Report Number: WUCSE-2003-36

2003-04-30

A Performance-driven Framework for Customizing CSP Middleware Support

Guoliang Xing

A Distributed Constraint Satisfaction Problem (DCSP) aims to find consistent assignments of values to a set of variables distributed on multiple nodes. Despite its simple definition, DCSPs can model a broad variety of traditional artificial intelligence problems. Furthermore, many problems found in emerging sensor-actuator networks can be formalized to DCSPs. However, due to the platform limitations of networked embedded systems such as sensor-actuators networks, building real-world applications for solving DCSPs not only requires the improved DCSP algorithms but also novel system approaches. This thesis first develops a performance-driven middleware framework for solving DCSP problems. Then the prototype system built...
Read complete abstract on page 2.

A Performance-driven Framework for Customizing CSP Middleware Support

Guoliang Xing

Complete Abstract:

A Distributed Constraint Satisfaction Problem (DCSP) aims to find consistent assignments of values to a set of variables distributed on multiple nodes. Despite its simple definition, DCSPs can model a broad variety of traditional artificial intelligence problems. Furthermore, many problems found in emerging sensor-actuator networks can be formalized to DCSPs. However, due to the platform limitations of networked embedded systems such as sensor-actuators networks, building real-world applications for solving DCSPs not only requires the improved DCSP algorithms but also novel system approaches. This thesis first develops a performance-driven middleware framework for solving DCSP problems. Then the prototype system built with the framework is used to evaluate the performance of special-purpose middleware called nORB that was designed for a Boeing experimental sensor-actuator platform. To validate the design of nORB, various experiments are performed to compare the performance of nORB with other existing DOC middleware platforms. In investigating the problems revealed by the empirical results, we explored various optimization techniques for nORB. The resulting performance of nORB has been improved significantly and is comparable with the high-performance middleware TAO with a decrease in overall footprint.

2003-36

A Performance-driven Framework for Customizing CSP Middleware Support - Master's Thesis, May 2003

Authors: Xing, Guoliang

April 30, 2003

Abstract: A Distributed Constraint Satisfaction Problem (DCSP) aims to find consistent assignments of values to a set of variables distributed
on multiple nodes. Despite its simple definition, DCSPs can model a broad variety of traditional artificial intelligence problems. Furthermore, many problems found in emerging sensor-actuator networks can be formalized to DCSPs. However, due to the platform limitations of networked embedded systems such as sensor-actuator networks, building real-world applications for solving DCSPs not only requires the improved DCSP algorithms but also novel system approaches.

This thesis first develops a performance-driven middleware framework for solving DCSP problems. Then the prototype system built with the framework is used to evaluate the performance of special-purpose middleware called nORB that was designed for a Boeing
experimental sensor-actuator platform. To validate the design of nORB, various experiments are performed to compare the performance of nORB with other existing DOC middleware

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

A PERFORMANCE-DRIVEN FRAMEWORK FOR CUSTOMIZING CSP MIDDLEWARE SUPPORT

by

Guoliang Xing B.E.,M.E.

Prepared under the direction of Prof. Ron K. Cytron and Prof. Christopher D. Gill

---

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

A PERFORMANCE-DRIVEN FRAMEWORK FOR CUSTOMIZING CSP MIDDLEWARE SUPPORT

by Guoliang Xing

ADVISOR: Prof. Ron K. Cytron and Prof. Christopher D. Gill

May, 2003
Saint Louis, Missouri

A *Distributed Constraint Satisfaction Problem (DCSP)* aims to find consistent assignments of values to a set of variables distributed on multiple nodes. Despite its simple definition, *DCSPs* can model a broad variety of traditional artificial intelligence problems. Furthermore, many problems found in emerging sensor-actuator networks can be formalized to *DCSPs*. However, due to the platform limitations of networked embedded systems such as sensor-actuator networks, building real-world applications for solving *DCSPs* not only requires the improved *DCSP* algorithms but also novel system approaches.

This thesis first develops a performance-driven middleware framework for solving *DCSP* problems. Then the prototype system built with the framework is used to evaluate the performance of special-purpose middleware called nORB that was designed for a Boeing experimental sensor-actuator platform. To validate the design of

nORB, various experiments are performed to compare the performance of nORB with other existing DOC middleware platforms. In investigating the problems revealed by the empirical results, we explored various optimization techniques for nORB. The resulting performance of nORB has been improved significantly and is comparable with the high-performance middleware TAO with a decrease in overall footprint.

To Yanni

Contents

List of Figures

Acknowledgments

First, I thank my advisors, Dr. Ron K. Cytron, for all of his help with this work, from introducing me the problem to his numberous inspiring discussions to the approach we took, and Dr.Chris Gill for for guiding me on various aspects of this work and for teaching a lot when I was working on two technical papers with him.

I would like to thank many former or current members in DOC group for providing help both in work and person. These include Venkita Subramonian, Anand Krishnan, Kitty Balasubramanian, Bala Natarajan, Sharath Cholleti, Nanbor Wang, Huangming Huang, Martin Linenweber etc.

Guoliang Xing

Washington University in Saint Louis
May 2003

Chapter 1

Introduction

1.1 *DCSP* and Applications

A *Distributed Constraint Satisfaction Problem (DCSP)* [12] aims to find consistent assignments of values to a set of variables distributed on multiple nodes. Despite its simple definition, a *DCSP* can model a broad variety of traditional open problems. Furthermore, many problems found in emerging research areas can be formalized to *DCSPs*. In particular, *DCSP* techniques have been shown to be very effective in solving many interesting problems of real-time sensor-actuator networks [27].

