then only one message will be sent to processor 3. Since processor 2 is on the shortest path from processor 1 to processor 3, it will receive a message from processor 1 to processor 3. A separate message from processor 1 to processor 2 is then not needed.

Discussion

The optimization information generated by the optimizer shows that interaction sets are constrained. This will result in fewer messages as was shown for Applications 1 and 2. Also, teleteaching application 3, in addition to ordering information within groups, utilizes ordering information between groups. The results from evaluation section show that our framework can utilize both local ordering (ordering within groups) and global ordering (ordering between groups) information for possible optimizations.

6.3 Effectiveness of the customization techniques

In this section we evaluate the effectiveness of the customization techniques by comparing the performance of the customized algorithms to those designed for specific operational contexts. We designed optimized algorithms for bidding application 1 from the evaluation section. We will compare the performance of these optimized algorithms to that of our customized algorithms.

Application description

The application involves twelve players making bids. Each player is located on a separate physical machine and the machines are connected as shown in Figure 6.1. The bids that each player receives from other players need to be totally ordered, and only one player can bid at a time. Players are organized into three groups as shown in Figure 6.2. Players in a group make bids in a round-robin fashion (e.g., in group 0, players bid in the order 0,1,2,3,0,1,2,3....). This order is enforced by the application itself. Each player's bid is based on their current group bidding probability, which decreases with each bid made. Once a player in a group decides not to bid, no other player in the group can make any more bids.

This application will require the use of several services. Since bids have to be made in mutually exclusive manner, we need a mutual exclusion service. Bids need to be received in the same order by all players. So, we need a service for total ordering of messages. We also need to know when bidding stopped, so we need to use termination detection service. Algorithms for these services optimized for this specific application are described next.

Optimized mutual exclusion algorithm description

We will use a permission based mutual exclusion algorithm (similar to Lamport's algorithm) for mutual exclusion service. The main idea is that to enter a critical section to make a bid, a process i needs to compete with the processes that might want to enter the critical section at the same time that process i wants to enter it. So, if we have the information about which processes can concurrently enter a critical section, we need to send a request message only

to those processes that can concurrently enter critical section with process i. If process i receives the acks from all those processes and its request is at the head of its request queue, then process i can enter critical section. When mutual exclusion component, mutex, receives a local request message from its application component, app, it sends a request message to the next would be requesting process in each group (those are the processes that might try to enter critical section when process i wants to enter it). This request is also put in mutex's local request queue. If process i receives a request message but is already in critical section (process i will not compete with anybody anymore - i is in critical section), then process i does not need to reply with an ack message since release message will be sent to everybody after process i exits critical section. Then the process that sent the request to process i will know to send a request to next requesting process in process i's group to compete for access to critical section. If process i already sent a request to some processes to compete for access to critical section, and then it receives a request from a process that process i did not send its request to, then process i needs to send its original request to that process too because now process i competes with that process to enter critical section as well. Next, process i needs to send an ack message to that process and add that process' id to wait_rel_from set. If process i already sent a request to that process, then it just needs to send an ack message to that process and add that process' id to wait_rel_from set. In all other cases process i just adds the new request to its request queue, sends an ack message to the process that sent request message to it, and adds the process' id to wait_rel_from set to remember that a release message from that process is expected. When mutex component receives an ack message from another process, it remebers that by storing it in a temporary set. If mutex component receives a release message from another process, it first checks if it is expecting a release message from the message's source. If so, it removes the corresonding request from its request queue. In either case, next_requesting_process information is updated and if the release message is from a process in my group, APP component is notified of that. If process i receives a release message from a process that it sent a request message to before receiving

an ack message from it (which means that that process did not receive my request before sending a release message to me), then process i does not need to compete with that process for critical section (because it already used it). But process i needs to compete now with the next requesting process in that group - therefore, process i needs to send an additional request message to that next requesting process in that group. If mutex component receives all the ack messages it expects and its own request is at the head of the queue, then it enters into in_cs state and notifies APP component that it can enter the critical section now. When APP component is done using critical section and notifies mutex component of that, mutex component broadcasts a release message. We want to broadcast the release message because we want every process to update its next_requesting_process for the group process i is in.

Optimized termination detection algorithm description

Termination detection service is used to determine if a computation is over. Optimized termination detection algorithm for our application is given next. Termination detection component, *tedet*, starts termination detection upon the receipt of START message from the application component, app. It sends out ISPASSIVE message to the next would be bidding process in each group (it is sufficient to do so because application is such that it is known for all other processes to be passive). Upon the receipt of ISPASSIVE message, the process queries its app component about its current status, and then sends corresponding message to the requesting process. If all the responses are passive messages, then termination is detected; if one of the responses is an active message, termination has not happened yet. A release message from mutual exclusion mechanism is also processed here to update next_requesting_process information and for the following situation: if process i initiates termination detection, but receives a release message after that, it means that the next process that is in the group of the process that sent the release message, will try to bid based on its group probability. So, termination did not clearly happen yet. But if that process decides not to bid anymore, it will start termination detection.

Optimized total order algorithm description

When a process receives a message, it is put into a local queue, ordered according to its timestamp. The receiver multicasts an acknowledgement to the other processes. Note, that if we follow Lamport's algorithm for adjusting local clocks, the timestamp of the received message is lower than the timestamp of the acknowledgement. All processes will eventually have the same copy of the local queue. Each message is multicast to all processes, including acknowledgements, and is assumed to be received by all processes. We assume that messages are delivered in the order that they are sent. Each process puts a received message in its local queue according to the timestamp in that message. Lamport's clocks ensure that no two messages have the same timestamp, but also that the timestamps reflect a consistent global ordering of events. A process can deliver a queued message to the application it is running only when that message is at the head of the queue and has been acknowledged by each other process. At that point, the message is removed from the queue and handed over to the application; the associated acknowledgements can simply be removed. Because each process has the same copy of the queue, all messages are delivered in the same order everywhere. In other words, we have established totally-ordered multicasting.

Optimization for total order protocol comes in the form of eleminating ack messages. Each application message carries with it a sequence number. When total order component, *to*, receives an application message for broadcast from application component, app, it broadcasts it and also puts it in local queue. When *to* component receives an application message from another process, it puts it in its local queue. A local sequence number counter is used to store information on the next sequence number expected. If the message at the head of the queue has the expected sequence number, the message is delivered to app component. The global sequence number is incremented each time a process enters a critical section. It is updated to the most current one upon receipt of an application message message in app component (app components keep track of that sequence number).

Comparison of customized and optimized algorithms

Table 6.3 and Figure 6.26 show the average number of messages per bid for five runs of our system. The averages are shown for mutual exclusion (ME), termination detection (TD), and total ordering (TO) algorithms. The averages are shown for customized algorithms with Static and Dynamic Optimization (SD_Opt) and for optimized algorithms.

	ME	TD	TO
Cusomized version with SD_Opt	46	17	293
Optimized version	46	15	28

Table 6.3: *Comparison of average number of messages per bid for customized and optimized algorithms for Application 1*

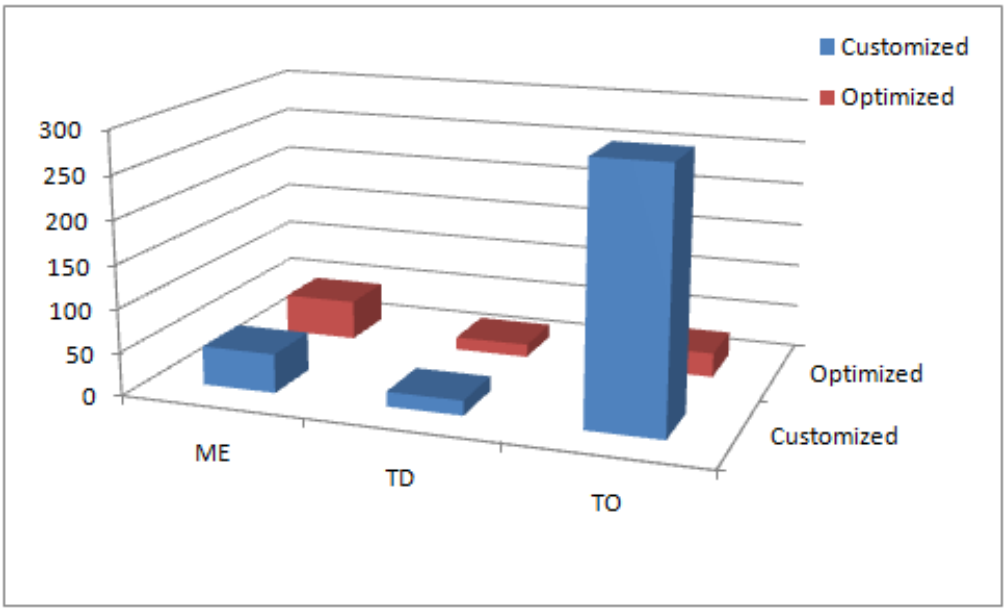

Figure 6.26: *Comparison of customized and optimized algorithms for Application 1*

The results show that our customization algorithms for mutual exclusion and termination detection perform as well as optimized algorithms developed from scratch. Optimized algorithm for total ordering outperforms the customized algorithm. The reason for that is that ack messages are completely eliminated in the optimized version by making application messages to carry more application specific information and by using control messages of the mutual exclusion algorithm.

Although, as we have shown, it is possible to come up with more efficient optimized algorithms for some services as opposed to our customized versions, it is possible to do so by manually overloading the message content and using information of other services. That also requires application level programming to assist services layer. That cannot be done automatically by tools. Also, such an approach is highly undesirable because services and application logic is interconnected and cannot be reused in other contexts.

6.4 Summary

InDiGO framework utilizes ordering information on events for possible optimizations and thus is best suited for applications whose components issue events in some order. In this chapter we looked at a class of bidding applications and showed that the more ordering constraints the application itself places on its components, the more optimization opportunities there will be. Our framework produced optimization information that resulted in higher optimization for the application with higher level of ordering constraints that the application itself inforced on its components. We also showed that ordering patterns could be effectively exploited by InDiGO framework as in the case of teleteaching applications where answer messages are issued in response to question messages. We also showed that the optimization achieved with InDiGO framework is comparable with optimization of algorithms developed from scratch for a bidding application that we studied in this thesis.

Chapter 7

Conclusion and future work

In this thesis, we proposed an extensible infrastructure to optimize distributed algorithms for specific applications. The capabilities of our framework include the following: (a) Infrastructure to capture application information, (b) Mechanisms to design customizable algorithms, and (c) Optimization tools. We demonstrated that by allowing the algorithm designer to capture and expose design knowledge, optimization opportunities can be realized. For example, by exposing application knowledge gained as a result of message passing within the algorithm, we were able to perform dynamic optimizations. We performed a series of experiments on the classes of bidding and teleteaching applications to demonstrate the different types of optimizations. As the number of constraints on the order in which the components can perform actions were increased in the application, we showed that more optimizations were possible.

Each of the capabilities provided by InDiGO can be extended for a richer set of optimizations. For example, we currently support interaction sets of an algorithm as a configurable option. The infrastructure allows the algorithm designers to expose any other design information which can be analyzed with respect to the ADG. The assertions, $alg.app\_assert$, is one example of additional knowledge which we have exploited for optimization. Although we have used specific algorithms as case studies, the framework applies to other algorithms as well. Furthermore, the framework is extensible in that more sophisticated analysis tools can be plugged in to analyze the already available information for more aggressive optimization.

Similarly, one can develop artifacts to expose more information about the application and the algorithm. As more information about the algorithm is made available, more will be the possible optimization opportunities. Note that the algorithm designers may not know of the potential optimizations (the designer simply exposes algorithm information). The optimization tools leverage this information to uncover optimization opportunities. Similarly, in constructing the ADG, we have abstracted several details of the application and retained structural information only. One can include more details in the application models to allow better analysis. Finally, the type of optimizations targeted by InDiGO are different from those performed by existing techniques which exploit application semantics. For example, techniques have been proposed to use conflict relations on messages to optimize message ordering and concurrency control algorithms.[3,17,19] Conflict relations are based on semantics of message types and the underlying algorithms implement the conflict relation irrespective of the order in which the application may send messages. Our work, on the other hand, analyzes the application structure for optimization.

In the future work, we plan to do the following:

• We will investigate a general problem of a domain specification language for the annotations that we used to specify middleware services in a distributed system. In this work, we used annotations which were sufficient for studied services only.

• In collaboration with Cadena developers, one of the capabilities to be built is a tool to derive the CPS files from the annotated Java files for each component. At present, for some of the experimental studies, we had to manually specify the CPS files.

• Another goal in Cadena is to derive more detailed CPS files for Java files which will result in more accurate application models. This will allow to capture more information about the application in the ADG graph, the analysis of which can reveal more optimization opportunities.

• Also, we will investigate if more sophisticated analysis algorithms could be developed to analyze the ADG graph for aggressive optimizations. These algorithms then can can be

plugged into the InDiGO tool-chain.

• We will include other customizable distributed services in InDiGO framework.

• In this work, we specified interaction sets specific to the algorithms studied here. We will investigate the use of interaction sets for a class of algorithms. For example, the SRT set could be common to mutual exclusion algorithms in general.

• We will investigate additional techniques to design distributed algorithms and middleware amenable to customization.

Bibliography

[1] K. Birman and R. van Renesse, *Reliable Distributed Computing with the ISIS toolkit*, IEEE Computer Society Press, 1994.

[2] R. Guerraoui and A. Schiper, Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems, in *IEEE International Symposium on Faul-Tolerant Computing Systems (FTCS)*, 1996.

[3] M. Kalantar and K. Birman, Causally ordered multicast: the conservative approach, in *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems*, 1999.

[4] R. F. K. Birman and M. Hayden, The maestro group manager: A structuring tool for applications with multiple quality of service requirements, in *Technical Report TR97-1619, Department of Computer Science, Cornell University*, 1997.

[5] B. Ban, *JavaGroups - A Reliable Multicast Communication Toolkit for Java*, http://www.cs.cornell.edu/Info/Projects/JavaGroupsNew, 1999.

[6] P. Felber and R. Guerraoui, Programming with object groups in corba, in *IEEE Concurrency*, 2000.

[7] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam, Group communication support for distributed collaboration systems, in *Proc. of IEEE 17th International Conference on Distributed Computing Systems*, 1997.

[8] L. Moser, P. Melliar-Smith, D. Agrawal, R. Budhia, and C. Lingley-Papadopoulos, Totem: a fault-tolerant multicast group communication system, in *Communication of the ACM*, volume 39, 1996.

[9] Y. Amir, D. Dolev, S. Kramer, and D. Malki, Transis: a communication subsystem for high availability, in *International Symposium on Fault-Tolerant Computing*, 1992.

[10] N. Bhatti and R. Schlichting, A system for constructing configurable high-level protocols, in *Proceedings of ACM SIGCOMM Conference*, 1995.

[11] R. Guerraoui and A. Schiper, Total order multicast to multiple groups, in *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems*, 1997.

[12] D. Sharp, Avionics product line software architecture flow policies, in *Proceedings of the Digital Avionics Systems Conference*, 1999.

[13] G. Kiczales et al., Aspect-oriented programming, in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, 1997.

[14] R. Pratap and R. Cytron, Transport layer abstraction in event channels for embedded systems, in *LCTES*, 2003.

[15] A. Agbaria and W. H. Sanders, Application-driven coordination-free distributed checkpointing, in *Proceedings of the IEEE Int'l Conference on Distributed Computing Systems*, 2005.

[16] G. Chockler, I. Kedar, and R. Vitenberg, Group communication frameworks: A comprehensive survey, in *ACM Computing Surveys*, 2001.

[17] P. Jensen, N. Soparkar, and A. Mathur, Characterizing multicast orderings using concurrency control theory, in *International Conference on Distributed Computing Systems*, 1997.

[18] L. Liu and C. Pu, A transactional activity model for organizing open-ended cooperative activities, in *Proceedings of the Hawaii International Conference on System Sciences*, 1998.

[19] V. Murty and V. Garg, Characterization of message ordering specifications and protocols, in *Proc. IEEE International Conference on Distributed Computing Systems 1997*, 1997.

[20] A. Sobeih et al., J-Sim: a simulation and emulation environment for wireless sensor networks, in *Proc. 38th Annual Simulation Symposium*, 2005.

[21] A. Childs et al., Cadena: An integrated development environment for analysis, synthesis, and verification of component-based systems, in *FASE*, edited by M. Wermelinger and T. Margaria, volume 2984 of *Lecture Notes in Computer Science*, pages 160–164, Springer, 2004.

[22] T. Harrison, D. Levine, and D. Schmidt, The design and performance of a real-time corba event service, in *Proceedings of OOPLSA*, 1997.

[23] C. Ma and J. Bacon, Cobea: A corba-based event architecture, in *Proceedings of USENIX COOTS*, 1998.

[24] O. M. Group, The common object request broker: Architecture and specification, revision 2.0, 1995.

[25] G. Ricart and A. K. Agarwala, An optimal algorithm for mutual exclusion in computer networks, in *Communications of the ACM*, volume 24, pages 9–17, 1981.

[26] Y.-J. Joung, Aynchronous group mutual exclusion in ring networks, in *Proceedings of International Parallel Processing Symposium*, 1999.

[27] M. Maekawa, A $\sqrt{N}$ algorithm for mutual exclusion, in *ACM Transactions on Computer Systems*, volume 2, 1985.

[28] K. Raymond, A tree-based algorithm for distributed mutual exclusion, in *ACM Transactions on Computer Systems*, volume 1, 1989.

[29] M. Neilsen and M. Mizuno, A dag-based algorithm for distributed mutual exclusion, in *Proceedings of the IEEE 11th International Conference on Distributed Computing Systems*, pages 354–360, 1991.

[30] B. Topol, M. Ahamad, and J. Stasko, Robust state sharing for wide area distributed applications, in *International Conference on Distributed Computing Systems*, 1998.

[31] H. Kung and J. Robinson, On optimistic methods for concurrency control, in *ACM Transactions on Database Systems*, volume 6, pages 213–226, 1981.

[32] S. Meldal, S. Sankar, and J. Vera, Exploiting locality in maintaining potential causality, in *Symposium on Principles of Distributed Computing*, pages 231–239, 1991.

[33] S. Quaireau and P. Laumay, Ensuring applicative causal ordering in autonomous mobile computing, in *Workshop on Middleware for Mobile Computing*, 2001.

