InDiGO: An Infrastructure for Optimization of Distributed Algorithms

Valeriy Kolesnikov and Gurdip Singh
234 Nichols Hall, Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Abstract: The developers of distributed 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. This paper proposes 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. We perform three types of optimizations: static, dynamic and physical topology-based optimization. We present experimental results to demonstrate the advantages of our infrastructure.

1 Introduction

Several middleware frameworks have been proposed to ease the development of distributed systems with the goal of isolating the developers from lower-level details [3, 2, 11]. The task of a middleware developer is to provide a library of distributed algorithms for common tasks such as computation of global functions, mutual exclusion, and termination detection to simplify programming. Algorithm 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. 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). In such cases, the generic algorithm may be inefficient and may perform redundant communication. 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. 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 invocations to the function use $c$ as the actual parameter). 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.

The problem described above is often addressed by re-implementing all or large parts of the algorithms or by using additional application level control. This can be tedious and error-prone. To address this, we propose InDiGO, an Infrastructure for Distributed alGorithm Optimization, which allows designers to develop generic but customizable algorithms, and provides the tools necessary to customize such algorithms for specific applications. Main elements of InDiGO, whose architecture diagram is shown in Figure 1, are:

- **Tools to extract application information**: We specify applications in Cadena, an integrated development environment for component-based systems. Cadena internally stores application information in an abstract syntax tree (AST) and maintains a component property specification (CPS) file for each component, which contains information relevant to the internal structure of the component. We have developed a tool that uses the CPS files and the AST to construct an application dependency graph (ADG) which cap-
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 paper, 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. 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 this communication may be unnecessary. To eliminate such communication in an algorithm $\text{alg}$, we allow a designer to identify the interaction sets, $\text{alg.interaction.set}$, and write $\text{alg}$ in terms of these sets. Next, the designer defines the membership criteria for each set. The criteria is problem-specific and must be defined in terms of the supported ADG queries. 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. The designer exposes this information by identifying a set of assertions, $\text{alg.app.assert}$, and control points in the algorithm at which each assertion becomes true or false. As shown later, this information can be used for dynamic updates to the interaction sets.

Given an application $\text{App}$, an algorithm $\text{alg}$, and a physical topology, the optimization engine optimizes $\text{alg}$ by removing communication from $\text{alg}$ which is redundant in the context of $\text{App}$ 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 $\text{alg.interaction.set}$. The analysis of the ADG, for instance, may reveal that some nodes can be excluded from intersection sets, which can reduce the number of messages.
- Dynamic optimization: For each assertion $\alpha$ in $\text{alg.app.assert}$, the optimization engine generates a set of dynamic optimization rules which specify whether any set in $\text{alg.interaction.set}$ can be further constrained when $\alpha$ is true. The algorithm $\text{alg}$ 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 [5] for evaluation. We have extended the Core Service Layer (CSL) of J-Sim to perform multi-destination routing. Our infrastructure analyzes the topology graph of 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 have performed experimental studies to demonstrate the advantages of InDiGO. As applications become 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 in 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 applications are 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 include:
- 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 algorithms.

This paper is organized as follows. The next section
discusses the methodology followed in Cadena to develop
component based systems and the problem motivation. Sec-
tion 3.1 discusses the construction of an ADG and Sec-
tion 3.3 discusses customizable algorithms. The following
section discusses the optimization performed. Experimental
results are presented in Section 4. Finally, we conclude
in Section 5.

2 Development of an application

In this section, we discuss aspects of the development
methodology in Cadena which are relevant to our approach.
Cadena is an integrated modeling environment for modeling
and building component based systems [4]. We will use a
simple example of Figure 2 to illustrate the various steps of
the development process. In this example, components are
arranged in disjoint clusters, and are bidding for an item.
Figure 2 shows the components for a single cluster.

- The first step is the specification of the components. A
component C is defined by its input ports, output ports, and
a set of attributes. For example, Figure 2 shows a compo-
nent C1 of type bidCompInit which has two input ports,
\textit{bidCompInit} and \textit{init} and one output port \textit{bidMade}.
Since we want to model asynchronous inter-process in-
teractions via message passing, we will restrict ourselves
to events ports (we do not consider ports for synchronous
method calls such as those allowed in the Corba Compo-
nent Model).