The Boeing Open Experimental Platform is one of the open experimental sensor-actuator networks built under DARPA NEST program to address the real-time and control problems arising from avionics domain. A typical application of such sensor-actuator networks normally consists of $100 - 100,000$ computational nodes. These nodes integrate physical sensors and actuators which can sense the physical environment, perform the in-node information fusion, as well as actively execute actions to adapt to the changes of the physical host environments. The problem of scheduling resources and activities of sensor-actuators in Boeing OEP have been formalized as a *DCSP* [27]

However, due to platform limitations of real-time sensor-actuator networks, building real-world applications for solving these problems not only needs the improved *DCSP* algorithms [23] but also requires novel system approaches.

1.2 Special-purpose DOC Middleware

General purpose standards-based middleware has been proved to be very effective in supporting a broad range of distributed applications. As general-purpose middleware is applied to a wide variety of distributed real-time and embedded applications, the set of features supported by the middleware evolves accordingly. The growth of features could cause fundamental conflicts with the requirements of individual applications, *i.e.*, the small footprint and high-performance [21]. For example, for a typical application composed of resource-constraint computational nodes, the underlying middleware has to be resource-efficient, e.g., with a small memory footprint. This requirement preclude the application of general purpose middleware in the platforms as Boeing OEP.

Special purpose middleware called nORB [21] has been developed to address this problem. By only adopting necessary features, nORB aims to support the multiple quality-of-service dimensions of special purpose Boeing OEP applications while keeping the footprint small.

However, to validate the design of special purpose middleware as nORB, nontrivial efforts are needed due to the following facts:

- The quality-of-service metrics required by the special purpose applications are different from general purpose applications. Thus the methodology of evaluating the general purpose middleware needs to be extended for evaluating special purpose middleware.

- To guarantee the comparable performance as high-performance general purpose middleware as TAO, careful performance analysis and tuning to special purpose middleware is needed.

- The scale of the Boeing OEP applications is large. A typical application of Boeing OEP normally involves 100s to 10,000s computational nodes.

1.3 Evaluating nORB Using a *DCSP* Application

In this thesis, we first develop a general middleware framework for supporting *DCSP* applications. Then the prototype system ColorCSP built with this general framework is used to evaluate the performance of nORB. To compare with low-level middleware, ACE, on which nORB was built, and general purpose full feature middleware TAO, we

implement ColorCSP on various middleware platforms, including ACE, nORB,TAO and Real-time Event Service of TAO, and perform extensive performance experiments. In investigating the problems revealed by the empirical results, we explored various techniques to optimize the performance of nORB. The resulting performance of nORB has been improved significantly and is comparable with the high-performance middleware TAO.

1.4 Contributions

The contributions of this work include:

- Developed a prototype system for solving the Distributed Constraint Satisfaction (DCSP) problem in Boeing Open Experimental Platform.
- Validated the design of nORB with extensive empirical performance results.
- Explored various optimization techniques for optimizing the critical performances of nORB.

1.5 Road Map

This thesis is organized as follows. In chapterřefcpt:background, we discuss the background for our work. Then we present the design of our prototype system ColorCSP in Chapter 4. We present the experiments in Chapter 5. Finally,we conclude the thesis in Chapter 6.

Chapter 2

Background

This chapter provides background information on *distributed constraint satisfaction problems(DCSP)* and their applications. After the discussion of a real-world DCSP application of NEST Open Experimental Platform, DOC middleware support for DCSP is presented.

2.1 Distributed Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) [12] aims to find consistent assignments of values to a set of variables, whose inter-dependencies represent the constraints of the problem. Even though the definition of CSP is very simple, a broad variety of artificial intelligence problems can be formalized as CSPs [25].

A distributed CSP [25] is a constraint satisfaction problem (CSP) in which variables and constraints are distributed among multiple automated agents. An example is distributed resource allocation problem in network, as described in [4]. In this problem, each agent has a set of tasks and there are several ways (plans) to complete each task. Since resources are shared among agents, constraints/contentions exist between plans. The goal of this problem is to find the combination of plans in which all the tasks could be performed simultaneously. This problem can be formalized as a distributed CSP, each task being a variable whose possible values can be represented by plans.

Since the knowledge of the problem (*e.g.*, variables and constraints) is distributed among different agents in DCSPs, *i.e.*, an agent only has the limited information about the global problem. For example, in some application problems as we will discuss in following section, accessing all information from one agent is cost

inefficient or physically infeasible. In these cases, multiple agents have to solve the problem without the knowledge of the global information.

2.2 DCSP Algorithms

In this section we briefly discuss existing algorithms for solving DCSP.

2.2.1 Trivial DCSP Algorithms

- Centralized Method [25] : The most trivial algorithm for solving DCSP is the central method in which a coordinating agent is selected among all agents and all information about the problem(*e.g.*, variables, constraints etc.) are gathered by the coordinating agent. Since the coordinating agent has the full knowledge of the problem, it can solve it using centralized constraint satisfaction techniques. However, as we discussed in previous sections, the cost of gathering all information of the CSP in a central node is cost infeasible in some application contexts, *e.g.*, sensor networks.

- Synchronous Backtracking [25]: Assume all agents agree on an instantiation order of their variables(*e.g.*, agent1 operates on its variables first, then the agent2 and so on). When the turn of an agent comes, it can receive a partial solution from the previous agent. After instantiating its variables based on its constraints, the agent can append the values of its variables to the partial solution and pass it to the next agent. If its constraints can't be satisfied, it sends a *backtracking* message to the upstream agent. Although this algorithm doesn't incur the high communication overhead of the central method, achieving a agreement on the instantiation order is also expensive or even infeasible in some contexts. Furthermore, all agents are totally synchronized, only one agent can proceed at any given time in this algorithm.