[34] C. Cowan and H. Lutfiyya, Formal semantics for expressing optimism: The meaning of HOPE, in *Symposium on Principles of Distributed Computing*, pages 164–173, 1995.

[35] R. Prakash, M. Raynal, and M. Singhal, An adaptive causal ordering algorithm suited to mobile computing environments, in *Journal of Parallel and Distributed Computing*, volume 41, 1997.

[36] S. Guyer and C. Lin, Broadway: A compiler for exploiting the domain-specific semantics of software libraries, in *Proceedings of the IEEE*, volume 93, pages 342–357, Feb. 2005.

[37] N. Francez and I. Forman, *Interacting Processes: A multiparty approach to coordinated distributed programming*, Addison-Wesley, 1996.

[38] M. Evangelist, N. Francez, and S. Katz, Multiparty interactions for interprocess communication and synchronization, in *IEEE Transactions on Software Engineering*, volume 15, 1989.

[39] D. Garlan and R. Allen, Formalizing architectural connections, in *IEEE International Conference on Software Engineering*, 1994.

[40] D. Luckman et al., Specification and analysis of system architecture using Rapide, in *IEEE Transactions on Software Engineering*, volume 21, 1995.

[41] D. Sturman, *Modular specification of interaction policies in distributed computing*, PhD thesis, University of Illinois at Urbana Champaign, 1996.

[42] G. Singh, P. S. Kumar, and Q. Zeng, Configurable event communication in cadena, in *IEEE Conference on Real-time Applications and Systems*, 2004.

[43] P. S. Kumar, Q. Zeng, and G. Singh, Constraining event flow for regulation in pervasive systems, in *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, pages 314–318, Washington, DC, USA, 2005, IEEE Computer Society.

[44] G. Trombetti et al., An integrated model-driven development environment for composing and validating distributed real-time and embedded systems, in *Model-Driven Software Development*, edited by M. B. S. Beydeda and V. Gruhn, Springer-Verlag, 2005.

[45] L. Chen and G. Singh, Enhancing multicast communication to support protocol design, in *IEEE International Conference on Computer Communication and Networks*, 2002.

[46] G. Singh and S. Das, Customizing event ordering middleware for component-based systems, in *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 359–362, Washington, DC, USA, 2005, IEEE Computer Society.

[47] G. Singh, B. Maddula, and Q. Zeng, Enhancing event channel for synchronization in

object oriented distributed systems, in *Proceedings of IEEE International Symposium on Object Oriented Real-time Computing*, 2002.

[48] G. Singh and Y. Su, Region synchronization in message passing systems, in *Proceedins of the International Conference on Parallel Processing*, 2002.

[49] Y. Su, *Synchronization in Message Passing Systems*, PhD thesis, Kansas State University, 2004.

[50] M. Mizuno, G. Singh, and M. Neilsen, A structured approach to develop concurrent programs in uml, in *Proceedings of the Third International Conference on the Unified Modeling Language*, 2000.

[51] L. Lamport, Time, clocks, and the ordering of events in a distributed system, in *Communications of the ACM*, volume 21, pages 558–565, 1978.

[52] K. M. Chandy and J. Misra, A paradigm for detecting quiescent properties in distributed computations, in *Logics and models of concurrent systems*, pages 325–341, New York, NY, USA, 1985, Springer-Verlag New York, Inc.

[53] A. Tanenbaum, Distributed systems, Prentice Hall, 2006.

Appendix A

Grammar for CPS files

```
compilationUnit     ::=  <MODULE> <IDENTIFIER> <LBRACE> moduleDeclaration <RBRACE> <EOF>
moduleDeclaration   ::=  ( componentBehavior )*
componentBehavior   ::=  <COMPONENT> <IDENTIFIER> <LBRACE> behaviorBody <RBRACE>
behaviorBody        ::=  declarations specifications
declarations        ::=  (<MODE> <IDENTIFIER> <SEMICOLON> )*
specifications      ::=  dependencies
dependencies        ::=  ( <DEPENDENCIES> <LBRACE> depBody <RBRACE> )?
depBody             ::=  ( relayStatement )*
relayStatement      ::=  <IDENTIFIER> <RIGHTARROW> actionPart
actionPart          ::=  ( action <SEMICOLON>)* action
                    |    rCaseStatement
rCaseStatement      ::=  <CASE> <IDENTIFIER> <OF> <LBRACE> rCaseList <RBRACE>
rCaseList           ::=  ( <IDENTIFIER> ":" ( action <SEMICOLON> )* action )+
action              ::=  <IDENTIFIER>
```

Figure A.1: *Grammar for CPS files*

Appendix B

Grammar for membership criteria

```
criteria        ::=   ( criterion )*

criterion       ::=   setName <COMMA> allnbr
                |     set <COMMA> allnbr <COMMA> query
                |     <IF> <EXISTS> setID <COLON> suchCondition <AND> query
                      <THEN> <MIN> <LBRACE> setID <COLON> suchCondition <AND> query <RBRACE>
                      <ELSE> <MIN> <LBRACE> setID <COLON> suchCondition <AND> query <RBRACE>
                |     <IF>  condition <THEN> assignment <ELSE> assignment

set             ::=   setName <DOT> setID

query           ::=   prefix queryName <COMMA> parameters

parameters      ::=   action <COMMA> action
                |     action <COMMA> action <COMMA> ifelseondition
                |     setCondition <COMMA> setCondition

suchCondition   ::=   setID sign setID  sign setID

action          ::=   setID <DOT> <IDENTIFIER>
```

Figure B.1: *Grammar for membership criteria (part 1)*

setCondition	::=	**setID** <DOT> <IDENTIFIER> <EQ> <IDENTIFIER>
condition	::=	<IDENTIFIER> **sign** **expression**
ifelsecondition	::=	<IF> **setID** <DOT> <STATE> <EQ> <IDENTIFIER>
		<THEN> **setID** <DOT> <IDENTIFIER> <EQ> **nodeState**
assignment	::=	<IDENTIFIER> <EQ> **expression**
expression	::=	<IDENTIFIER> | <IDENTIFIER> (<PLUS> | <MINUS>) <IDENTIFIER>
nodeState	::=	<ENABLED> | <DISABLED>
allnbr	::=	<ALL> | <NEIGHBOURS>
sign	::=	<LT> | <LEQ>
queryName	::=	<IDENTIFIER>
prefix	::=	<NOT> | <EMPTY>
setName	::=	<IDENTIFIER>
setID	::=	<IDENTIFIER>

Figure B.2: *Grammar for membership criteria (part 2)*

Appendix C

Case study

In this appendix we apply InDiGO framework to study a class of distributed applications. In particluar, we apply our framework to study a class of bidding applications. In evaluation chapter, we did not study the effect of increasing the number of clusters on optimization level. Nor did we study the effect of increasing the number of components per cluster. So, in this appendix we take the same bidding application that we used in evaluation chapter, but we want to study how optimization level is affected by varying several application parameters, such as number of clusters, number of components per cluster, number of clusters with local ordering and number of components per process. The results of this study will help us to answer questions like: What type of application information is useful for optimization in our approach? or How does the application structure or its size affect the level of optimization?

Application description

The application is the same as in evaluation chapter and involves players making bids. The bids that each player receives from other players need to be totally ordered, and only one player can bid at a time. Players are logically organized into groups. Players in each group make bids in a round-robin fashion. This order is enforced by the application itself. Each player's bid is based on their current group bidding probability, which decreases with each bid made. Once a player in a group decides not to bid, no other player in the group can make any more bids. We need to know when bidding stops.

Varying number of clusters

In this section we will vary the number of clusters from 1 to 2 to 4 to 8. The number of players per cluster is four and will remain the same. Players in each group make bids in a round-robin fashion. Each player component is located on a separate physical machine. Application and physical topologies for each case are shown in Figures C.1 - C.8.

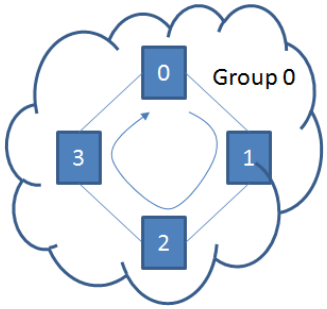

Figure C.1: *Case study - varying number of clusters - logical topology of application with 1 cluster*

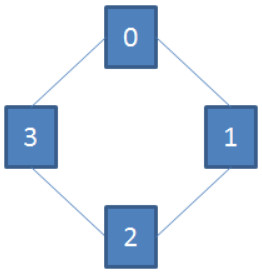

Figure C.2: *Case study - varying number of clusters - physical topology of application with 1 cluster*

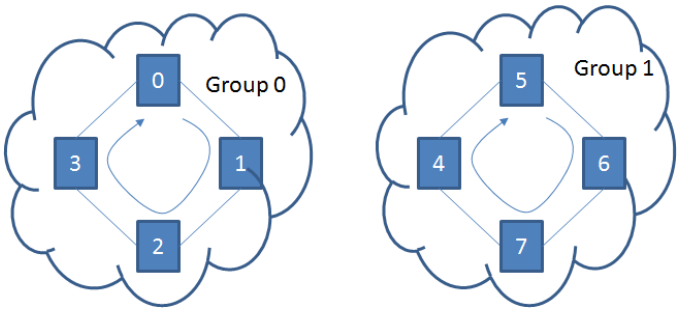

Figure C.3: *Case study - varying number of clusters - logical topology of application with 2 clusters*

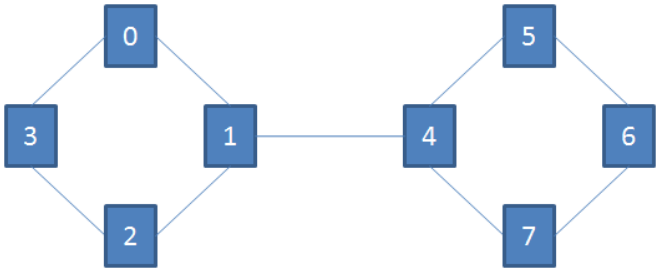

Figure C.4: *Case study - varying number of clusters - physical topology of application with 2 clusters*

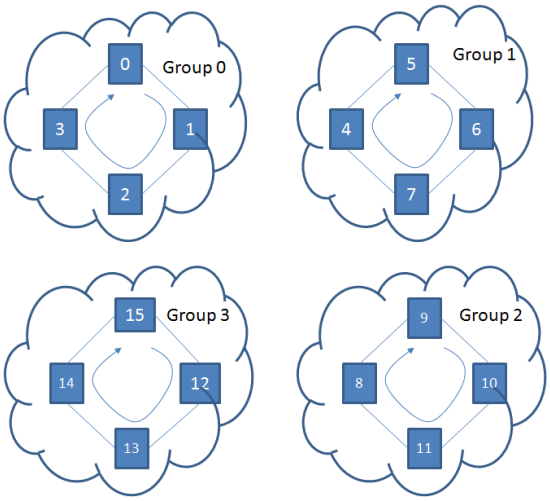

Figure C.5: *Case study - varying number of clusters - logical topology of application with 4 clusters*

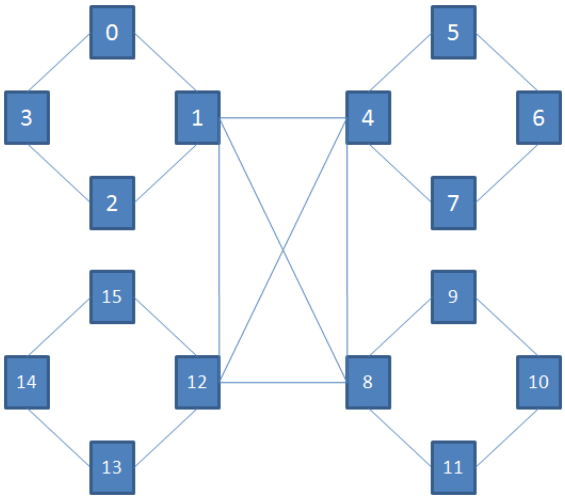

Figure C.6: *Case study - varying number of clusters - physical topology of application with 4 clusters*

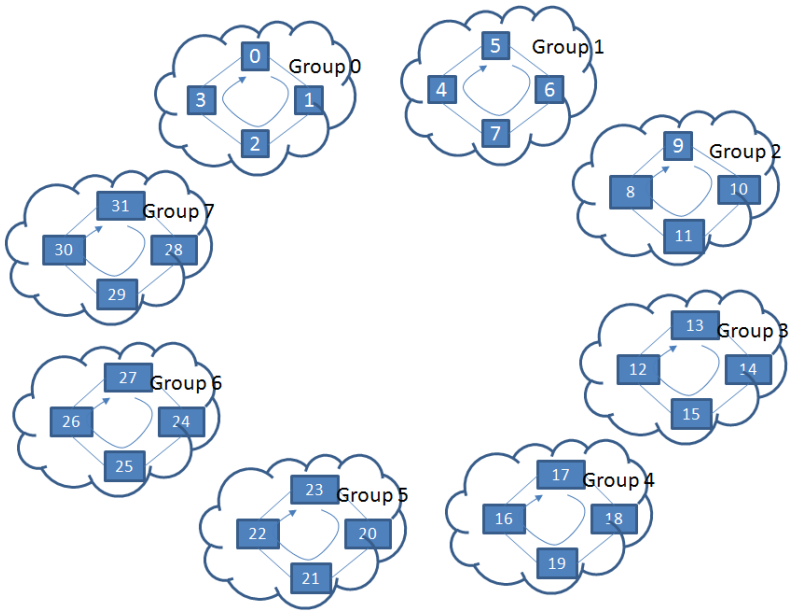

Figure C.7: *Case study - varying number of clusters - logical topology of application with 8 clusters*

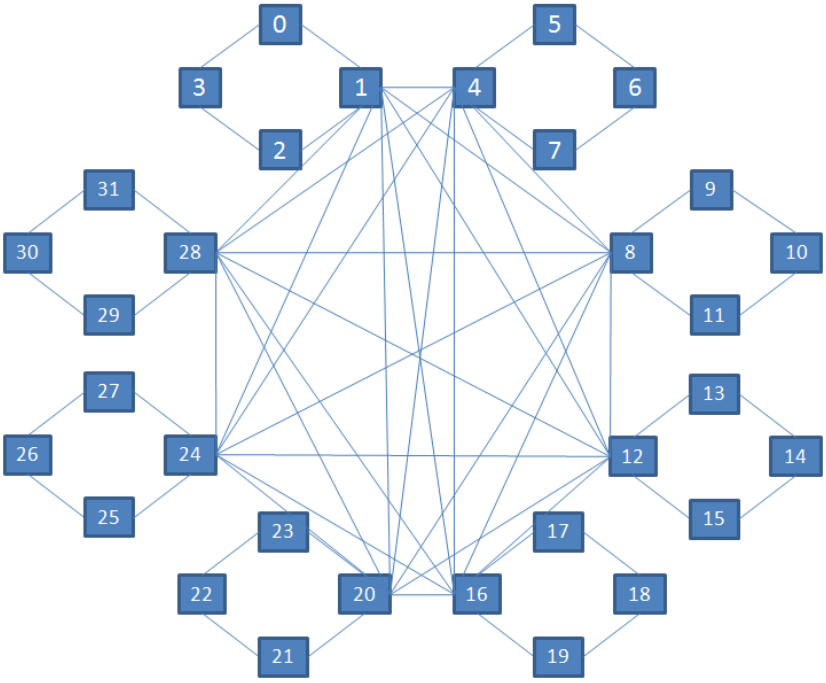

Figure C.8: *Case study - varying number of clusters - physical topology of application with 8 clusters*

The results for the improvement over No_Opt case for studied services are shown in Figures C.9 - C.12.

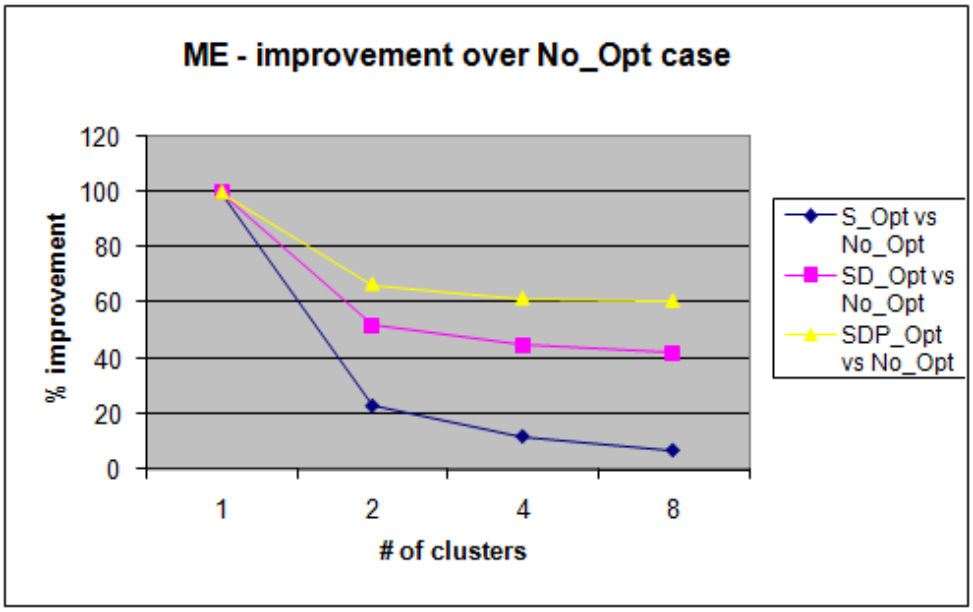

Figure C.9: *Case study - varying number of clusters - % improvement over No_Opt case for mutual exclusion service*

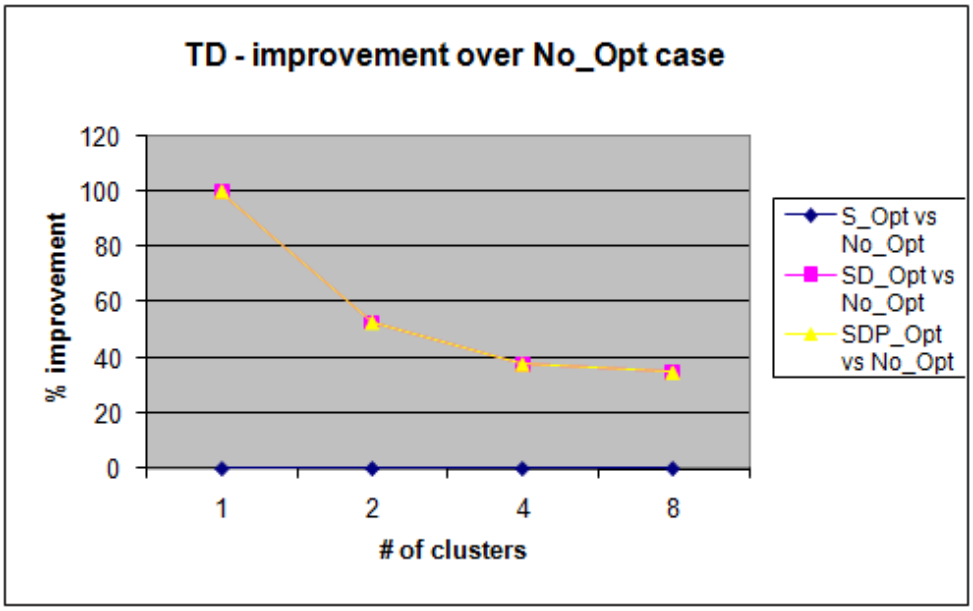

Figure C.10: *Case study - varying number of clusters - % improvement over No_Opt case for termination detection service*

The results for the improvement over previous level of optimization for studied services are shown in Figures C.13 - C.16.