- The next step is to assemble a system by identifying
  the component instances and their interconnections. Fig-
  ure 2 shows the graphical representation of the scenario in
  Cadena. This scenario has one instance, C1, of compo-
nent type \textit{bidCompInit} and two instances, C2 and C3
  of type \textit{bidComp}. In this system, for example, output port
  \textit{bidMade} of C1 is connected to port \textit{nextToBid} of C2.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Bidding Application}
\end{figure}

- The next step is the generation of code and the configu-
raton metadata. Cadena tools generate an implementation
(Java) file for each component into which the designer is
supposed to fill the business logic.

2.1 Problem Motivation

Applications may require several services to accomplish
its tasks. For example, in bidding applications, we may
want to constrain the components to bid one at a time. 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 algorithms implementing different types of services. In
providing such a library, the designers may be tempted to
develop generic reusable algorithms that can be used in a
wide variety of applications. However, stringent perfor-
ance requirements of the applications may tempt the ap-
lication designer to develop customized versions of the al-
gorithms. The following examples illustrate this tradeoff:

- To broaden the applicability, designers often work under
  the “pessimistic” assumption that the application compo-
nents may request critical section entries in any order and
  include the communication necessary to ensure mutual ex-
clusion. 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 applica-
tion, there may be several operating phases, and in a partic-
ular phase, the participants may request access to a shared
document in a cyclic manner. In such cases, one might be
able to take advantage of this application information to re-
duce the number of messages.

- As another example, algorithms for termination detection
  are typically written to determine whether all components
  are passive and all channels are empty. In a particular ap-
lication, however, the passive states of the components may
  be dependent on each other. As a simple example, if com-
ponent 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.

If the algorithms in the library are used as-is in the exam-
pies discussed above, the resulting implementations may be
inefficient. In such cases, an application developer may be
tempted to develop algorithms from scratch suited to the ap-
lication. Such conflicts are especially problematic in prod-
uct line software architectures wherein a fixed middleware
infrastructure may have been developed for a class of appli-
cations (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 algo-
rithms to satisfy their requirements.

In the context of distributed systems, researchers have
addressed this problem in specific domains. For example, a number of algorithms for mutual exclusion optimized for topologies such as ring, trees, and complete networks have been proposed. A number of concurrency control and multicasting algorithms which take advantage of application semantics (e.g., conflict relation among operations) have been proposed [6, 18, 12, 8]. Our work is complimentary to this as it targets application semantics by analyzing the structure of the application. [1] 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). Partial evaluation and aspect-oriented programming have also been used to optimize programs [10, 13]. In our earlier preliminary work, we have designed customizable causal and total ordering protocols for event service middleware [14, 15]. The focus in [14, 15] was on developing a customizable protocol rather than an infrastructure for optimizing distributed algorithms.

3 The proposed framework

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

- **Infrastructure to capture application information**: Exploiting application information will require the application representation in a form amenable to analysis. We provide tools to construct an application dependency graph (ADG), and provide algorithms to analyze the ADG for relevant information.

- **Customizable Algorithms**: To customize the algorithms, they must also 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.

- **Optimization Tools**: Given the ADG and a customizable algorithm, we have developed tools to analyze the ADG to optimize the algorithm by removing communication redundant. 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.

3.1 Application dependency graph

To analyze an application, we construct an application dependency graph (ADG) from two sources of information. First, the Cadena infrastructure stores the application in an internal data structure (an Abstract Syntax Tree (AST)). The AST can be accessed via an API which our analysis algorithm uses to obtain the set of component instances and the port interconnections. 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 3 (the full grammar is given in [4]). For example, the case statement in Figure 3 specifies that when an event on port nextToBid is received, if the variable bidstate’s value is continueBid, then the component executes actions enter_cs, bid, exit_cs, and emits an event on port bidMade. If the value is stopBid, then the component becomes passive and no event is emitted.

![Figure 3. CPS file for component bidComp](image)

We have developed a tool that constructs the ADG from the CPS and AST. The ADG construction algorithm is given in the full paper [7]. For example, the ADG construction tool generates the graph shown in Figure 4 for the application in Figure 2.

![Figure 4. Application graph for scenario in Figure 2.](image)

Each node in an ADG corresponds to an action, and each occurrence of an action is modeled as an event. We use $a^e$ to represent the $e^{th}$ 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. 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. Once a node is added to a state, it is enabled and can be executed.

3.2 Query interface for ADG

The optimization engine uses the ADG to derive information necessary to optimize the algorithms. We have identified a set of queries useful in a number of algorithms, which include the following:

- `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 \( b^{x+1} \) occurs before \( a^{x+1} \) in all ADG executions.
- `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, \( \forall x, x \in \text{actions} = b \Rightarrow (\exists y < x, x \in \text{actions} = a \land \forall z, y < z < x, x \in \text{actions} \neq c) \). Informally, it states that whenever a 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.