2.2.2 Distributed DCSP Algorithms

Two effective DCSP algorithms are proposed in [25], namely asynchronous backtracking and weak-commitment search algorithm. Asynchronous backtracking algorithm overcomes the drawback of synchronous backtracking by allowing the agents to instantiate their own variables asynchronously. Each agent makes the local decision

on solving its constraints by changing the assignment of its variables and notify the agents with which the agent shares constraints the new values of its variables. If a valid assignment to an agent' variables can't be found to satisfy the constraints, the agent executes a *backtracking* process in which the information of current inconsistency is sent to the corresponding agents, which will change their value assignments to resolve the violations to the constraints. This process is repeated in each agent until a global solution satisfying all constraints is found.

Weak-commitment algorithm behaves similarly with asynchronous backtracking except that the agents are prioritized dynamically according to their current states. And every agent will try to minize the violations with lower priority agents by choosing new values to its variables. When a solution can't be found to satisfy the constraints, the agent will increase its priority and notify its inconsistency of the constraints to the corresponding agents. In asynchronous backtracking algorithm, agents make decision independently to solve the constraints and backtrack whenever a constraint is violated. From the global view, the strategy resembles the exhaustive search in solution space. By assigning higher priorities to inconsistent agents, weak-commitment algorithm can revise a bad decision without exhaustive search and thus outperforms asynchronous backtracking in large-scale DCSP problems.

Both weak-commitment and asynchronous backtracking algorithms are complete, *i.e.*, they will always find a solution, or find the fact that no solution exists.

In [26], distributed breakout algorithm(DBA) is proposed to solve DCSP. In DBA, a partial solution which has unsatisfied constraints will be improved by local changes until a global solution is found, *i.e.*, all constraints are resolved. Each variable pair whose value assignments violate a constraint is assigned a weight. The sum of weights of such variable pairs, which is called the *value* of current evaluations, is used to evaluate the current solution. Each agent will try to locally reduce the *value* of current evaluations by changing the current assignments to its variables. The reduction to the sum of constraint violating weights is called *improvement* of current evaluation. By allowing the agents to make such decisions locally, the *value* of current evaluation might not be reduced if multiple agents choose conflicting value assignments. DBA solves this problem by only allowing the agent who has the maximal *improvement* to change its value assignments. If the *value* of current evaluation can't be reduced by changing any variable, the state of the algorithm is called a *local- minimum* in which no agent can improve the solution. DBA tries to escape the *local-minimum* state

by incrementing the weights of constraint violating variable pairs. Thus the *value* of current evaluation is changed and the solution can be improved possibly.

In [23], the authors proved that the complexity of DBA is $O(n^2)$ on acyclic graph coloring problem. However, DBA is not complete on cyclic constraint graphs, *i.e.*, DBA may never terminates on a cyclic graph even if there is a coloring solution. In addition, the authors proposed two stochastic variations to the original DBA, namely DBA(wp) and DBA(sp). The new algorithms differ with DBA when two neighboring agents have the same *improvement* for the current evaluation. Instead of only allowing the agent who has the highest *improvement* to change the value assignments to its variables, DBA(wp) allows both agents to change their value assignments probabilistically. In DBA(sp), even an agent who doesn't have the highest *improvement* can change its value assignments probabilistically. The experimental results on grid and random graphs show that both DBA(wp) and DBA(sp) are comparable with DBA while they increase the probability of convergence to optimal solutions.

[27] studied an existing distributed stochastic algorithm(DSA) and its variations for solving distributed CSPs. The authors prosed 5 stochastic strategies which an agent can take to decide the next values to its variables based on the current states of itself and its neighbors. The experimental results show that the best strategy of proposed DSAs is competitive for finding good or optimal solutions which needs a long execution.

2.3 Sensor Networks

In recent years, advances in micro-electro-mechanical systems(MEMS) have introduced the possibility of producing small-size low-cost sensor devices on a mass scale. A large amount of sensors and actuators with limited information processing capabilities have been developed and deployed in many real-world applications [11]. In these applications, various types of physical information such as temperature, pressure, photo, etc. are monitored by sensors. Since the computational resources (memory,CPU etc.) are highly constrained in each sensor node, multiple sensor nodes must collaborate effectively to perform complex tasks, *e.g.*, tracking the intruders. Furthermore, since sensor nodes are battery operated, the energy cost is one of the most important factors when building such kind of systems. Since the energy consumption of wireless communication is much higher than CPU cycles, network traffic should be reduced to prolong the system life time. Thus a new class of algorithms,

namely *local algorithms* [6], which process most of the data locally and only need the information of the neighbors are preferred in sensor network.

Many problems in sensor networks can be formulated as DCSPs due to the characteristics of sensor networks:

1. The sensor networks are distributed in nature. A typical sensor networks application involves the order of 10-1000 sensor nodes.

2. The computational resources (memory,CPU etc.) are highly constrained in each sensor node. Thus the sensor nodes must collaborate effectively to perform complex tasks.

3. In collaborating with its neighboring nodes to perform a complex task, a sensor node in a sensor network normally only has the partial knowledge of the global problems. Thus the inter-dependencies across multiple sensor nodes are introduced in solving the problems.

[1] presents SenorCSP, a distributed constraint satisfaction problem (DCSP) based on a system of wireless sensors tracking multiple mobile nodes. This problem can be formalized to two DCSP models: one in which agents are associated with the sensor nodes, and one in which agents are associated with the target mobile nodes which are to be tracked by sensor nodes. In addition, the differences in term of the number of intra and inter-agent constraints and the cost of finding a distributed solution are discussed.

[13] formalizes the distributed resource allocation arised from distributed sensor networks to dynamic distributed constraint satisfaction model.

In the next section, we introduce the Boeing OEP, a real-world sensor-actuator testbed , which was a motivating platform for our work.