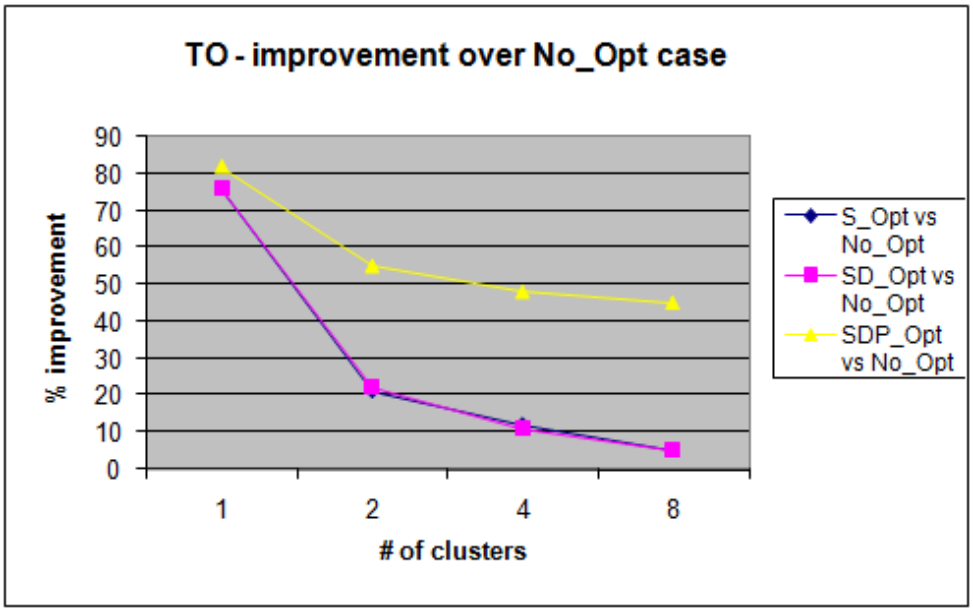

Figure C.11: *Case study - varying number of clusters - % improvement over No_Opt case for total ordering service*

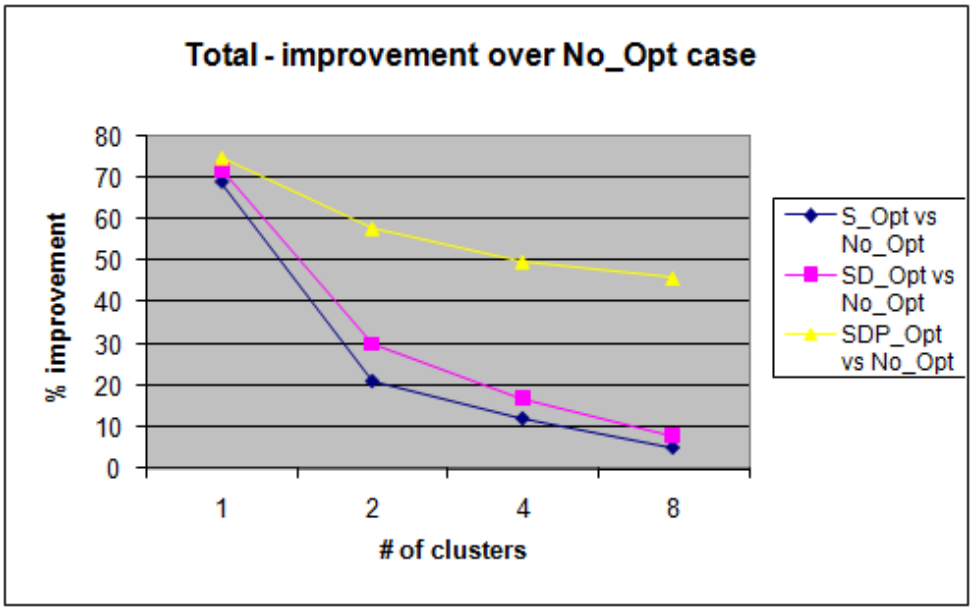

Figure C.12: *Case study - varying number of clusters - % improvement over No_Opt case for total number of messages*

The results show the following trends:

For mutual exclusion service, as the number of clusters with local ordering information

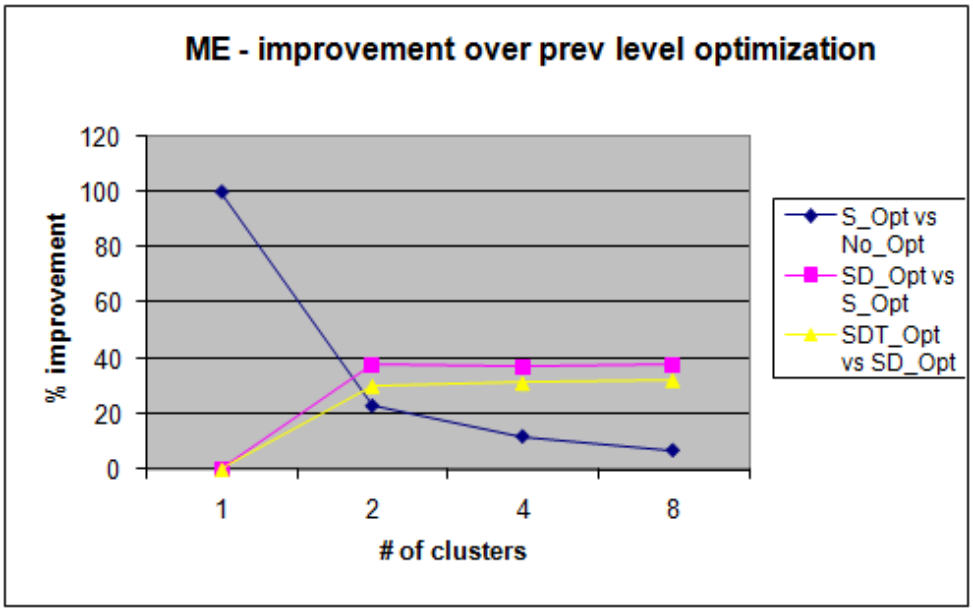

Figure C.13: *Case study - varying number of clusters - % improvement over previous level of optimization for mutual exclusion service*

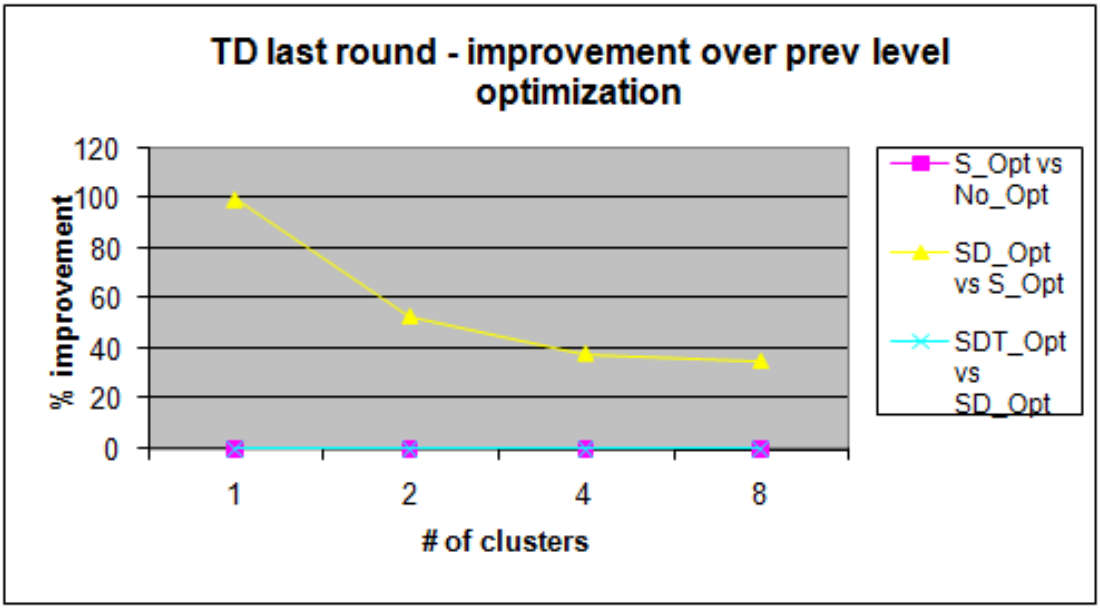

Figure C.14: *Case study - varying number of clusters - % improvement over previous level of optimization for termination detection service*

becomes larger, improvements due to static optimization become less and less significant. Dynamic optimization is significant. It does not depend on the number of clusters and stays

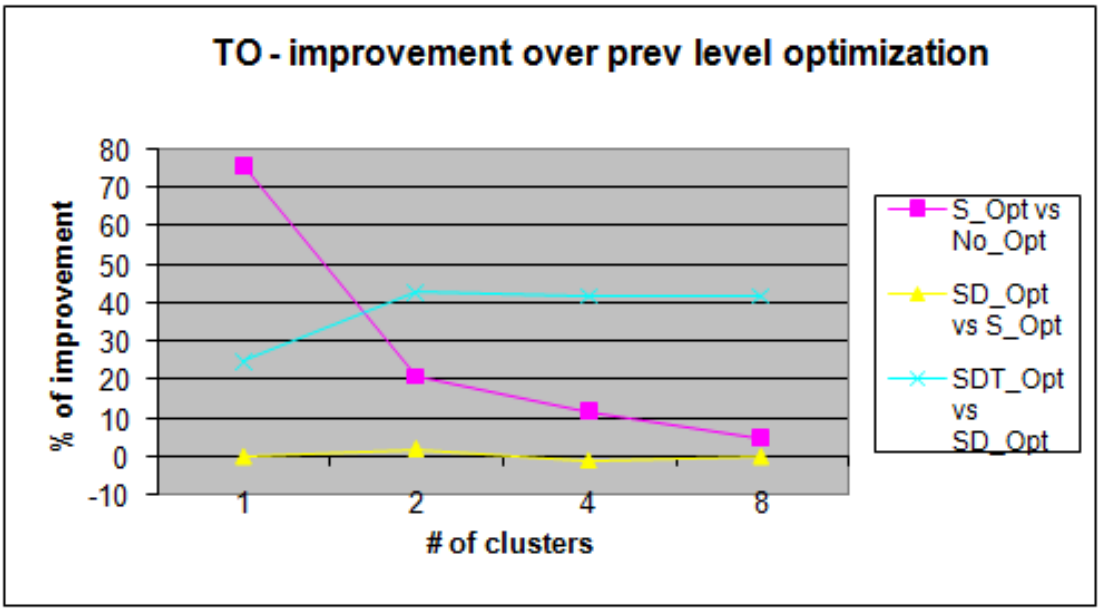

Figure C.15: *Case study - varying number of clusters - % improvement over previous level of optimization for total ordering service*

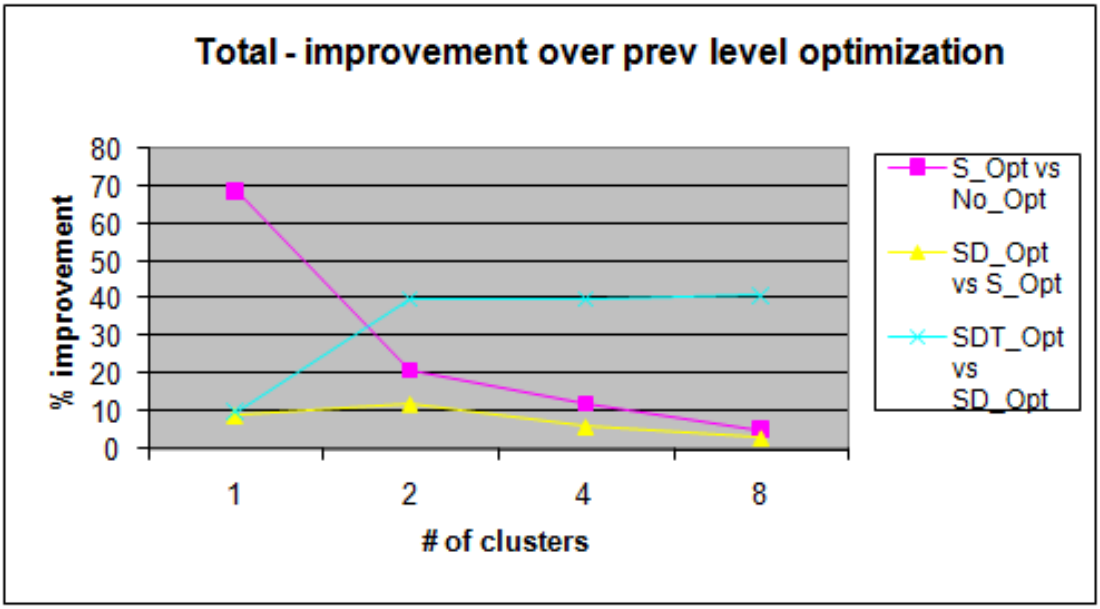

Figure C.16: *Case study - varying number of clusters - % improvement over previous level of optimization for total number of messages*

the same. Physical topology based optimization further contributes to improvement in the number of messages and stays the same as the number of clusters grow. For termination

detection service, only dynamic optimization contributes to the improvement in the number of messages. As the number of clusters with local ordering information grows, the improvement does not change significantly. For total ordering service, as the number of clusters with local ordering information becomes larger, improvements due to static optimization become less and less significant. Dynamic optimization is not a contributing factor. Physical topology based optimization is significant and stays the same as the number of clusters grow.

Some results were not obvious but our framework tools correctly captured necessary information that resulted in optimization. For example, in the case of one cluster for termination detection algorithm no messages needed to be sent out for the last round. Application information was rightly utilized and dynamic optimization resulted in no messages for the last round.

Varying number of components per cluster

In this section we will vary the number of components per cluster from 1 to 2 to 4 to 8. The number of clusters is four and will remain the same. Players in each group make bids in a round-robin fashion. Each player component is located on a separate physical machine. Application and physical topologies for each case are shown in Figures C.17 - C.24.

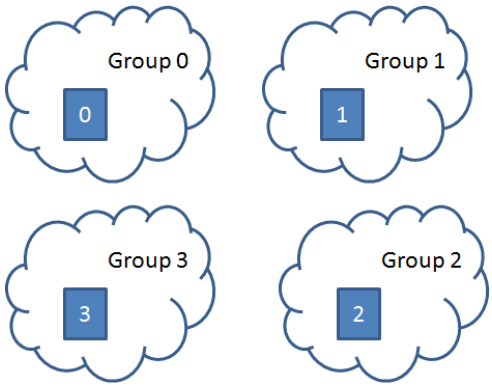

Figure C.17: *Case study - varying number of components per cluster - logical topology of application with 1 component per cluster*

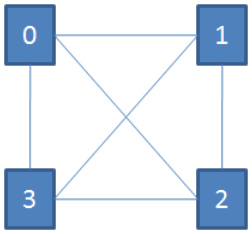

Figure C.18: *Case study - varying number of components per cluster - physical topology of application with 1 component per cluster*

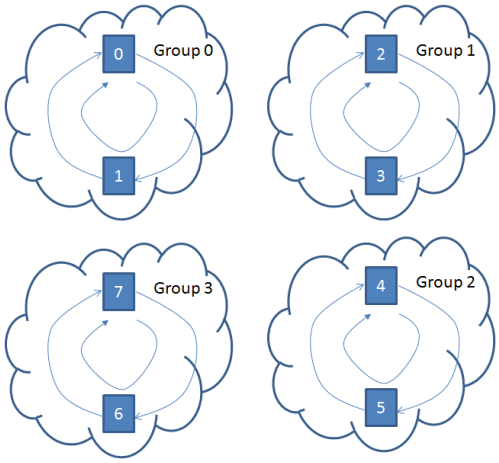

Figure C.19: *Case study - varying number of components per cluster - logical topology of application with 2 components per cluster*

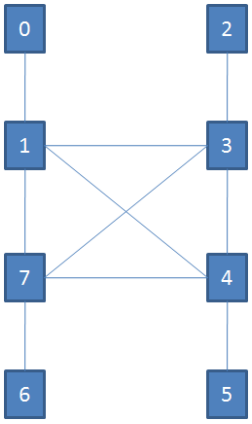

Figure C.20: *Case study - varying number of components per cluster - physical topology of application with 2 components per cluster*

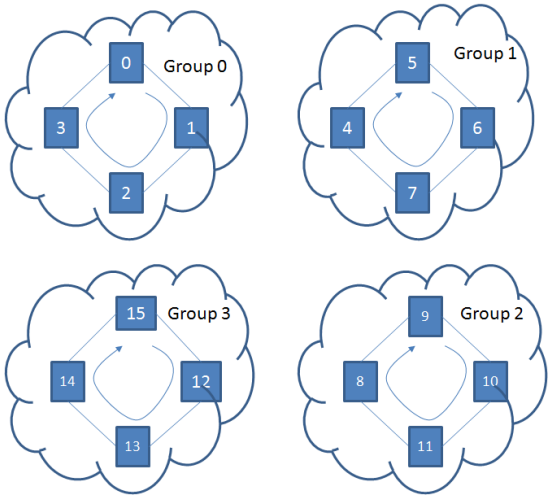

Figure C.21: *Case study - varying number of components per cluster - logical topology of application with 4 components per cluster*