We have developed algorithms to analyze the ADG to answer these queries. They are based on depth first traversal of the graph. The description of the algorithms has been omitted due to space constraints.

3.3 Development of customizable algorithms

A library of customizable algorithms can be built in several ways. One approach is to develop a set of algorithms for the same problem, with each algorithm offering advantages over its alternatives under certain conditions. We developed a customizable infrastructure for event communication using this approach in [16]. In this paper, we follow a complimentary approach wherein we want to customize specific algorithms themselves. To customize an algorithm, the algorithm developer must expose the design knowledge in terms of a customizable interface. In this paper, we study mechanisms to expose knowledge related to the communication structure of an algorithm for possible optimizations. A distributed algorithm may have to implement a number of interactions to accomplish several tasks. In implementing interactions, designers typically do not made any assumptions about the application, and perform communication between all possible sites that could participate in the interaction (e.g., either all neighbors or all processes in the system). While such conservative assumptions make the algorithm generic, they may lead to communication that is unnecessary in particular applications. To alleviate this problem, we require the designers to adopt the following problem-centric approach: For an algorithm `alg`, the designer first identifies the "interaction sets" - the set of sites participating in each interaction. This set is denoted by `alg.interaction_set`. The designer then writes the algorithm in terms of these sets. Next, the designer specifies the membership criteria for each interaction set. Here, the designer uses the design knowledge to specify problem-specific properties that must be satisfied for a process to participate in an interaction. Finally, we allow the designer to further leverage the design knowledge to provide information for dynamic updates to the interaction sets. During the execution of the algorithm, a process may gain knowledge of the state of the application entities at other nodes via incoming messages. This information can be used to dynamically update the interaction sets. We allow the designer to provide such information by identifying a set of `alg.app_assert` pertaining to the application, and two sets, `αpos` and `αneg`, of control points where each assertion `α ∈ alg.app_assert` becomes true and false respectively.

We will illustrate these concepts using the mutual exclusion algorithm. Other examples are given in the full paper [7]. Consider the Lamport’s permission based mutual exclusion algorithm [9] to arbitrate access to a shared resource. The communication structure of this 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` (SReIT) is the processes to whom a Release message must be sent.

The algorithm written using these sets is shown in Figure 5. The algorithm is assumed to use logical clocks (clock update instructions are not shown). Next, we define the membership criteria for these sets. The criteria must be defined in terms of the interface actions (such as those to request access to 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. Then, we define \( SRT_i = \{ j : C_j \in S \land \neg exclusive(C_i, enter_cs, C_j, enter_cs) \} \). That is, \( j \) belongs to \( SRT_i \) if \( j \) could potentially request access to critical section concurrently with \( i \). Both WFA_i and SReIT_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 "\( enabled(C_j, cs, enter) \)" (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. [17] proposed a mutual exclusion algorithm using a similar concept to keep track of the application states for determining which processes are likely to have the token so that messages are sent only to those sites. However, in this algorithm, this optimization is already built into the algorithm. Our goal is to perform such optimizations automatically using the application knowledge.

3.4 Static Optimization

Static optimization is performed by computing the initial values of the interaction sets by analysis of the ADG. The inputs for this are:
(a) An application App specified in Cadena along with the algorithm annotations,
(b) Algorithms used by App,
(c) A mapping Map identifying the location of each component in the physical topology.

The first step is the ADG construction. In the next step, for each algorithm used by App, the optimization engine computes the values of the interaction sets by issuing queries on the ADG. 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. The correctness proof for the optimizations (to ensure that the original algorithm’s properties are still satisfied) is delegated to the full paper.

3.5 Dynamic Optimization

Dynamic optimization looks at the set of assertions, \( \text{alg.app.assert} \), for each algorithm \( \text{alg} \) used in \( \text{App} \). For each such assertion \( \alpha \), we declare a boolean variable \( \text{cond}_\alpha \), and insert \( \text{cond}_\alpha := \text{true} \) at each control point in \( \alpha_{\text{pos}} \) and \( \text{cond}_\alpha := \text{false} \) at each control point in \( \alpha_{\text{neg}} \). Next, the optimization engine re-evaluates all of the queries with \( \alpha \) as the condition (e.g., exclusive\((a, b, \alpha)\)). Based on the responses to the queries, we generate 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 \( k \).