2.4 NEST OEP

The main tasks of NEST program are to provide [5]:

- application independent time-bounded coordination services that adapt to environmental changes,

- time-bounded synthesis methods for resource reallocation, schedules etc. in NEST systems,

- automated composition and adaptive run-time customization of coordination services, and
- formal verification and assurance of algorithms, software, and system properties.

To meet the requirements of NEST systems in different domains, a variety of endsystem, operating system, middleware, and networking infrastructures are being explored. For example, Boeing's Open Experimental Platform (OEP) for the Networked Embedded Software Technology (NEST) program is designed to provide an open platform for researching the real-time, monitoring and control problems arising from the domain of DoD air and space vehicles. Typical Boeing OEP applications are characterized by scale factors of 100s to 10,000s nodes, each of them is equipped with Micro Electro-Mechanical Sensor (MEMS) vibration sensors and actuators and runs small-scale common-of-the-shelf (COTS) operating system, communicates with other nodes via embedded middleware framework over wired network connections.

Figure 2.1 illustrates one possible topology of the Boeing NEST OEP, with each node connected to each of four neighbors via wired connections to form a consistent mesh of squares. All nodes are distributed uniformly across the physical surface whose vibrating activity is monitored and controlled by the sensor-actuators equipped on the nodes.

An important challenge addressed by the Boeing OEP is the vibration damping problem whose goal is to reduce the vibro-acoustic affect for DoD air and space vehicles by finer grain control of acoustic and structural vibration mode in vehicle subsystems. This kind of control is achieved by 100s to 10,000s of nodes driving sensors and actuators, which are mounted on the surface of the fairing structure. The system designed to address the damping problem in a simulated hardware environment includes the following basic functionality:

- A simulation of the fairing structure.
- A 100 computational sensor-actuator node mounted on the surface of the fairing structure. Each node consists of a sensor, actuator, and processing element. The sensor and actuator will be simulated and interfaced with the fairing simulation.
- A hierarchical control and system identification component to adapt the system responses to changes in the structural vibration modes of the fairing that would occur during launch.

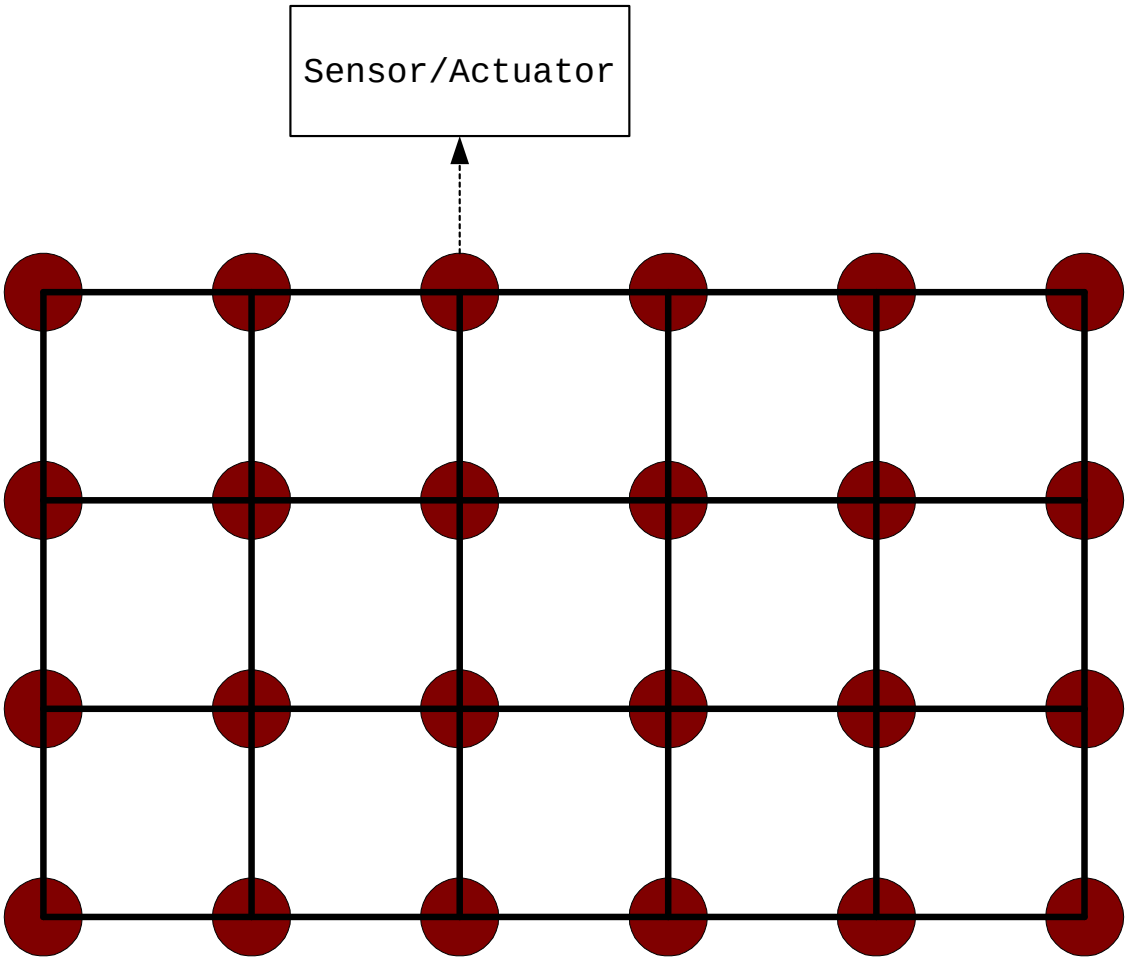


Figure 2.1: A possible Network Topology in the Boeing OEP

- Processes for detecting and reacting to simple node (e.g., sensor, actuator, processor) failures.