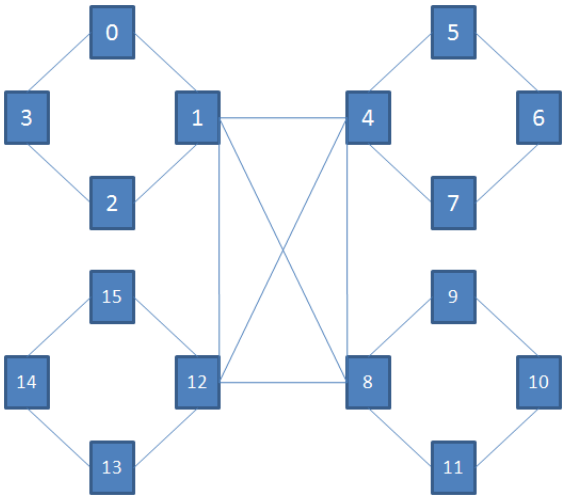

Figure C.22: *Case study - varying number of components per cluster - physical topology of application with 4 components per cluster*

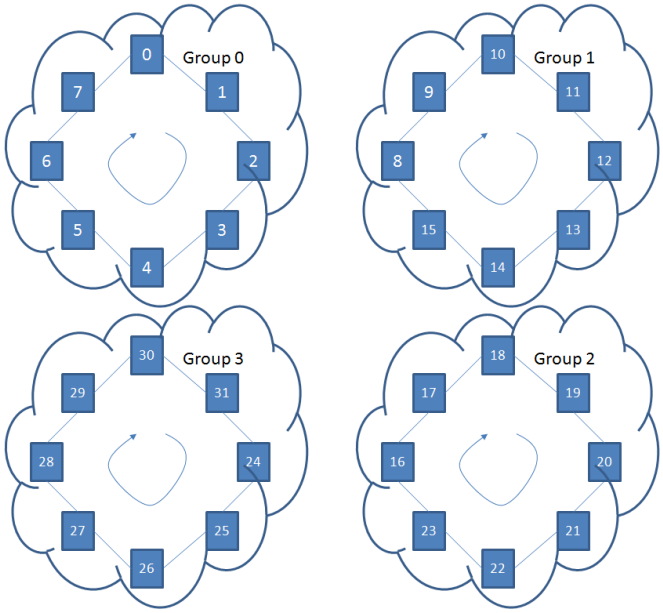

Figure C.23: *Case study - varying number of components per cluster - logical topology of application with 8 components per cluster*

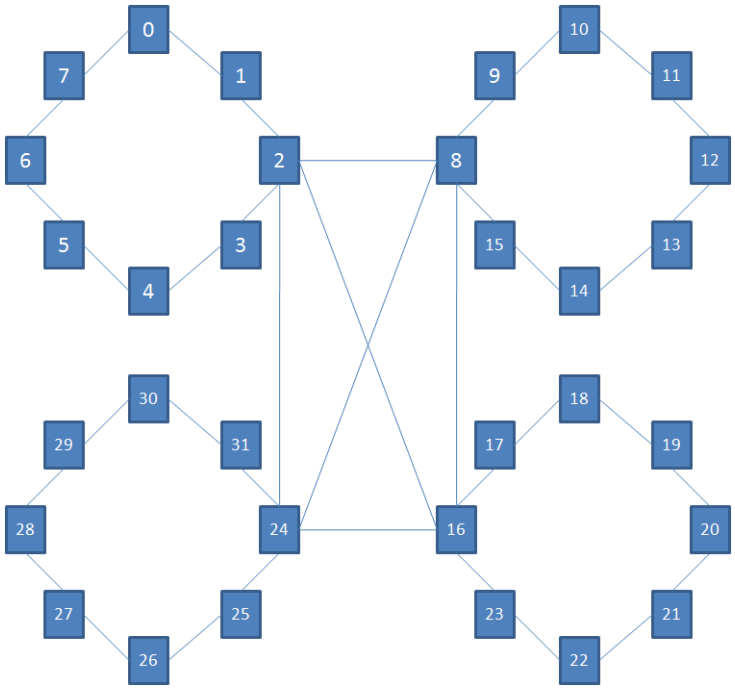

Figure C.24: *Case study - varying number of components per cluster - physical topology of application with 8 components per cluster*

The results for the improvement over No_Opt case for studied services are shown in Figures C.25 - C.28.

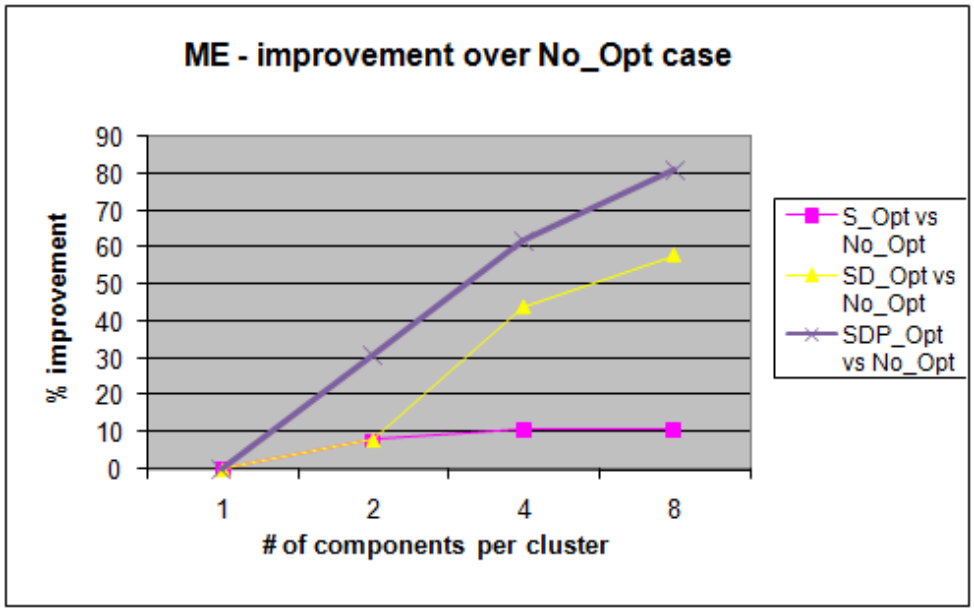

Figure C.25: *Case study - varying number of components per cluster - % improvement over No_Opt case for mutual exclusion service*

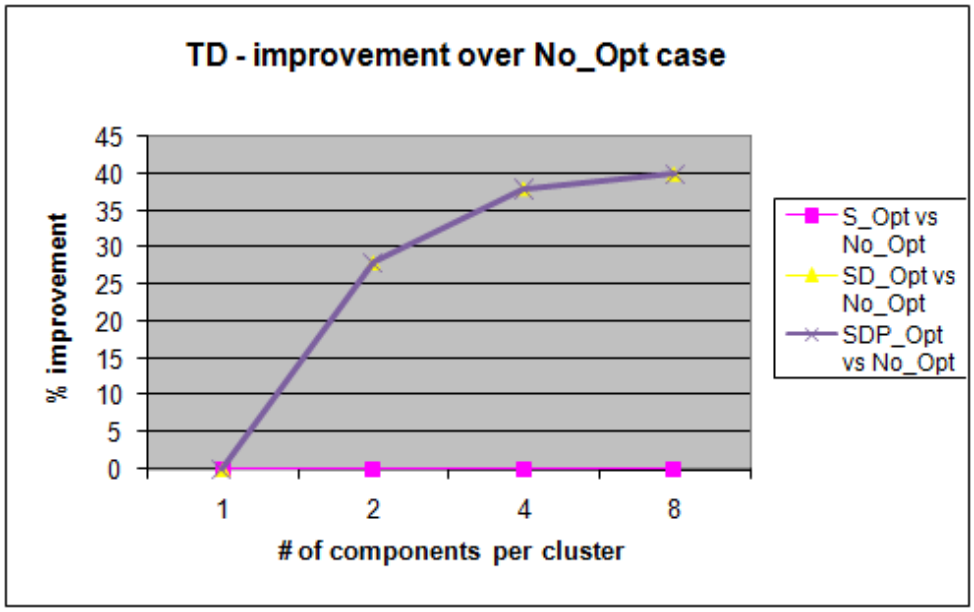

Figure C.26: *Case study - varying number of components per cluster - % improvement over No_Opt case for termination detection service*

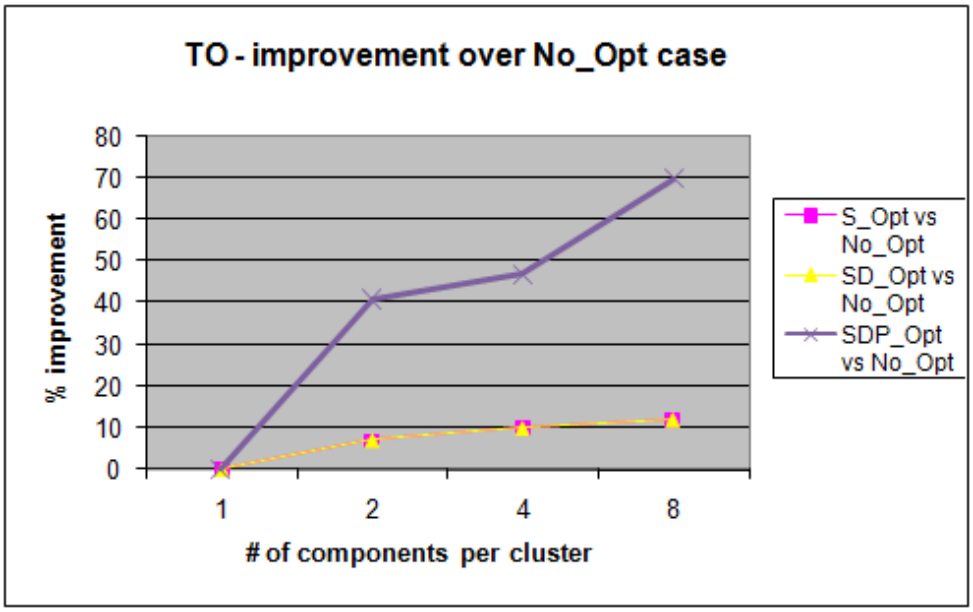

Figure C.27: *Case study - varying number of components per cluster - % improvement over No_Opt case for total ordering service*

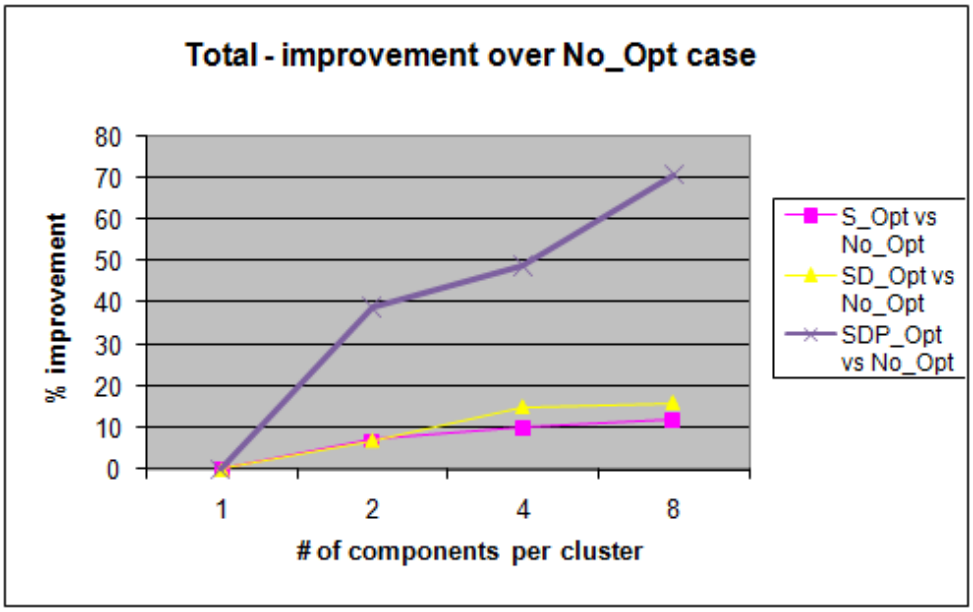

Figure C.28: *Case study - varying number of components per cluster - % improvement over No_Opt case for total number of messages*

The results for the improvement over previous level of optimization for studied services are shown in Figures C.29 - C.32.

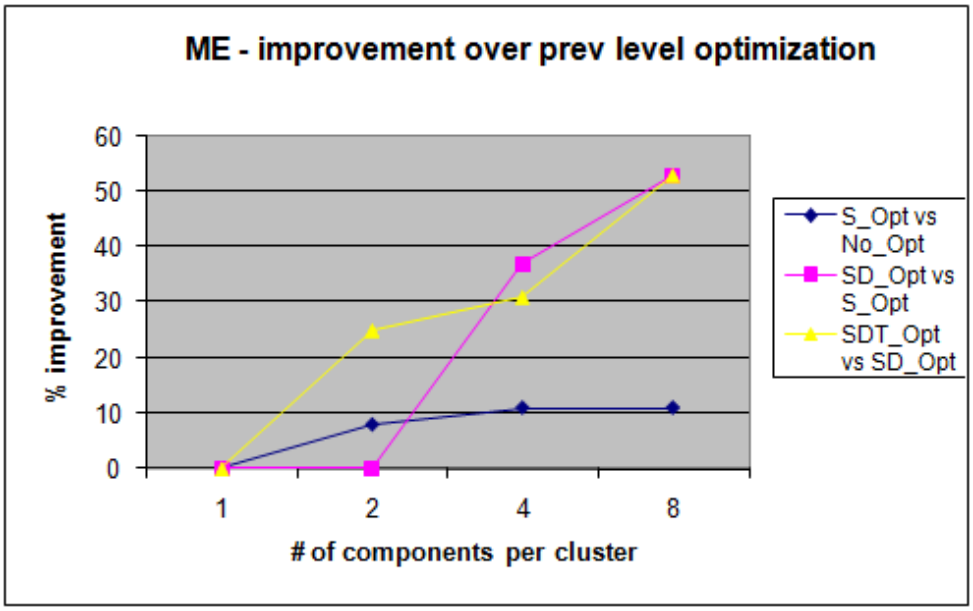

Figure C.29: *Case study - varying number of components per cluster - % improvement over previous level of optimization for mutual exclusion service*

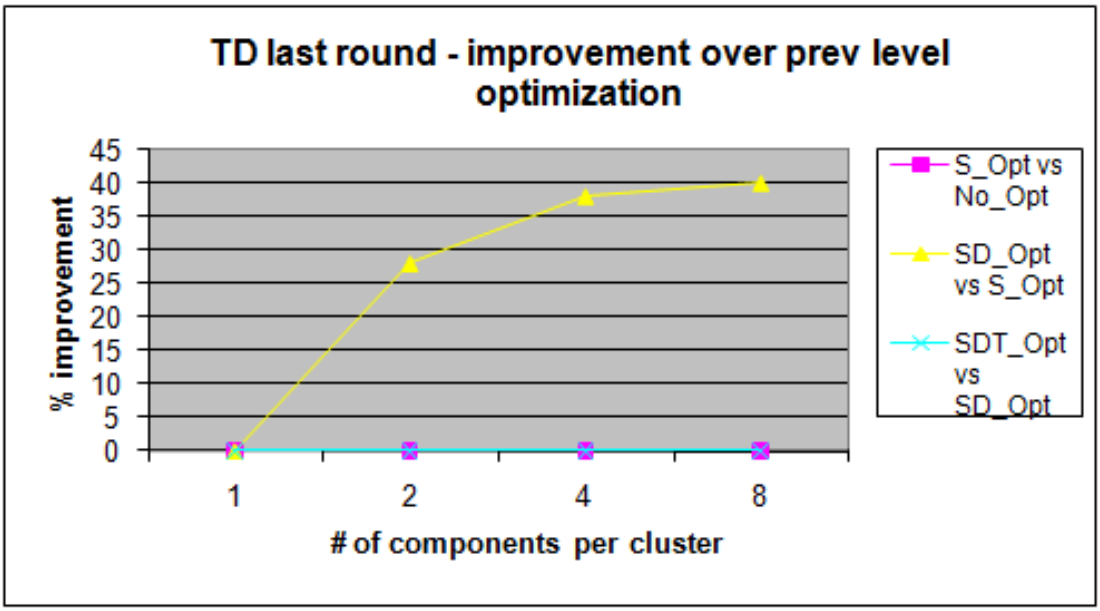

Figure C.30: *Case study - varying number of components per cluster - % improvement over previous level of optimization for termination detection service*

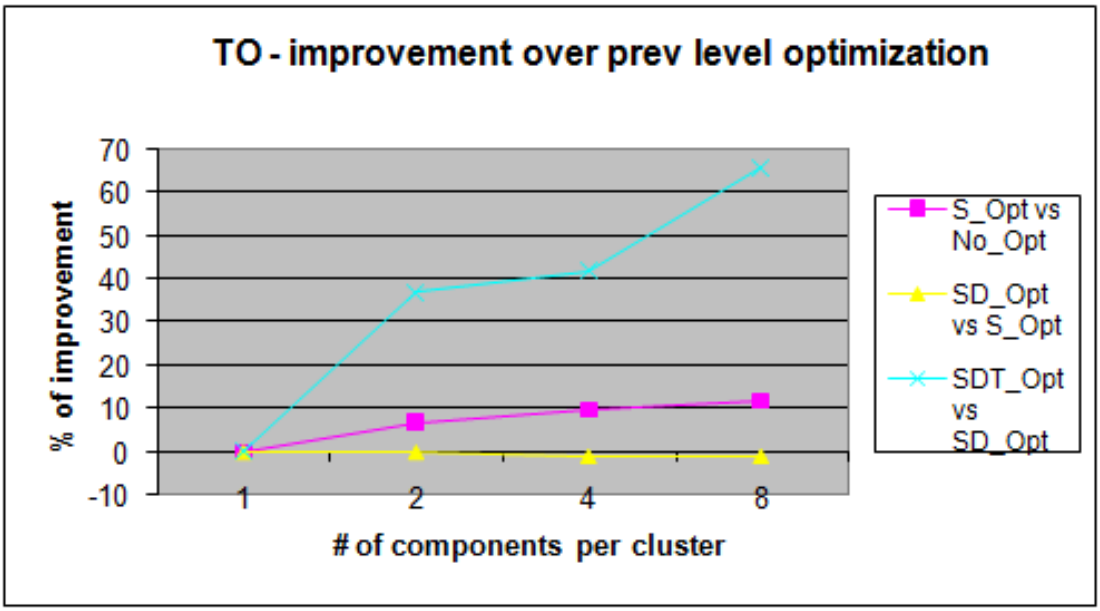

Figure C.31: *Case study - varying number of components per cluster - % improvement over previous level of optimization for total ordering service*