3.6 Physical topology-based optimizations

The Core Service Layer (CSL) in J-Sim provides the basic communication services [5]. 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 duplicates are eliminated. The physical topology graph (PTG) describes the underlying network on which the application is to be deployed. We have written a program which analyzes a PTG to generate a path matrix containing the shortest path information, which we then use to initialize the CSL data structures.

3.7 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 paper, 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 ADG can be leveraged by the optimization engines for possible customization. The assertion set, \texttt{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 ADG, which can reveal more optimization opportunities. Also, more sophisticated analysis algorithms can be plugged into the tool-chain to analyze ADG for aggressive optimizations.

4 Experimental results

We implemented several distributed applications to perform experiments that would help us evaluate the optimizations performed. All applications were implemented on J-Sim, a component-based simulation environment [5]. Below, 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.

- **Bidding application 1:**

This application involves twelve players making bids. Each player is located on a separate physical machine and the players are organized into three groups as shown in Figure 6. 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,...).

Figure 6. Application 1 topology

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.

To specify this application in Cadena, we defined two component types, bidCompInit and bidComp. Next, we created twelve component instances and specified the port connections via the graphical interface of Cadena. We constructed the ADG using our ADG construction tool. The optimization engine then used the query interface of the ADG to initialize the interaction sets. 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. bidder1 through bidder3 are excluded because the ADG analysis shows that bidder1, bidder2 and bidder3 cannot make bids concurrently with bidder0. For dynamic optimization, 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). Note that such optimizations are difficult to arrive at by manual inspection. Similar optimization rules are derived for termination detection and total ordering algorithms.

Table 1 shows 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).

<table>
<thead>
<tr>
<th></th>
<th>ME</th>
<th>TD</th>
<th>TO</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_Opt</td>
<td>83</td>
<td>47</td>
<td>336</td>
<td>469</td>
</tr>
<tr>
<td>S_Opt</td>
<td>71</td>
<td>33</td>
<td>293</td>
<td>400</td>
</tr>
<tr>
<td>SD_Opt</td>
<td>45</td>
<td>17</td>
<td>293</td>
<td>355</td>
</tr>
<tr>
<td>SDP_Opt</td>
<td>31</td>
<td>17</td>
<td>170</td>
<td>219</td>
</tr>
</tbody>
</table>

Table 1. Average number of messages per bid

The results show improvement in the number of messages when optimizations are performed. 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 1, for the mutual exclusion algorithm, static application optimization results in 14% improvement as compared to the case with no optimization. Static and dynamic application optimization results in 46% improvement, and addition of platform optimization results in 63% improvement. Similar improvement are observed for other algorithms as well. Since the termination detection algorithm may be initiated several times, the results correspond to the final initiation.
• **Bidding application 2:**

Application context is the same as in Application 1. However, Application 2 imposes round-robin ordering of bids only in group 0 (others can request in any order). Thus, this application imposes less constrains on the components. So, one would expect fewer optimization opportunities. Simulation results are shown in Table 2. As can be seen, the performance improvements are less as compared to Application 1. For example, for mutual exclusion algorithm, the improvement for S\_Opt over No\_Opt is 4% compared to 14% in Application 1. Similarly, the improvement between static and dynamic application optimization over non-optimized case is 14% compared to 46% in Application 1.

<table>
<thead>
<tr>
<th></th>
<th>ME</th>
<th>TD</th>
<th>TO</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>No_Opt</td>
<td>83</td>
<td>47</td>
<td>336</td>
<td>469</td>
</tr>
<tr>
<td>S_Opt</td>
<td>80</td>
<td>40</td>
<td>321</td>
<td>445</td>
</tr>
<tr>
<td>SD_Opt</td>
<td>71</td>
<td>36</td>
<td>322</td>
<td>432</td>
</tr>
<tr>
<td>SDP_Opt</td>
<td>51</td>
<td>36</td>
<td>187</td>
<td>275</td>
</tr>
</tbody>
</table>

**Table 2. Average number of messages per bid**

### 5 Conclusions

In this paper, we proposed an extensible infrastructure to optimize distributed algorithms for specific applications. The capabilities of our infrastructure include the following: (a) Tools to extract and represent application information in a form amenable to analysis, (b) Mechanisms to design customizable algorithms, and (c) Tools to use the application and platform information to customize algorithms. We demonstrated that by allowing the algorithm designer to capture and expose design knowledge, optimization opportunities can be realized. We performed a series of experiments on the class of bidding 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.

### References