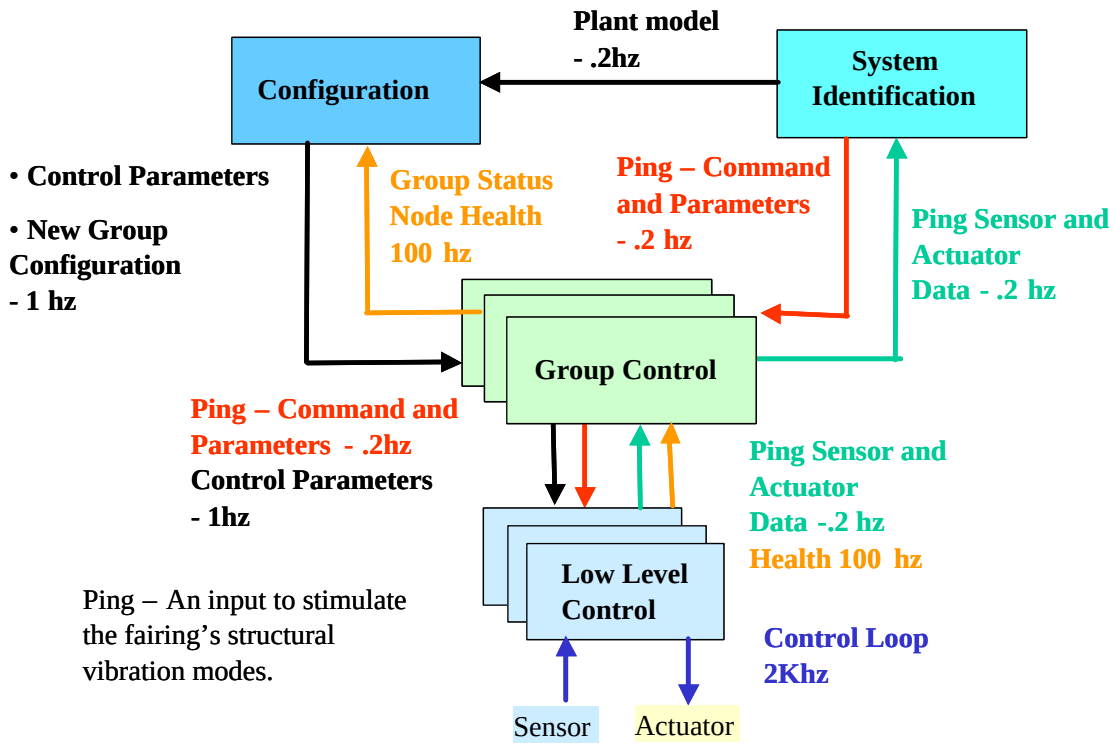


Figure 2.2: Functional Structure for the Vibration Damping Problem in the Boeing OEP

Figure 2.2 illustrates the functional diagram of the Boeing NEST OEP for solving damping problem. The top level system components are: configuration, system identification, group control and low level control. The Sensor-Actuator component is composed of 100 nodes which are coupled with small-scale sensor-actuator devices. As illustrated in Figure 2.1, these nodes typically have a grid topology and communicate via wired connections.

The *System Identification* component is responsible for identifying the current system vibration mode. This is achieved by sending *Ping* commands to the sensor/actuator hardware. After processing the The *Ping* response data from the sensors, the *System Identification* component passes the refined data to the *Configuration* component.

To achieve the most effective damping, the 100 nodes will be partitioned into groups by *Configuration*. Each group will act to damp a particular vibration mode of the fairing. The group control component coordinates actions of nodes in a group. A physical node might be acting as a group controller which will coordinate the actions of other nodes in the group. The factors considered by the *Configuration* component in deciding the groups include the frequency of the mode of interest, the energy required to achieve the damping and the number of nodes available. There will be as many as 20 vibration modes to address. Furthermore, the *Configuration* component will decide when and how to reconfigure the groups and update damping control parameters based on feedback of how well damping performs under the current group configuration and damping parameters.

Low level control implements control logic performed by each individual node.