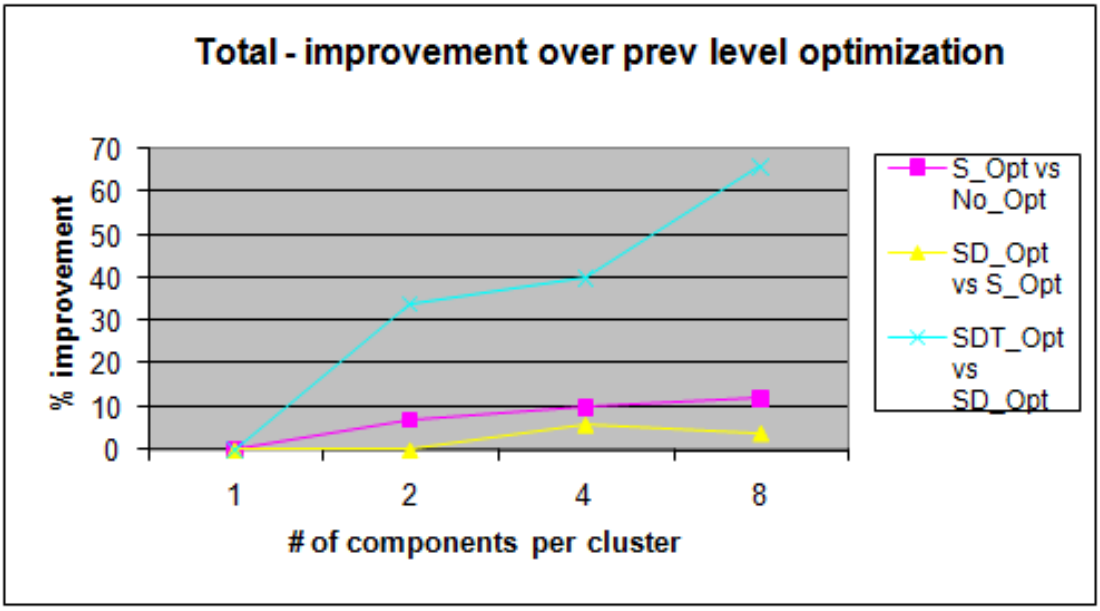

Figure C.32: *Case study - varying number of components per cluster - % improvement over previous level of optimization for total number of messages*

The results show the following trends:

For mutual exclusion service, as the number of components per cluster with local ordering information becomes larger, improvements due to static optimization are not significant and stay the same. Dynamic optimization is significant and increases with the increase of number of components per cluster. Physical topology based optimization further contributes to improvement in the number of messages and increases with the increase of number of components per cluster. For termination detection service, only dynamic optimization contributes to the improvement in the number of messages. As the number of components per cluster with local ordering information grows, the improvement does not change significantly. For total ordering service, as the number of components per cluster with local ordering information becomes larger, improvements due to static optimization are not significant and stay the same. Dynamic optimization is not a contributing factor. Physical topology based optimization is significant and increases as the number of components per cluster grows.

Varying number of clusters with ordering

In this section we will vary the number of clusters with ordering from 1 to 2 to 4 to 8. The number of players per cluster is four and remains the same. The number of clusters is eight and remains the same also. Players in each group with an ordering make bids in a round-robin fashion. Players in the groups without an ordering make bids in no particular order. Each player component is located on a separate physical machine. Application and physical topologies for each case are shown in Figures C.33 - C.37.

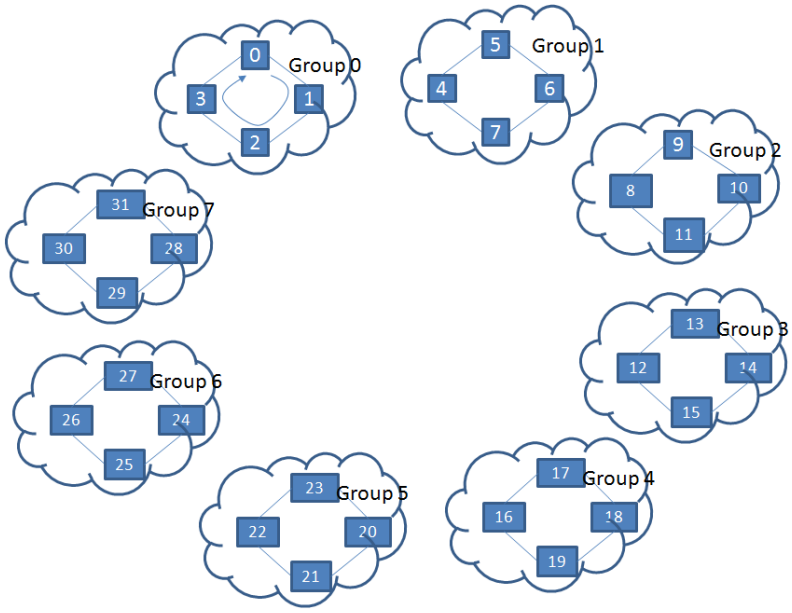

Figure C.33: *Case study - varying number of clusters with ordering - logical topology of application with 1 cluster with ordering*

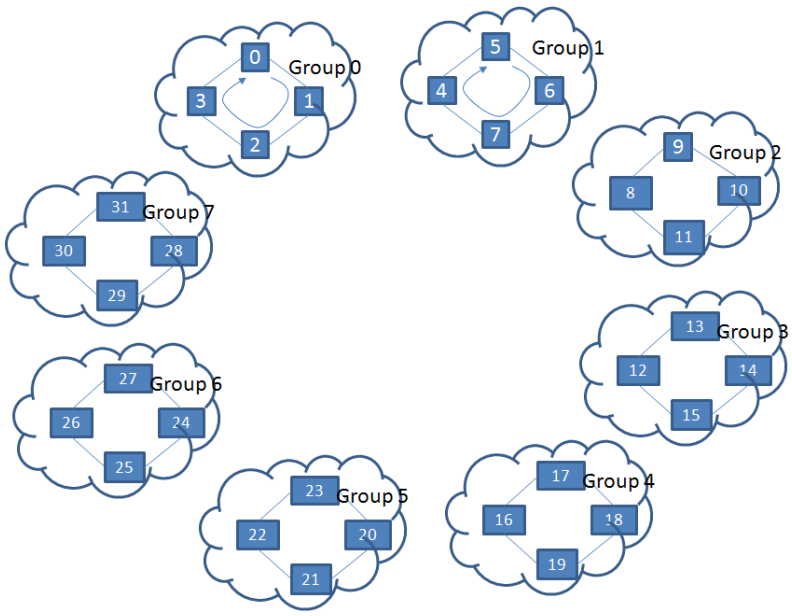

Figure C.34: *Case study - varying number of clusters with ordering - logical topology of application with 2 clusters with ordering*

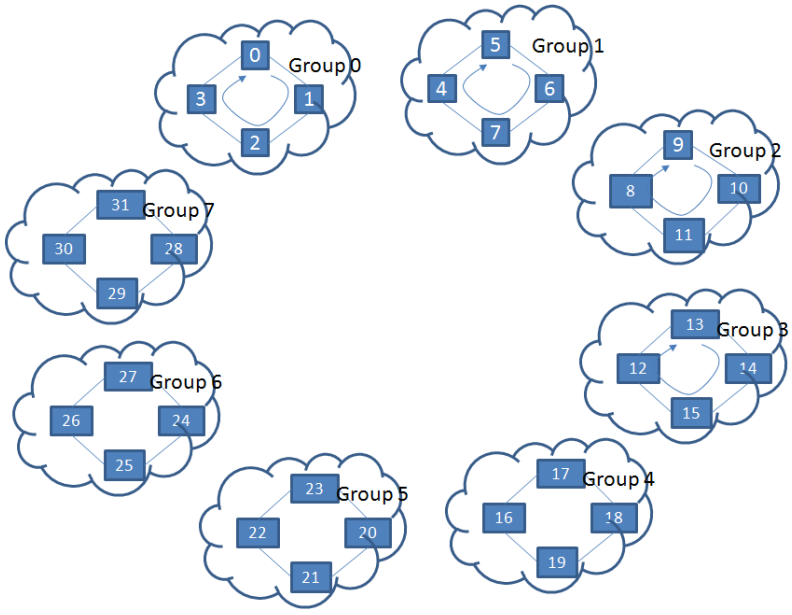

Figure C.35: *Case study - varying number of clusters with ordering - logical topology of application with 4 clusters with ordering*

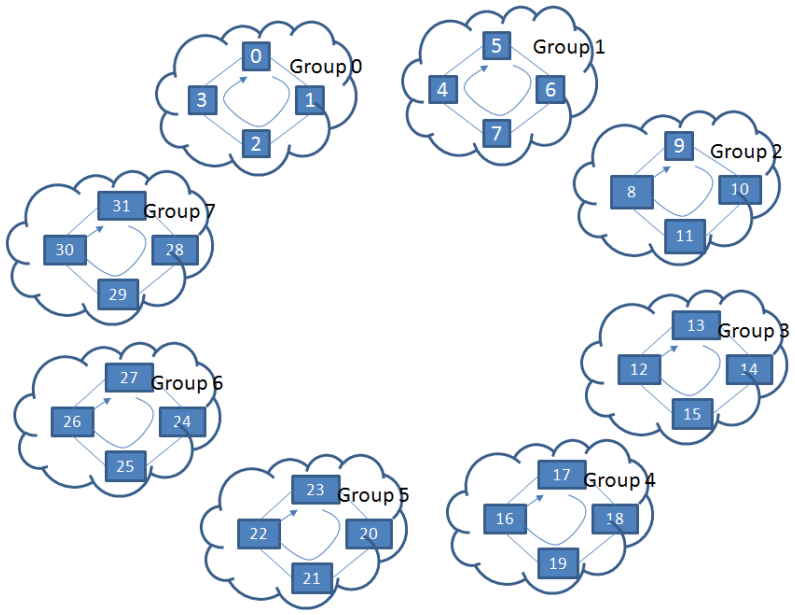

Figure C.36: *Case study - varying number of clusters with ordering - logical topology of application with 8 clusters with ordering*

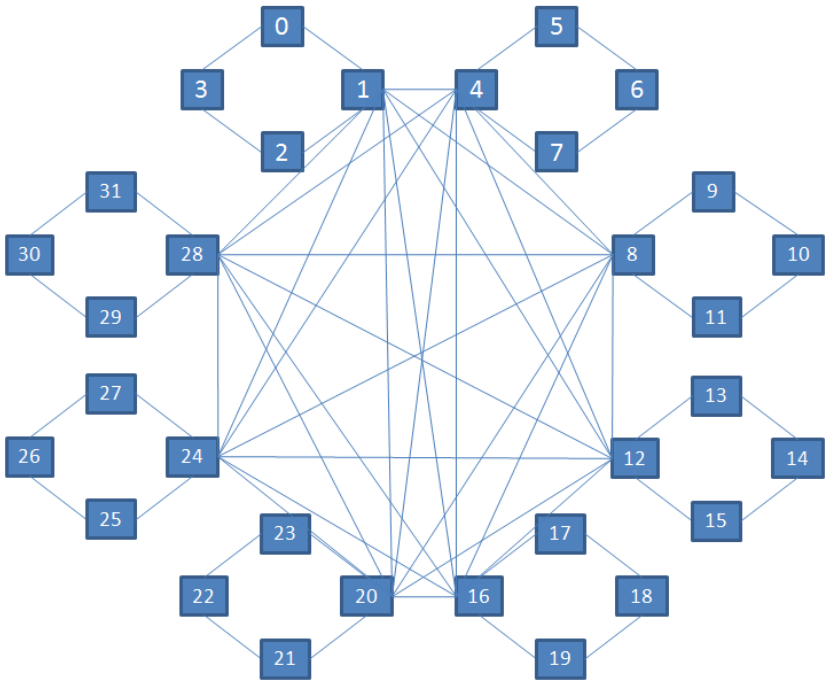

Figure C.37: *Case study - varying number of clusters - physical topology of application*

The results for the improvement over No_Opt case for studied services are shown in Figures C.38 - C.41.

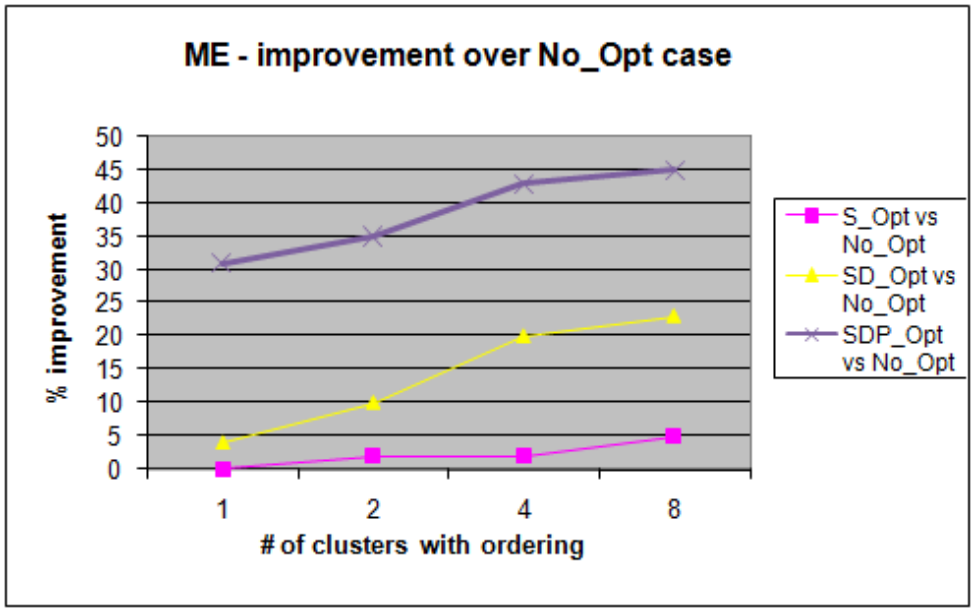

Figure C.38: *Case study - varying number of clusters with ordering - % improvement over No_Opt case for mutual exclusion service*

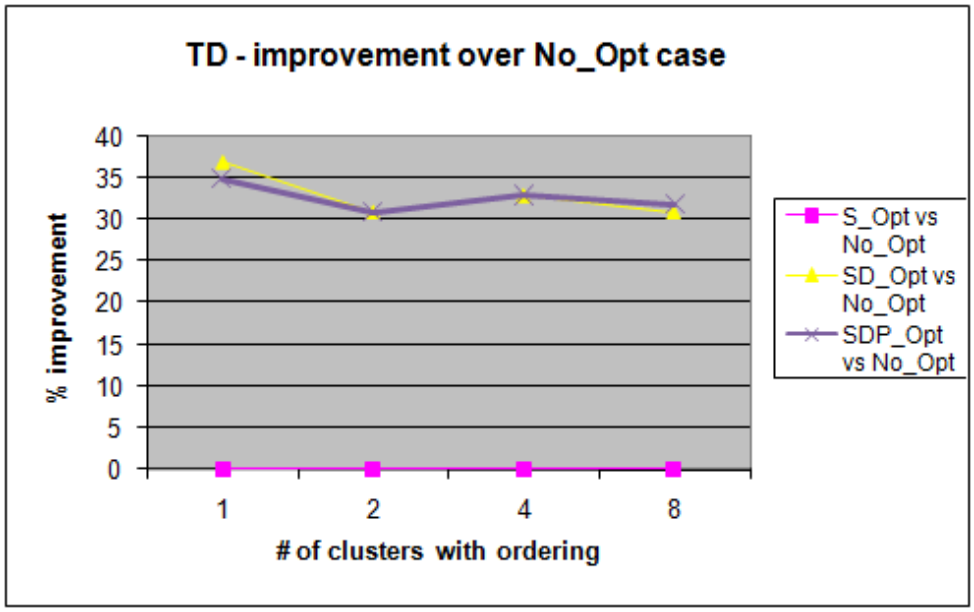

Figure C.39: *Case study - varying number of clusters with ordering - % improvement over No_Opt case for termination detection service*

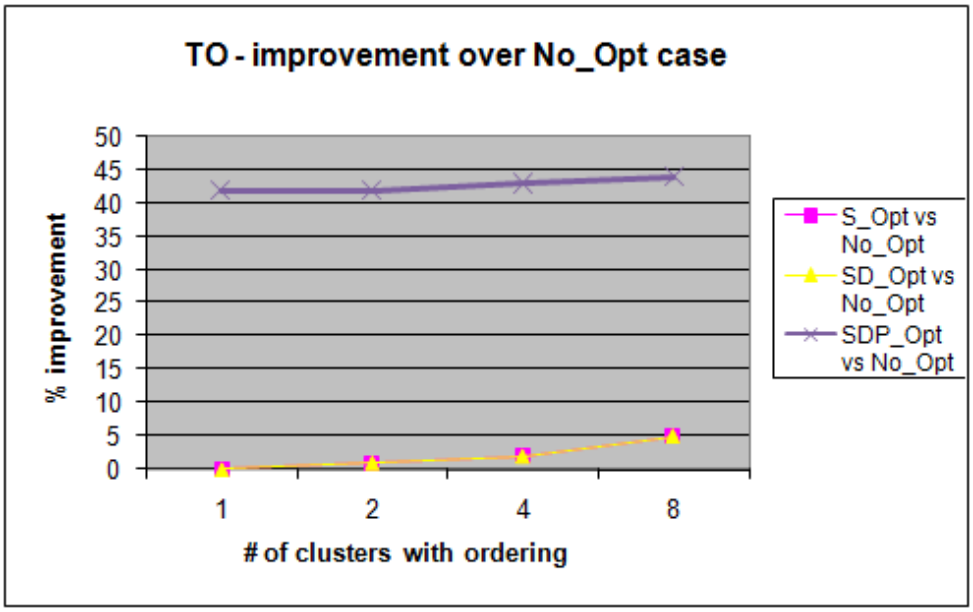

Figure C.40: *Case study - varying number of clusters with ordering - % improvement over No_Opt case for total ordering service*

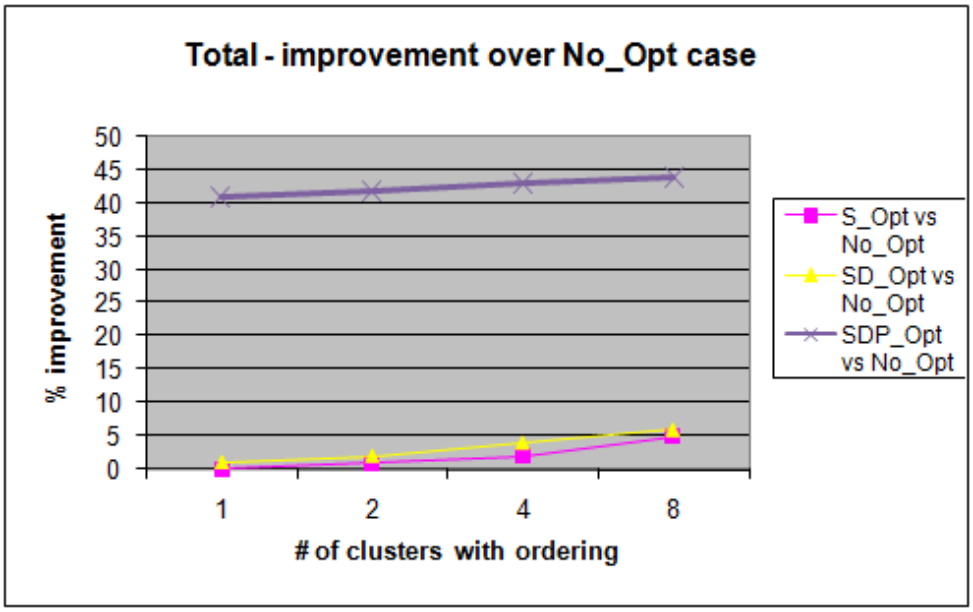

Figure C.41: *Case study - varying number of clusters with ordering - % improvement over No_Opt case for total number of messages*

The results for the improvement over previous level of optimization for studied services are shown in Figures C.42 - C.45.