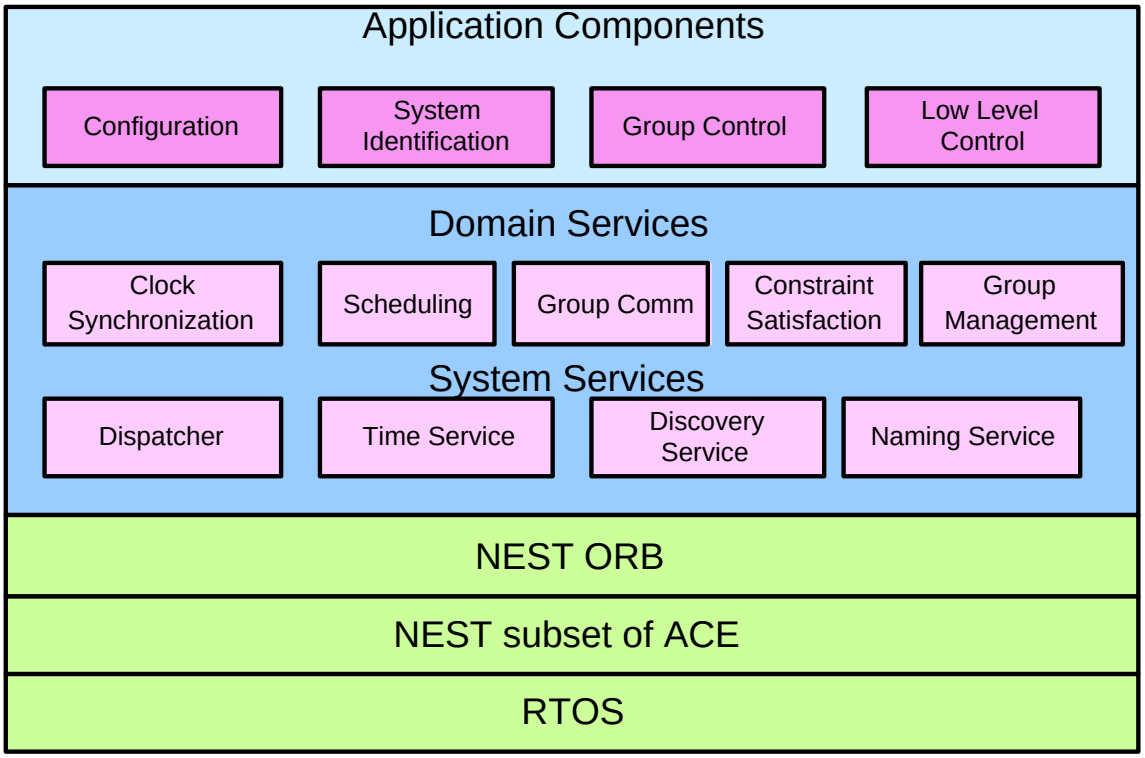


Figure 2.3: The Structure of Boeing OEP

The Boeing OEP architecture can be divided into layers as shown in Figure 2.3. This facilitates the separation of application domain level concerns from the reusable middleware services like Group Management, Constraint Satisfaction,

etc. This separation reduces the application-level development efforts and improves system reusability.

One example illustrating the interaction between the application and system service is the *Ping node scheduling* problem we will discuss in the next section.

2.5 Ping Node Scheduling

In this section, we describe the Ping Node Scheduling problem which is a representative of sensor network CSP applications. To identify the vibration mode the system is currently in, the *system identification* component needs the system vibration information obtained by the nodes. This is done by sending a *Ping* command to sensor-actuator nodes. One constraint of this problem is the signaling actions of two overlapping ping nodes should be synchronized so that no interfering signals will be generated. The problem of finding a schedule for ping node responses can be modeled as a distributed constraint satisfaction problem [27].

In particular, the ping scheduling problem can be formulated in terms of a well-known distributed CSP: *distributed graph coloring* [27]. In *distributed graph coloring*, the goal is to find a valid color assignment for all vertices of a graph, each an autonomous node in a distributed network, and the constraint being that two adjacent vertices (i.e., two vertices connected by an link) cannot be assigned the same color. In the context of the ping scheduling problem, the network of sensors corresponds to a graph, a sensor-actuator node corresponds to a vertex in the graph, and connections between sensor-actuator nodes are represented by edges in the graph. The time slot scheduled for a ping node corresponds to a vertex color assignment in the distributed graph coloring problem.

2.6 DOC Middleware Support for DCSP

One of primary goals of our work is to provide a general framework for solving *distributed constraint satisfaction problem* arising from different domains as we discussed in previous sections. We decided to build our framework on Distributed Object Computing(DOC) middleware which has been shown to be very effective in meeting a wide variety of requirements for distributed real-time and embedded(DRE) systems. The DOC middleware platforms we used in this work are discussed in following sections.

2.6.1 ACE and TAO

The ADAPTIVE Communication Environment (ACE) [16, 3] provides a set of highly resuable key framework components for delevoping object-oriented network applications. The ACE provides following key features in a unified framework:

- a set of C++ wrapper facades encapsulating various underlying operating system APIs.
- a set of higher-level object-oriented programming abstractions for various key system services for distributed systems, *e.g.*, threads,IPC etc. and
- a set of key design patterns for programming concurrent and networked systems [18].

With the highly portable pattern rich middleware infrastructure provided by ACE, the overall complexity of developing distributed applications is redued.

TAO [7, 2] is a widely used standards-compliant real-time object request broker (ORB) built using the ACE framework. TAO provides a highly optimized [15, 22] high-performance ORB implementation as well as a variety of higher level CORBA services [10, 8].

2.6.2 nORB-Special Purpose DOC Middleware

Although general purpose standards-based middleware have been shown to be very effective in supporting a diversity of distributed applications, they fail to address the challenges of supporting special-purpose applications that require both the real-time performance and small memory footprint. Reducing features for lower footprint while maintaining or improving performance in comparison to existing high-performance real-time general purpose middleware frameworks like TAO is a significant challenge. Careful analysis for multi-dimensional requirements from application is required when selecting the necessary features. To address this problem, special purpose middleware [21] has been developed to meet the requirements of small footprint and stringent quality-of-service assurances. Two design forces exist for special purpose middleware:

- The middleware should be general enough that common abstractions can be re-used across applications with different requirements in the same domain.

- To meet the requirements of individual applications, the middleware should be highly customizable and flexible fine-grain modifications and tuning.