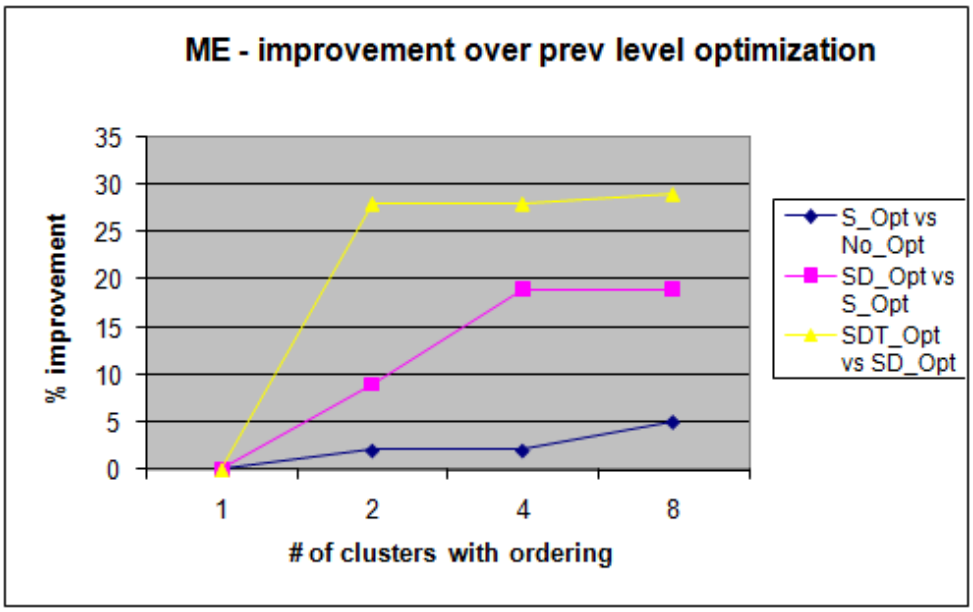

Figure C.42: *Case study - varying number of clusters with ordering - % improvement over previous level of optimization for mutual exclusion service*

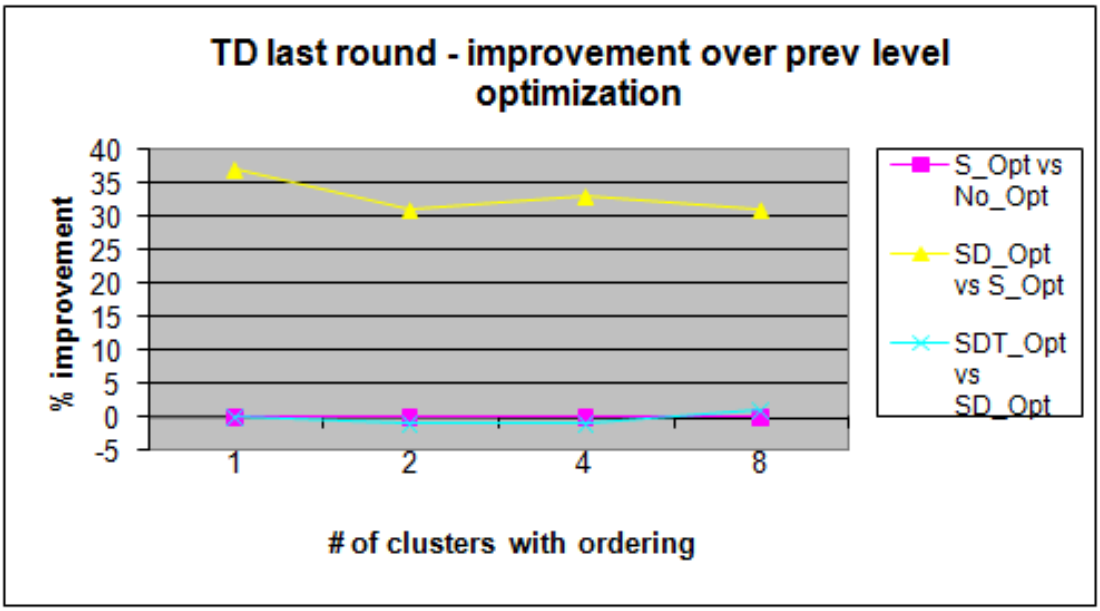

Figure C.43: *Case study - varying number of clusters with ordering - % improvement over previous level of optimization for termination detection service*

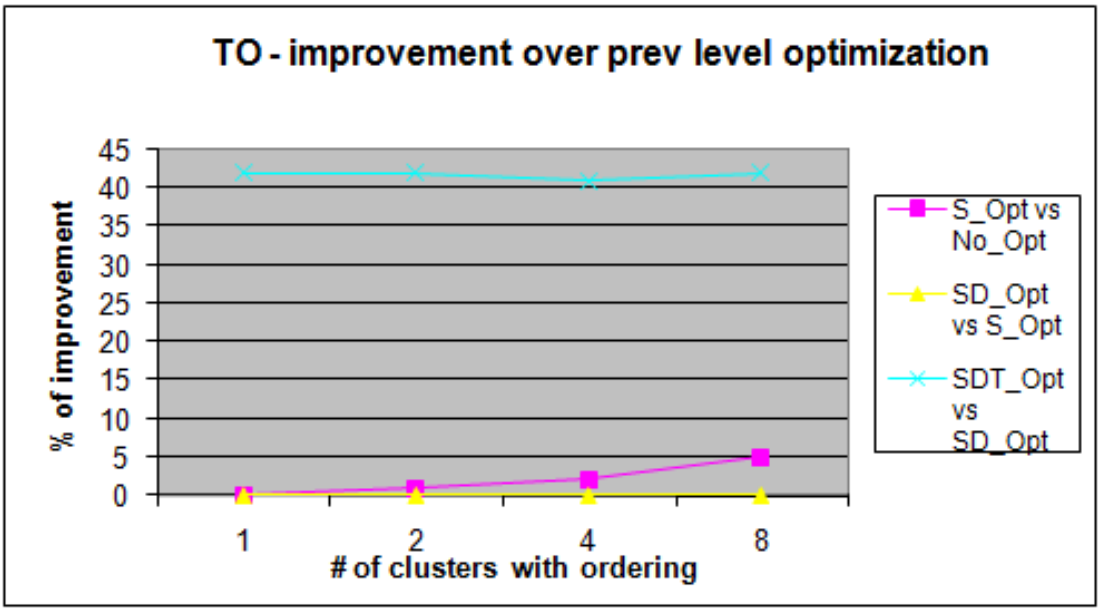

Figure C.44: *Case study - varying number of clusters with ordering - % improvement over previous level of optimization for total ordering service*

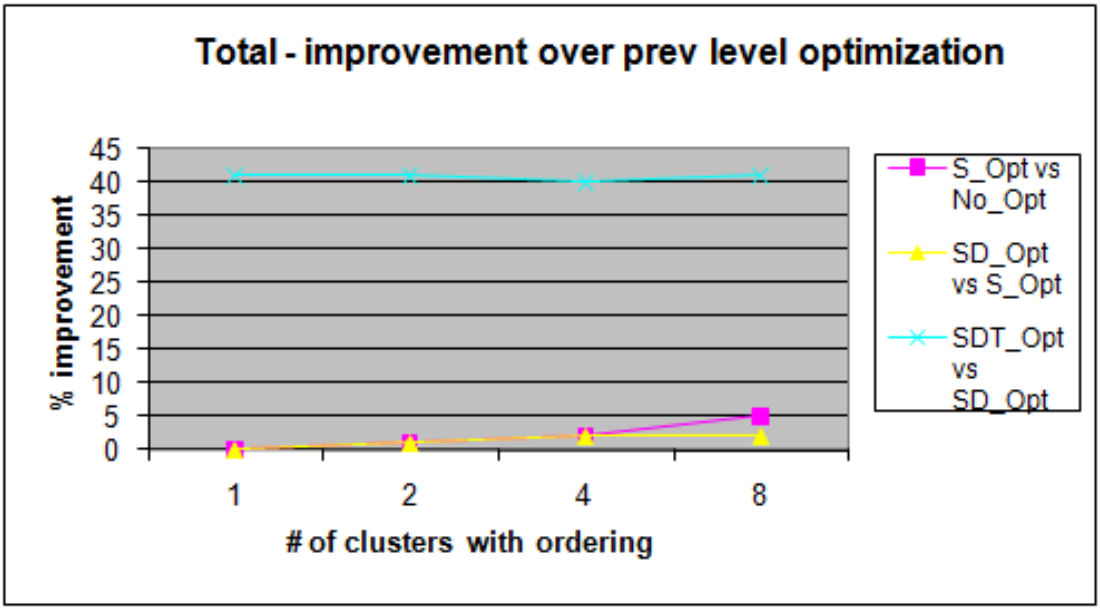

Figure C.45: *Case study - varying number of clusters with ordering - % improvement over previous level of optimization for total number of messages*

The results show the following trends:

For mutual exclusion service, as the number of clusters with local ordering information becomes larger, improvements due to static optimization are not significant. Dynamic optimization is significant. It increases with the increase of number of clusters with ordering. Physical topology based optimization further contributes to improvement in the number of messages and stays the same as the number of clusters with ordering grows. For termination detection service, only dynamic optimization contributes to the improvement in the number of messages. As the number of clusters with local ordering information grows, the improvement does not change. For total ordering service, as the number of clusters with local ordering information becomes larger, improvements due to static optimization are not significant. Dynamic optimization is not a contributing factor. Physical topology based optimization is significant and stays the same as the number of clusters with ordering grows.

Varying number of components per processor

In this section we will vary the number of components per processor from 1 to 2 to 4 to 8. The number of clusters is four and remains the same. The number of components is thirty-two and also remains the same. Players in each group make bids in a round-robin fashion. Application and physical topologies for each case are shown in Figures C.46 - C.50.

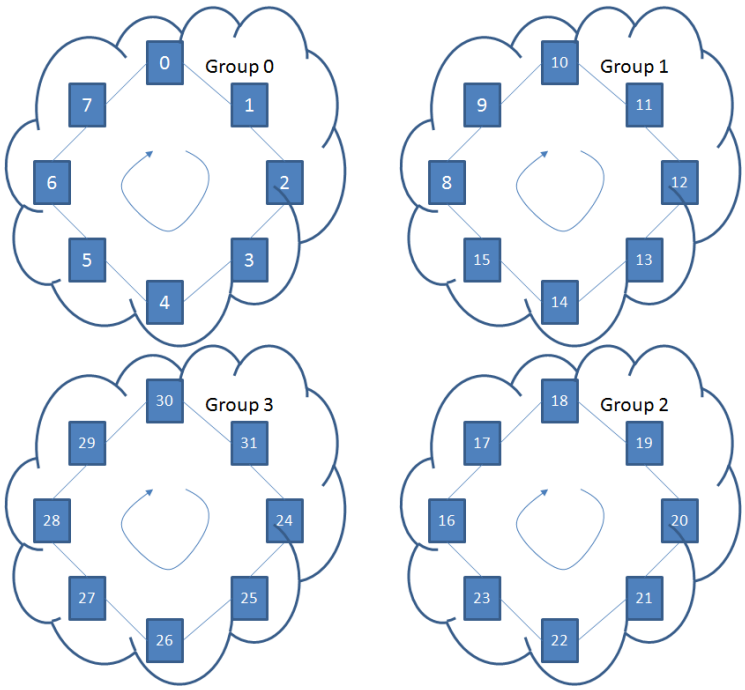

Figure C.46: *Case study - varying number of components per processor - logical topology of application with 8 components per cluster*

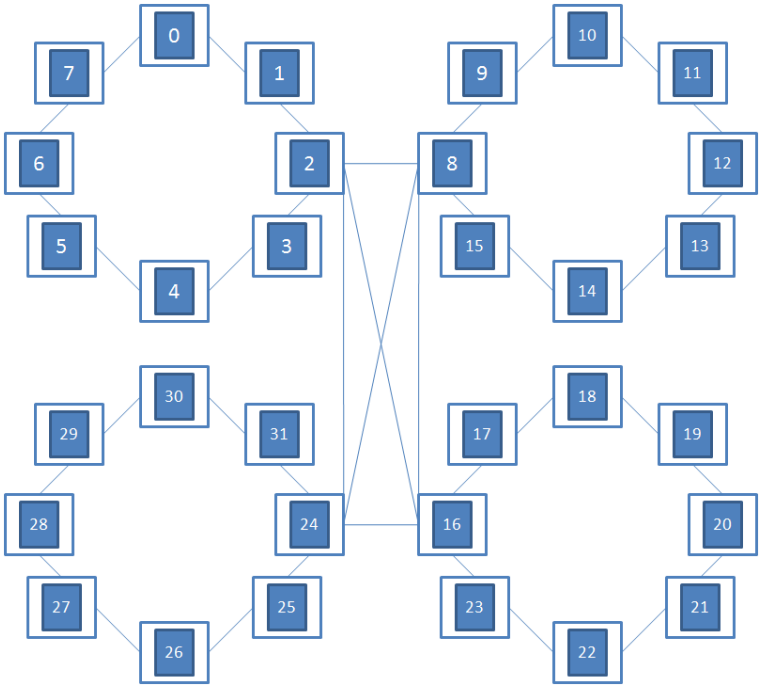

Figure C.47: *Case study - varying number of components per processor - physical topology of application with 1 component per processor*

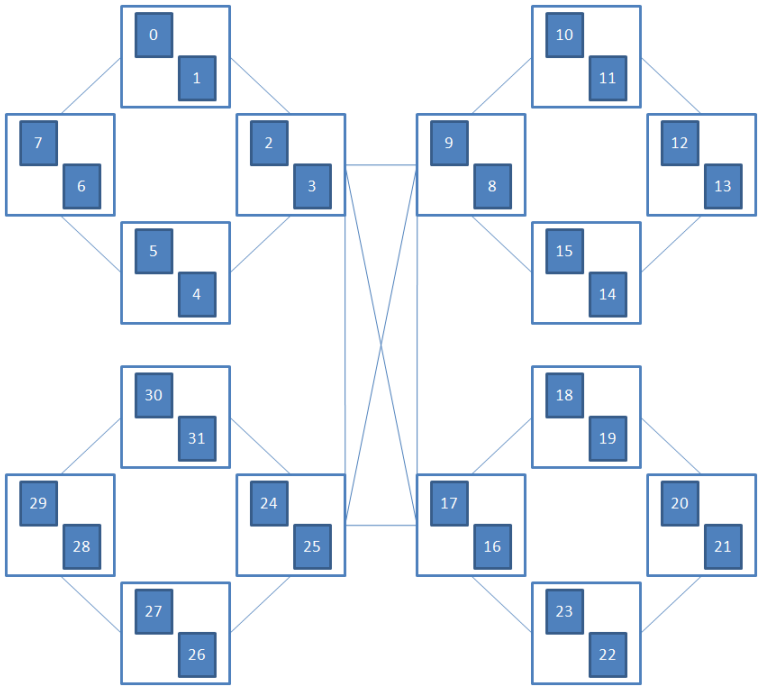

Figure C.48: *Case study - varying number of components per processor - physical topology of application with 2 components per processor*

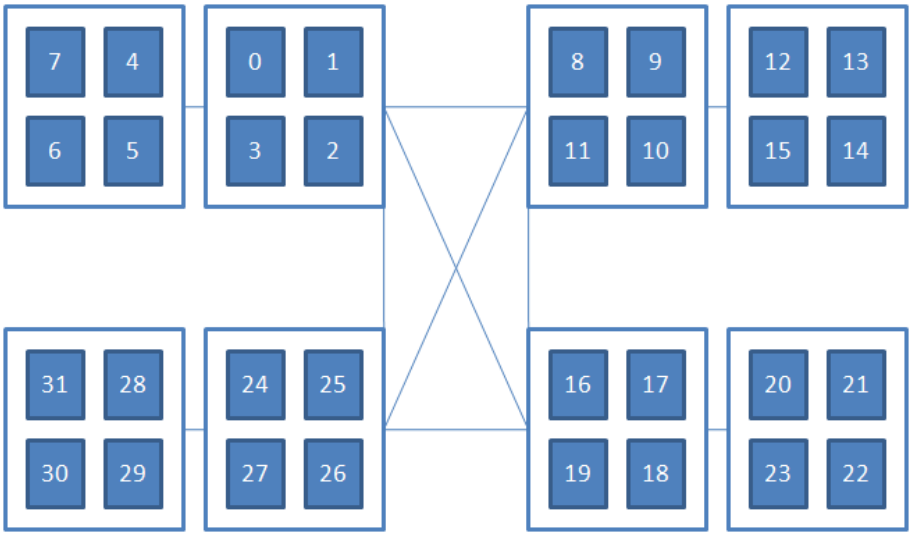

Figure C.49: *Case study - varying number of components per processor - physical topology of application with 4 components per processor*

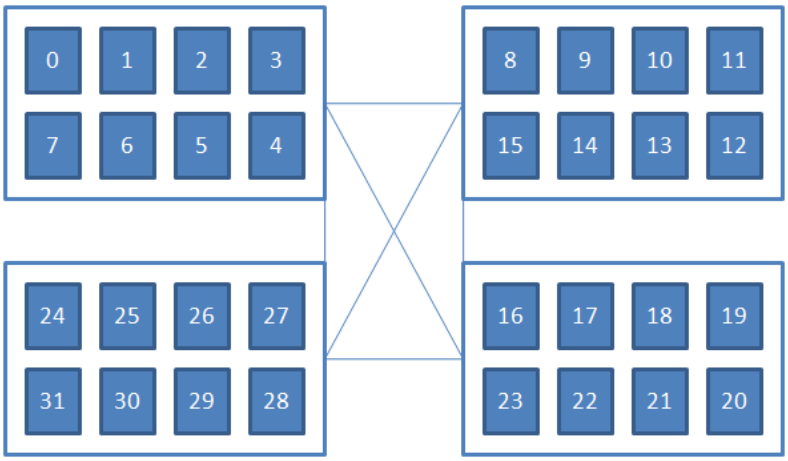

Figure C.50: *Case study - varying number of components per processor - physical topology of application with 8 components per processor*

The results for the improvement over No_Opt case for studied services are shown in Figures C.51 - C.54.