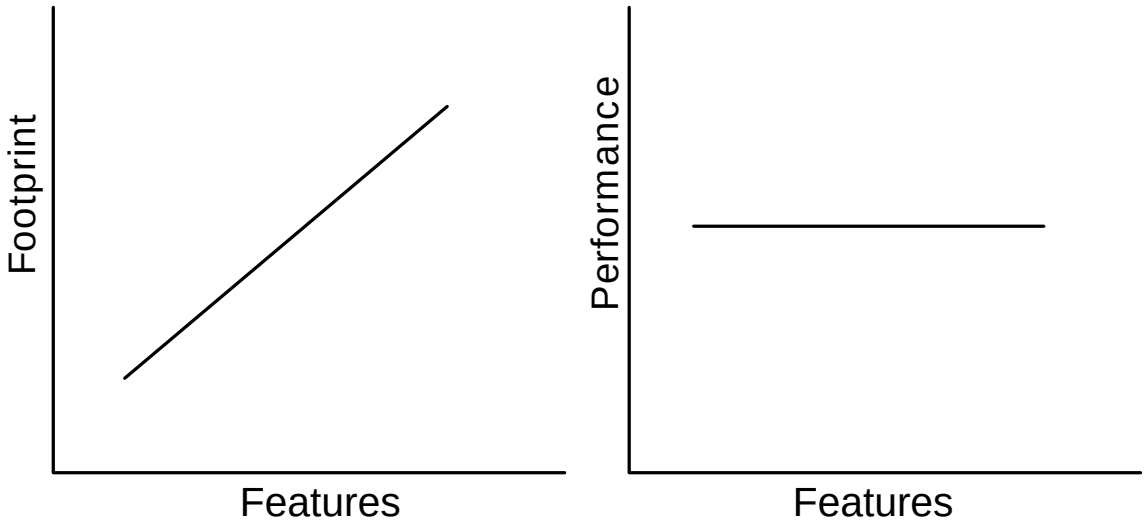


Figure 2.4: Features, Footprint, and Performance Goals

Figure 2.4 illustrates the ideal relationships between features, footprint and performance.

nORB is a small footprint ORB implementation developed by DOC group at Washington University in St.Louis [20]. nORB aims to achieve the small footprint by only adopting necessary features, meanwhile keep the real-time performances. However,validating the design of such special purpose middleware needs highly customized benchmark applications and careful analysis to the performance. One of our primary tasks is to validate the design goals of nORB with the application of *distributed constraint satisfaction problem(DCSP)*

Chapter 3

The System Model

In this chapter, we present a system model on which our framework for solving CSP problem is built. The model aims not only to abstract *Distributed Constraint Satisfaction Problems* but also to express the high-level system architecture and behavior of our framework. We first discuss the formalization of DCSP [25], and then the system model.

3.1 Distributed CSP Model

A distributed CSP is a CSP in which the involved variables and constraints which need to be satisfied are distributed among different automated agents. Each automated agent is a computational node which only has the partial knowledge of the CSP(*i.e.*, the variables and constraints). In addition, the following assumptions [25] are made:

1. Each agent has an unique identifier.
2. Agents communicate by sending messages which have well-defined formats.
3. The delay in delivering a message is finite. For any pair of agents, messages are received in the order in which they are sent. However, the global order of the messages is not required.
4. Each agent has exactly one variable.
5. All constraints are binary.
6. Each agent knows all constraints involving its variable.

In a general DCSP, the last three assumptions can be relaxed.

3.2 The System Model

A CSP [25] consists of n variables and a set of constraints on the values of the variables. For variables $x_1, x_2, \cdots, x_n$, the possible values are taken from finite, discrete domains $D_1, D_2, \cdots, D_n$ respectively. A constraint is defined by a predicate $pk(x_{k1}, x_{kj})$ that is defined on the $D_{k1} \times \cdots \times D_{kj}$. The predicate becomes true iff the value assignment of these variables satisfies the constraint. Solving a *Constraint Satisfaction Problem* is equivalent to finding an assignment of values to all involved variables such that all constraints of the problem are satisfied.

AS illustrated in Figure 3.1, We model the system as follows:

1. The system is composed of a number of *Agents* which are automatous computational nodes.

2. Each *Agent* embeds a *CSP Solver* which runs a distributed algorithm to solve the CSP defined by a set of *Constraints* and *Variables*.

3. *CSP Solver* is an abstraction to the CSP algorithm used in the system,*e.g.*, distributed breakout algorithm etc.

4. An *Agent* can communicate with other *Agents* only by sending *Messages* which include the intermediate states exchanged between *Agents* to solve the CSP. The *Message* sent by one *Agents* will trigger some actions, *e.g.*, computation or sending a *Message* in other *Agents*. However, the *Messages* don't need to be in total order.

5. *Communication Channel* represents the communication mechanism via which a *Agent* can communicate with other *Agents*. It should note that the Communication Channel only exists between two *Agents* which need to exchange *Messages*. It could employ different underlying communication mechanisms as long as the message delivery is achieved.

3.3 Distributed Coloring-A Case Study

In this section, we instantiate our system model using an instance of the general DCSP problem - a distributed coloring problem.