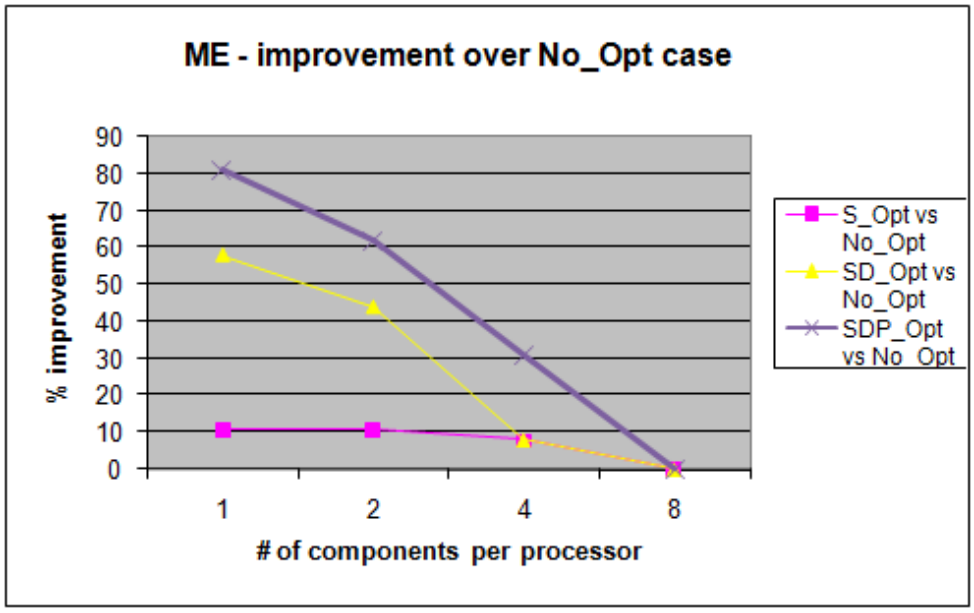

Figure C.51: *Case study - varying number of components per processor - % improvement over No_Opt case for mutual exclusion service*

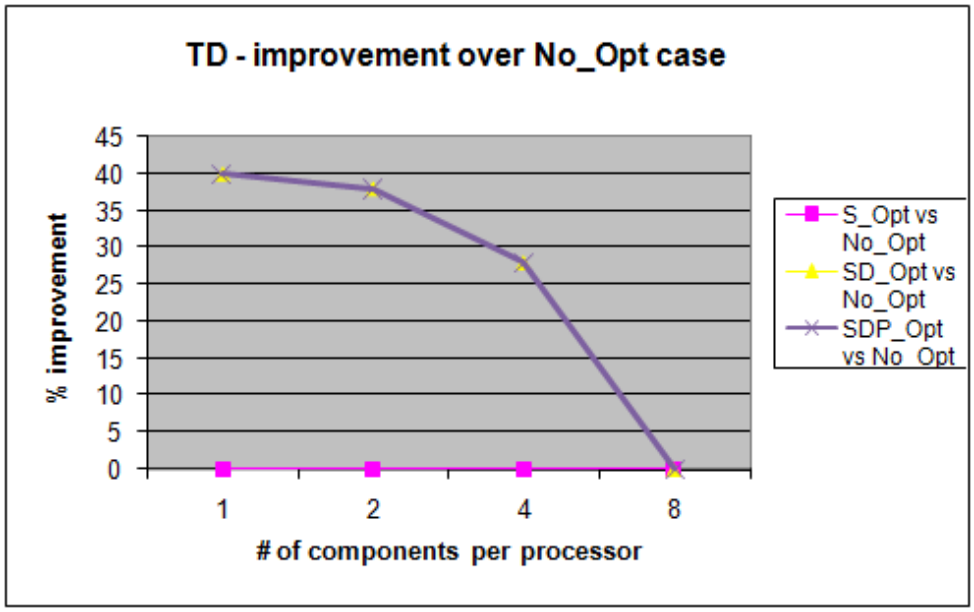

Figure C.52: *Case study - varying number of components per processor - % improvement over No_Opt case for termination detection service*

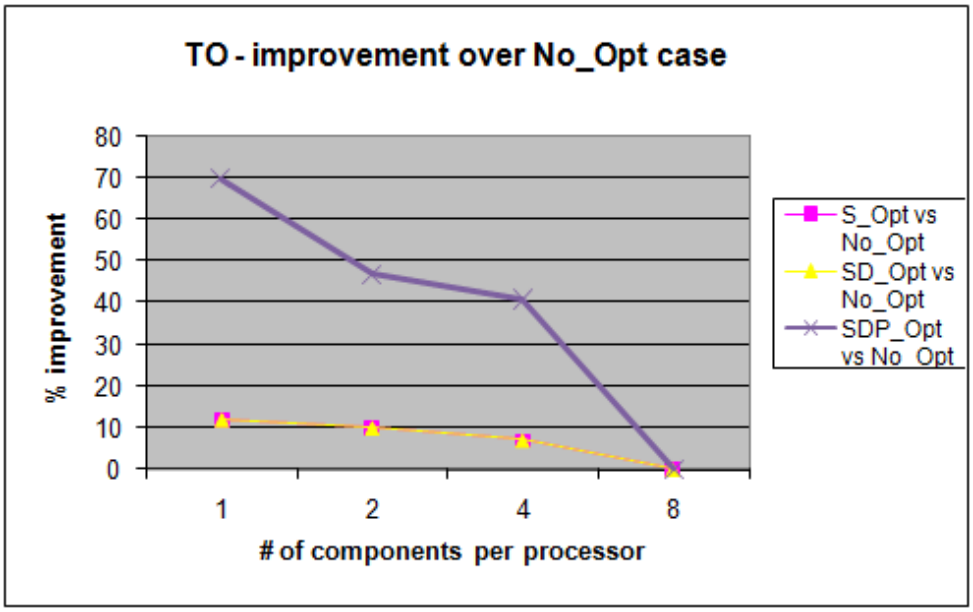

Figure C.53: *Case study - varying number of components per processor - % improvement over No_Opt case for total ordering service*

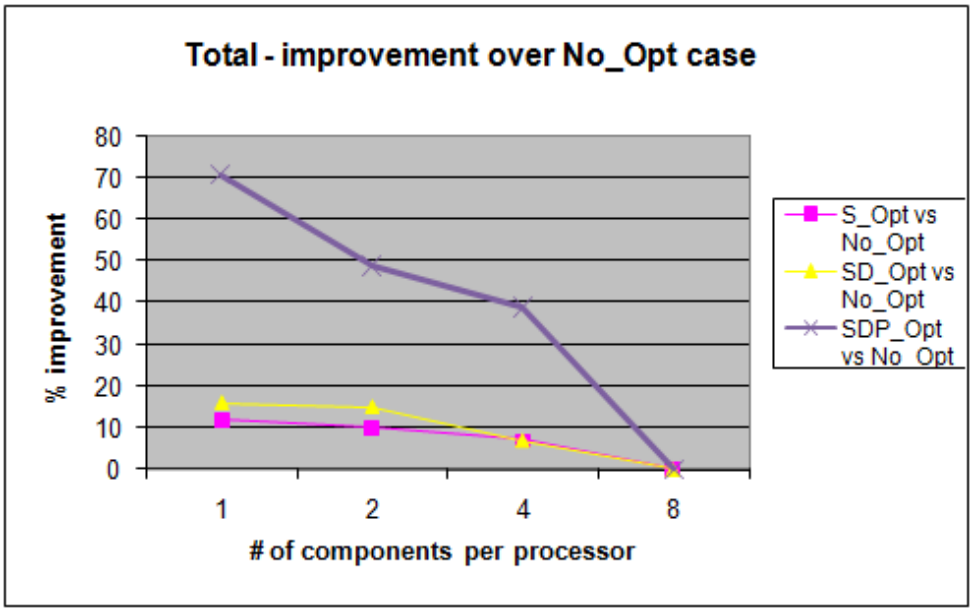

Figure C.54: *Case study - varying number of components per processor - % improvement over No_Opt case for total number of messages*

The results for the improvement over previous level of optimization for studied services are shown in Figures C.29 - C.32.

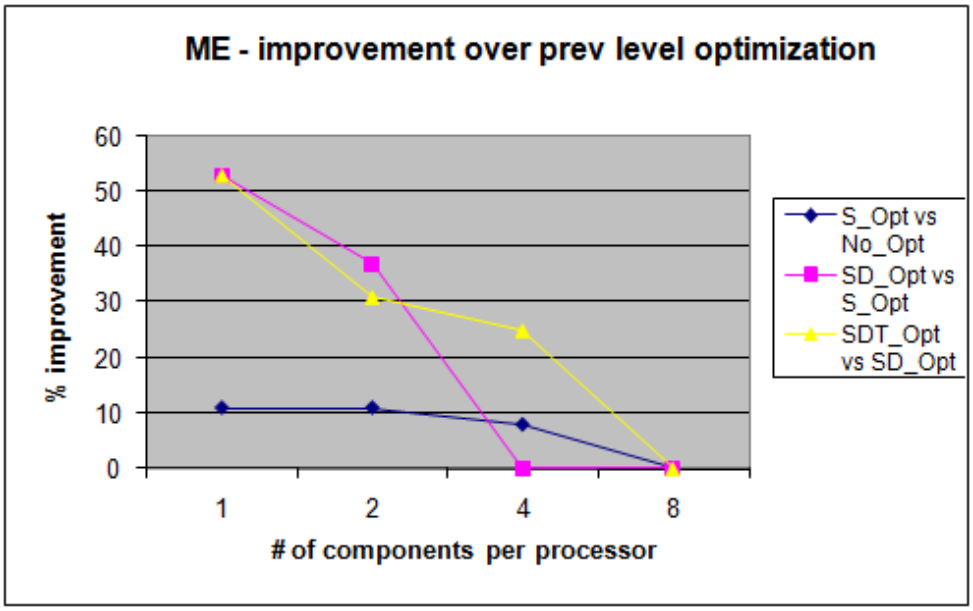

Figure C.55: *Case study - varying number of components per processor - % improvement over previous level of optimization for mutual exclusion service*

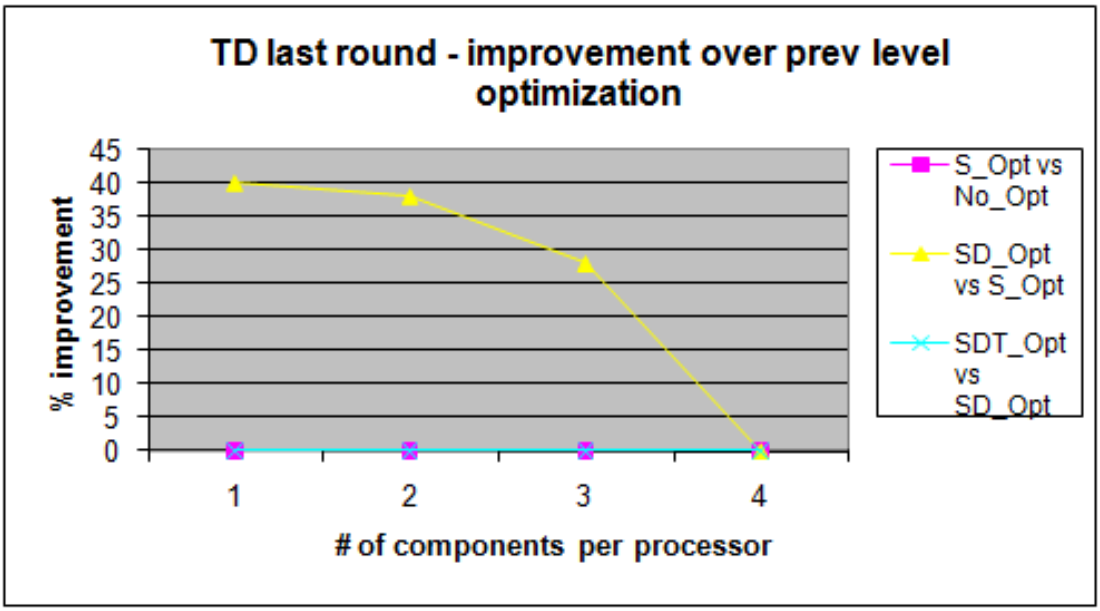

Figure C.56: *Case study - varying number of components per processor - % improvement over previous level of optimization for termination detection service*

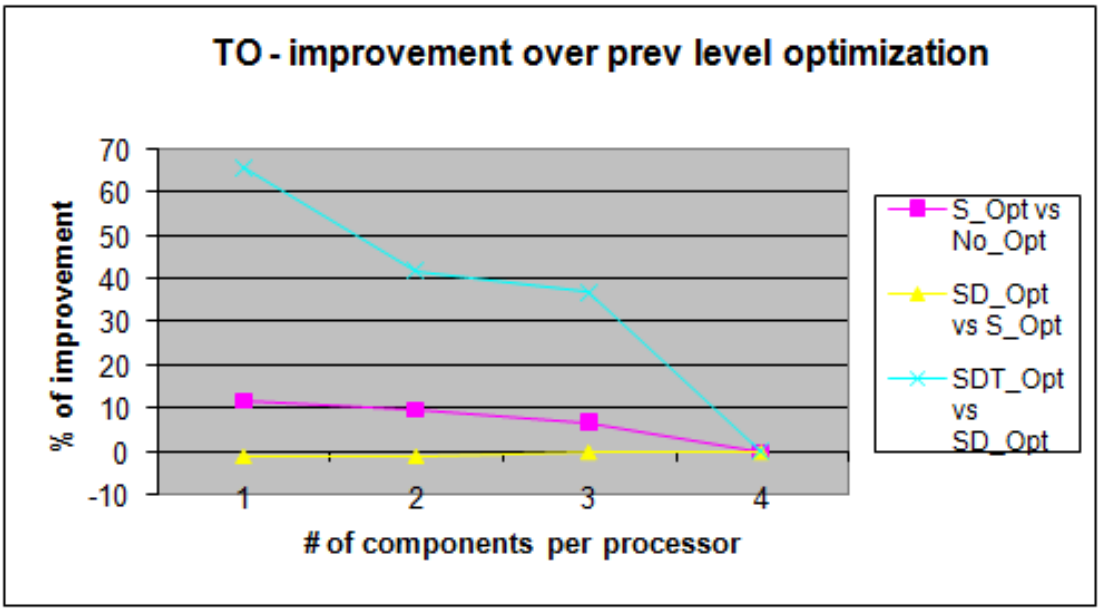

Figure C.57: *Case study - varying number of components per processor - % improvement over previous level of optimization for total ordering*

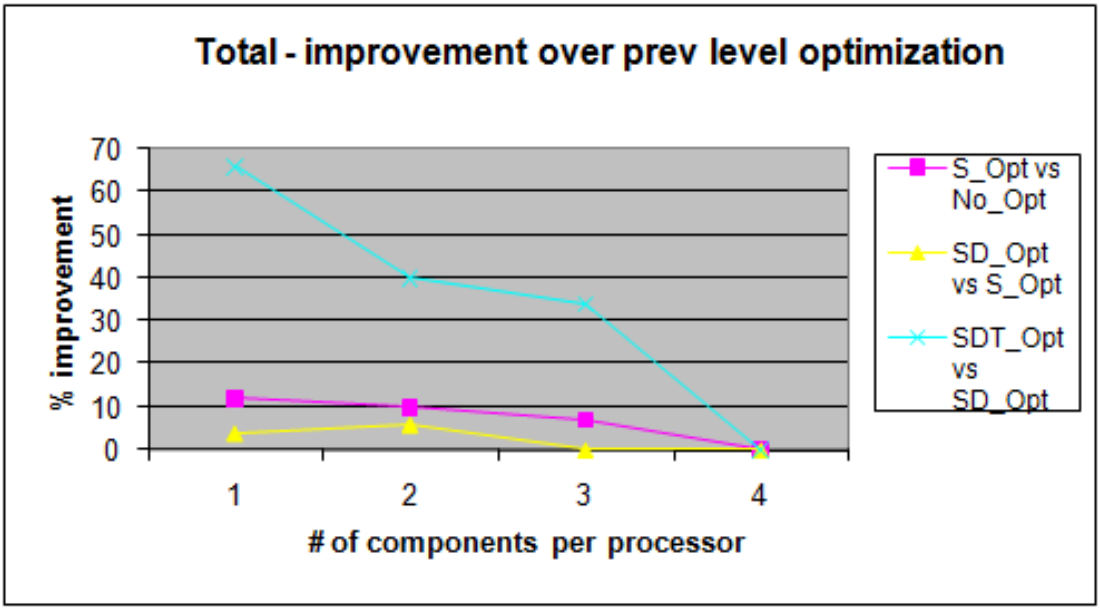

Figure C.58: *Case study - varying number of components per processor - % improvement over previous level of optimization for total number of messages*

The results show the following trends:

For all services, as the number of components per processor becomes larger, improvements due to all optimizations become less significant and approach zero. The more components we put on a processor, the fewer messages we need to send from one processor to another. If all components are placed on the same processor, no network messages will be needed.

Summary

The results of case studies allow us to make several observations. InDiGO framework utilizes ordering information on events for possible optimizations and thus best suited for applications whose components issue events in some order.

The applications that have higher number of localized clusters of components with ordering show lower relative optimization level than those with fewer number of localized clusters of components with ordering. But, as the number of components in a localized cluster grows, so does the level of optimization.

For physical topology optimization, the sparser the physical topology that connects components, the better physical topology optimization. Also, the fewer application components are mapped on a single physical processor, the better the optimization achieved by utilizing our framework.

In terms of algorithms, the algorithms that involve communication with higher number of components tend to be optimized better than those which involve communication with only a few components. For example, the algorithms that involve communication with all components could be optimized better than those that involve communication with neighbours only.

Appendix D

Examples of other distributed algorithms and their customization

In this section, we will describe other distributed algorithms that we looked at. We will also present their customized versions. As could be seen, the trasformation to a customized version with interaction sets is simple. All algorithms that we present here have been implemented on J-Sim simulator.

D.1 Token based mutual exclusion distributed algorithm for arbitrary topology

Here we present a token based mutual exclusion distributed algorithm for an arbitrary topology. The token is requested by process P_i using a timestamped *request* message sent to all processes. P_i does not know which process has the token. Along with the token we pass a vector *vector* that contains a timestamp of the last visit the token made to each of the Pk processes. Once process P_j, which holds the token, exits its critical section, it looks for the first process P_k (k in the order j + 1, . . . , n, 1, . . . , j - 1) such that the timestamp of the last request from P_k is greater than the timestamp stored in the vector during its last visit to P_k. The timestamps of requests received by a process P_k are stored in its local array variable *requests*. The algorithm is presented in Figure D.1 below. This algorithm is due to Ricart and Agrawala[25].

Code for process P_i:

local variables:

msn: 1 .. ∞ // initialized to 1
bool have_token ← false
bool request_sent ← false
bool in_cs ← false
vector: array[1..n] of 0 .. ∞ // all elements initialized to 0
requests: array[1..n] of 0 .. ∞ // all elements initialized to 0

```
:: (want_to_enter_CS ∧ have_token)
    in_CS ← true
    want_to_enter_CS ← false
    want_to_exit_CS ← false
    request_sent ← false
:: (want_to_enter_CS ∧ !have_token ∧ !request_sent)
    msn ← msn + 1
    send(request, msn, i)
    request_sent ← true
:: receive(request, k, j)
    requests[j] ← max(requests[j], k)
    if (have_token ∧ !in_CS)
        for j = i + 1 to n, 1 to i - 1
            if ((requests[j] > vector[j]) ∧ have_token)
                have_token ← false
                send(token, vector, i) to j
:: (have_token ∧ in_CS ∧ want_to_exit_CS)
    vector[i] ← msn
    in_CS ← false
    for j = i + 1 to n, 1 to i - 1
        if ((requests[j] > vector[j]) ∧ have_token)
            have_token ← false
            send(token, vector, i) to j
:: receive(token, v, j)
    have_token ← true
    for k = 1 to n
        vector[k] ← max(vector[k], v[k])
```

Figure D.1: *Token based mutual exclusion distributed algorithm for arbitrary topology*

Customized version.

The following interaction sets are needed for a customized version:

- SRT - "send request to" set
- RRF - "request received from" set - reverse of SRT

The customized version is shown in Figure D.2 below:

Code for process P_i:

local variables:

msn: 1 .. ∞ // initialized to 1
bool have_token ← false
bool request_sent ← false
bool in_cs ← false
vector: array[1..n] of 0 .. ∞ // all elements initialized to 0
requests: array[1..n] of 0 .. ∞ // all elements initialized to 0

```
:: (want_to_enter_CS ∧ have_token)
    in_CS ← true
    want_to_enter_CS ← false
    want_to_exit_CS ← false
    request_sent ← false
:: (want_to_enter_CS ∧ !have_token ∧ !request_sent)
    msn ← msn + 1
    send(request, msn, i) to SRT
    request_sent ← true
:: receive(request, k, j)
    requests[j] ← max(requests[j], k)
    if (have_token ∧ !in_CS)
        for j = i + 1 to n, 1 to i - 1
            if (j ∈ RRF)
                if ((requests[j] > vector[j]) ∧ have_token)
                    have_token ← false
                    send(token, vector, i) to j
:: (have_token ∧ in_CS ∧ want_to_exit_CS)
    vector[i] ← msn
    in_CS ← false
    for j = i + 1 to n, 1 to i - 1
        if (j ∈ RRF)
            if ((requests[j] > vector[j]) ∧ have_token)
                have_token ← false
                send(token, vector, i) to j
:: receive(token, v, j)
    have_token ← true
    for k = 1 to n
        vector[k] ← max(vector[k], v[k])
```

Figure D.2: *Customized version of token based mutual exclusion distributed algorithm for arbitrary topology*

D.2 Distributed termination detection algorithm for arbitrary topology

Here we present a distributed termination detection algorithm for arbitrary topology. If a process Q receives the marker for the first time, it considers the process that sent that marker as its predecessor. Then, it returns to its predecessor either a *DONE* message or a *CONTINUE* message. A *DONE* message is returned only when the following two conditions are met:

1 All of Q's successors have returned a *DONE* message.

2 Q has not received any message between the point it recorded its state, and the point it had received the marker along each of its incoming channels.

In all other cases Q sends a *CONTINUE* message to its predecessor.

Eventually, the original initiator, say process P, will receive either a *CONITNUE* message, or only *DONE* messages from its successors. When only *DONE* messages are received, it is known that no regular messages are in transit, and thus the computation has terminateed. Otherwise, process P initiates another round, and continues to do so until onlly *DONE* messages are eventually returned.

The algorithm described above is presented in Figures D.3 and D.4. This algorithm was adopted from discussion about using a snapshot to solve termination detection problem in[53].

Code for process P_k that initiates termination detection:

```
init(){
    initiated ← false
    replies ← 0
}

init()
sn ← 1 // sequence number
:: (!initiated ∧ state = passive)
    send(marker, k, neighbours, sn)
    initiated ← true
:: receive(CONTINUE, source, dest, seq)
    if (seq = sn)
        init()
        sn ← sn + 1
:: receive(DONE, source, dest, seq)
    if (seq = sn)
        replies ← replies + 1
        if (replies = neighbours.size)
            termination detected
:: receive(app)
    init()
    sn ← sn + 1
    state ← active
```

Figure D.3: *Distributed termination detection algorithm for any arbitrary topology - initiating process*

Code for process $P_i,\ i \neq k$:

```
init(){
    marker_received ← false
    predecessor ← -1
    replies ← 0
}
init()
sn ← 0
:: receive(marker, source, dest, seq)
    if (seq > sn)
        sn ← seq; init()
        if (state = active)
            send(CONTINUE, i, source, sn)
        else // state is passive
            marker_received ← true
            predecessor ← source
            if (neighbours.size > 1)
                send(marker, i, neighbours - source, sn)
            else // only one neighbour - source
                send(DONE, i, source, sn)
    else if (seq = sn)
        send(ALREADY, i, source, sn)
:: receive(CONTINUE, source, dest, seq)
    if (seq ⩾ sn ∧ marker_received)
        send(CONTINUE, i, predecessor, seq)
        init()
:: receive(ALREADY, source, dest, seq)
    if (seq ⩾ sn ∧ marker_received)
        replies ← replies + 1
        if (replies = neighbours.size - 1) // -1 - account for predecessor
            send(DONE, i, predecessor, seq)
:: receive(DONE, source, dest, seq)
    if (seq ⩾ sn ∧ marker_received)
        replies ← replies + 1
        if (replies = neighbours.size - 1)
            send(DONE, i, predecessor, seq)
:: receive(app)
    state ← active
    if (predecessor ≥ 0)
        send(CONTINUE, i, predecessor, sn)
    init()
```

Figure D.4: *Distributed termination detection algorithm for any arbitrary topology - not initiating process*

Customized version.

The following interaction sets are needed for a customized version:

- SMT - "send marker to" set - all neighbors - "not important" neighbors
- WRF = SMT - "wait response from" set - all neighbors - "not important" neighbors

The customized version is shown in Figure D.5 and Figure D.6 below.

Code for process P_k that initiates termination detection:

```
init(){
    initiated ← false
    replies ← 0
}

init()
sn ← 1 // sequence number
:: (!initiated ∧ state = passive)
    send(marker, k, sn) to SMT
    initiated ← true
:: receive(CONTINUE, source, dest, seq)
    if (seq = sn)
        init()
        sn ← sn + 1
:: receive(DONE, source, dest, seq)
    if (seq = sn)
        replies ← replies + 1
        if (replies = WRF.size)
            termination detected
:: receive(app)
    init()
    sn ← sn + 1
    state ← active
```

Figure D.5: *Customized version of distributed termination detection algorithm for any arbitrary topology - initiating process*

Code for process $P_i,\ i \neq k$:

```
init(){
    marker_received ← false
    predecessor ← -1
    replies ← 0
}
init()
sn ← 0
:: receive(marker, source, dest, seq)
    if (seq > sn)
        sn ← seq; init()
        if (state = active)
            send(CONTINUE, i, source, sn)
        else // state is passive
            marker_received ← true
            predecessor ← source
            if (SMT.size > 1)
                send(marker, i, sn) to SMT
            else // only one neighbour - source
                send(DONE, i, source, sn)
    else if (seq = sn)
        send(ALREADY, i, source, sn)
:: receive(CONTINUE, source, dest, seq)
    if (seq ⩾ sn ∧ marker_received)
        send(CONTINUE, i, predecessor, seq)
        init()
:: receive(ALREADY, source, dest, seq)
    if (seq ⩾ sn ∧ marker_received)
        replies ← replies + 1
        if (replies = WRF.size)
            send(DONE, i, predecessor, seq)
:: receive(DONE, source, dest, seq)
    if (seq ⩾ sn ∧ marker_received)
        replies ← replies + 1
        if (replies = WRF.size)
            send(DONE, i, predecessor, seq)
:: receive(app)
    state ← active
    if (predecessor ≥ 0)
        send(CONTINUE, i, predecessor, sn)
    init()
```

Figure D.6: *Customized version of distributed termination detection algorithm for any arbitrary topology - not initiating process*

D.3 Distributed termination detection algorithm for a star topology

Next we present a distributed termination detection algorithm for a star topology. We assume that all processes are arranged in a star topology with one special process P_0 in the center of the star and all the rest of the processes directly connected to P_0 but not to any other process.

To detect termination, when the app component signals to the termination detection component that the state of P_0 becomes *passive*, termination detection component sends a *marker* message to all other processes. On receiving a *marker* message, if the process is *passive*, it sends *DONE* message back to P_0. Otherwise, it sends *CONTINUE* message to P_0. If P_0 receives *DONE* relplies from all processes and remains *passive* since sending the *marker* message out, it concludes that termination did happen. Otherwise, termination did not happen. We present two versions for P_0. In version 1, if termination is not detected, then P_0 responds with a NotTerminated message to the app component, and the app component needs to initiate another round of termination detection (see Figure D.7). In version 2, after detecting that termination did not happen, P_0 waits until all the processes respond, and initiates termination detection algorithm anew if its state is *passive* (see Figure D.8).

The number of processes is denoted by n. If APP component receives a NOTTERMINATED message and still wants to detect termination, it needs to initiate TEDET algorithm again by sending START message to TEDET component.

Next, we present algorithm for $P_i,\ i \neq 0$ shown in Figure D.9.

Code for process P_0 (version 1):

```
state ← active
initiated ← false
pcolor ← black
number_of_replies ← 0
:: receive(START)
    state ← passive
:: (!initiated && state = passive)
    pcolor ← white
    send(marker, all)
    initiated ← true
    number_of_replies ← 0
:: receive(app)
    pcolor ← black
:: receive(DONE)
    number_of_replies ← number_of_replies + 1
    if (number_of_replies = n - 1)
        if (pcolor = white)
            send(TERMINATED, APP) and halt
        else
            send(NOTTERMINATED, APP)
        state ← active
        initiated ← false
:: receive(CONTINUE)
    number_of_replies ← number_of_replies + 1
    pcolor ← black
    if (number_of_replies = n - 1)
        send(NOTTERMINATED, APP)
        state ← active
        initiated ← false
```

Figure D.7: *Distributed termination detection algorithm for a star topology (process P_0) - version 1*

Code for process P_0 (version 2):

```
state ← active
initiated ← false
continued ← false
pcolor ← black
number_of_replies ← 0
:: receive(START)
    state ← passive
:: ((!initiated || continued) && state = passive)
    pcolor ← white
    send(marker, all)
    initiated ← true
    number_of_replies ← 0
:: receive(app)
    pcolor ← black
:: receive(DONE)
    number_of_replies ← number_of_replies + 1
    if (number_of_replies = n - 1)
        if (pcolor = white)
            send(TERMINATED, APP) and halt
        else
            state ← active
:: receive(CONTINUE)
    number_of_replies ← number_of_replies + 1
    pcolor ← black
    if (number_of_replies = n - 1)
        state ← active
:: (initiated && state = active)
    continued ← true
    send(ISPASSIVE, APP)
:: receive(PASSIVE)
    state ← passive
:: receive(ACTIVE)
    skip
```

Figure D.8: *Distributed termination detection algorithm for a star topology (process P_0) - version 2*

Code for process $P_i,\ i \neq 0$:

```
:: receive(marker)
    send(ISPASSIVE, APP)
:: receive(PASSIVE)
    send(DONE, 0)
:: receive(ACTIVE)
    send(CONTINUE, 0)
```

Figure D.9: *Distributed termination detection algorithm for a star topology (process $P_i,\ i \neq 0$)*

Customized version.

The following interaction sets are needed for a customized version:

- SMT - "send marker to" set - all neighbors - "not important" neighbors
- WRF = SMT - "wait response from" set - all neighbors - "not important" neighbors

The customized versions are shown in Figures D.10 and D.11 below.

Code for process P_0 (version 1):

```
state ← active
initiated ← false
pcolor ← black
number_of_replies ← 0
:: receive(START)
    state ← passive
:: (!initiated && state = passive)
    pcolor ← white
    send(marker) to SMT
    initiated ← true
    number_of_replies ← 0
:: receive(app)
    pcolor ← black
:: receive(DONE)
    number_of_replies ← number_of_replies + 1
    if (number_of_replies = WRF.size)
        if (pcolor = white)
            send(TERMINATED, APP) and halt
        else
            send(NOTTERMINATED, APP)
        state ← active
        initiated ← false
:: receive(CONTINUE)
    number_of_replies ← number_of_replies + 1
    pcolor ← black
    if (number_of_replies = WRF.size)
        send(NOTTERMINATED, APP)
        state ← active
        initiated ← false
```

Figure D.10: *Customized version of distributed termination detection algorithm for a star topology (process P_0) - version 1*

Code for process P_0 (version 2):

```
state ← active
initiated ← false
continued ← false
pcolor ← black
number_of_replies ← 0
:: receive(START)
    state ← passive
:: ((!initiated || continued) && state = passive)
    pcolor ← white
    send(marker) to SMT
    initiated ← true
    number_of_replies ← 0
:: receive(app)
    pcolor ← black
:: receive(DONE)
    number_of_replies ← number_of_replies + 1
    if (number_of_replies = WRF.size)
        if (pcolor = white)
            send(TERMINATED, APP) and halt
        else
            state ← active
:: receive(CONTINUE)
    number_of_replies ← number_of_replies + 1
    pcolor ← black
    if (number_of_replies = WRF.size)
        state ← active
:: (initiated && state = active)
    continued ← true
    send(ISPASSIVE, APP)
:: receive(PASSIVE)
    state ← passive
:: receive(ACTIVE)
    skip
```

Figure D.11: *Customized version of distributed termination detection algorithm for a star topology (process P_0) - version 2*

D.4 The total order algorithm that uses one process as a sequencer

Next we present the total order algorithm that uses one process as a sequencer. We can designate one processor to be a sequencer. In that case, every processor that wishes to send a message, sends that message to the sequencer. Upon receiving a message, the sequencer assigns a sequence number to the message and forwards it to the destination process. It is easy to see that the sequencer will maintain the total order of messages based on sequence number.

A message carries with it a message id. When a process receives a message, it stores it in its local priority queue. The sequencer process assigns timestamps to messages as before thus providing total order mechanism. After assigning a timestamp to a message, the sequencer broadcasts message id along with the timestamp instead of broadcasting the whole message. In this way, if the sequencer process goes down, total order would not be provided. But all messages will be delivered to other processes. On receiving a message id/timestamp pair message from the sequencer, a process updates the corresponding message in the queue with the timestamp assigned by the sequencer process.

```
Code for process P_i:
sn ← 0
queue ← empty
mid ← 0

:: receive(app) from APP component
     mid ← mid + 1
     broadcast(app, i, mid) // broadcast app to all processes incl the sequencer
:: receive(app, source, id)
     queue.add(app, id, source)
:: receive(mid, seq)
     updatequeue(mid, seq)
:: (!queue.IsEmpty() ∧ queue.head.seq = sn + 1)
     deliver (app, source) to APP component
     sn ← sn + 1

Code for process Sequencer:
sn ← 0

:: receive(app, source, mid)
     sn ← sn + 1
     broadcast(mid, sn)
```

Figure D.12: *Total order algorithm that uses one process as a sequencer*

Customized version.

The following interaction sets are needed for a customized version:

- SAMT - "send app message to" set
- SSNT - "send sequence number to" set

The customized version is shown in Figure D.13 and D.11 below.

Code for process P_i:
sn ← 0
queue ← empty
mid ← 0

:: receive(app) from APP component
    mid ← mid + 1
    send(app, i, mid) **to SAMT**
:: receive(app, source, id)
    queue.add(app, id, source)
:: receive(mid, seq)
    updatequeue(mid, seq)
:: (!queue.IsEmpty() ∧ queue.head.seq = sn + 1)
    deliver (app, source) to APP component
    sn ← sn + 1

Code for process Sequencer:
sn ← 0

:: receive(app, source, mid)
    sn ← sn + 1
    send(mid, sn) **to SSNT**

Figure D.13: *Customized version of total order algorithm that uses one process as a sequencer*