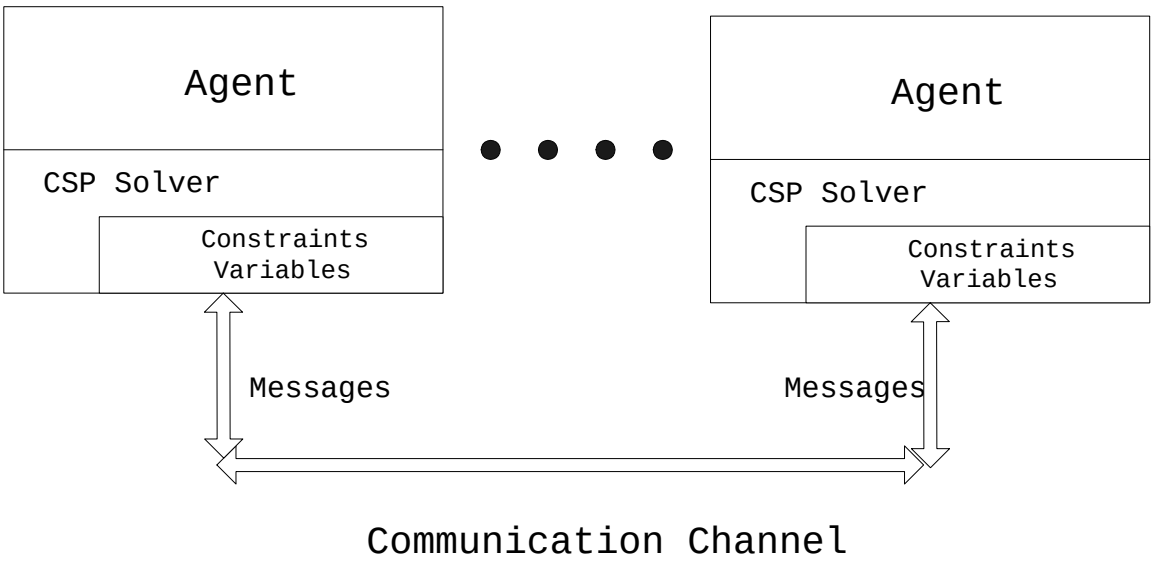


Figure 3.1: System Model

In the distributed coloring problem, each vertex of the graph is represented by an autonomous node in a distributed network. The goal is to find a valid color assignment for all vertices of the graph. The variable each *agent* holds represents the value of the current color assignment to the vertex. The constraint is that two adjacent vertices, (*i.e.*, two vertices connected by an link) cannot be assigned the same color. Thus the topology of the graph represents the constraints of the problem. In applying our system model presented in last section, we can view *agents* as distributed processes running in a network. In each process, a *CSP Solver*, *i.e.*, distributed CSP algorithm runs until a valid solution for the problem is reached.

Chapter 4

The Design of ColorCSP

In this chapter, we present the design of ColorCSP, our middleware framework for supporting CSP applications following the architecture presented in Section 3.

Our primary goal of developing such a system is to evaluate the capacity of various DOC middleware platforms for supporting the *DCSP* applications. In particular, the system is used as benchmark application for comparing the performance of small footprint middleware nORB with other existing DOC middlewares.

The challenges of developing a middleware system for supporting *DCSP* applications include:

1. There exist a variety of DOC middleware platforms, which have very different architectures and programming models. The system must have an open architecture and generic design to adapt to the different underlying middleware platforms.

2. The research on related algorithms is very active. To take advantage of advances in CSP algorithms, the design and implementation of the CSP algorithm in the system must be separated from other system components so that different CSP algorithms can be integrated into the system easily.

3. Since the CSP problem we want to solve arises from the embedded real-time domain 2.5, performance is a primary driving factor for the design and implementation of the system. The system has to be built with the understanding of the performance issues, arising from both the application level and middleware-level.

4. To evaluate the underlying middleware platforms and analyze the performance results, we need to instrument the operations in both the application-level and middleware-level. To separate different performance concerns, the system design must be modularized and easy to instrument.

First, the high-level system architecture is presented, then we describe the design of the generic framework that provides the portability across multiple middleware platforms. At last, the designs of different versions of ColorCSP are presented.

4.1 The System Architecture

The high-level system architecture follows the system model we presented in Chapter 3. As illustrated in Figure 4.1, the system is composed of the following components:

1. *Constraint Factory* stores all constrains that need to be satisfied. Each constrain is expressed by the dependence between the CSP variables of *nodes*.

2. *CSP Solver* module embeds the CSP algorithm used in the system,*e.g.*, the distributed breakout algorithm.

3. *Node* represents an autonomous agent in which a *Constraint Solver* is embedded. A Node can obtain the constraints involving itself from *Constraint Factory* and communicate with other *nodes* to solve the CSP problem.

4. *Communication Channel* represents the communication mechanism via which a *node* can communicate with *Constraint Factory* or other *nodes*.

5. *Nodes* communicate with each other by exchanging *parameter messages* that include the intermediate parameters produced in the evaluation process of the *CSP Solvers*. The *parameter message* sent by one *nodes* will trigger some actions, *e.g.*, sending a *parameter message* in other nodes. However, the *parameter messages* sent by *nodes* are not in total order.

To reduce the complexity in composing the different components of the system, the event driven model is used as our central design paradigm in delevoping ColorCSP. *nodes* are modeled as event processors. Each *parameter message* received

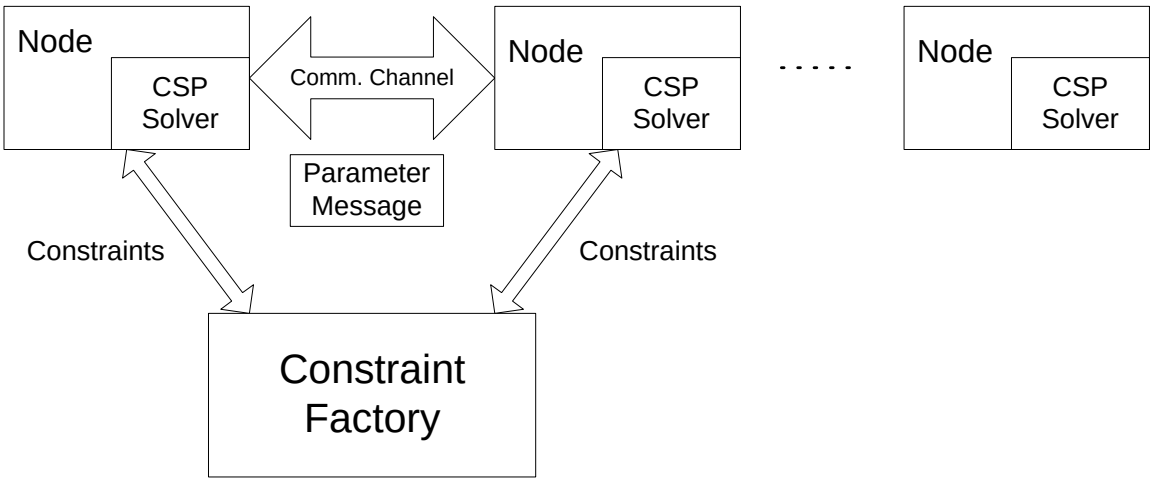


Figure 4.1: ColorCSP System Architecture

from another *node* serves as a event that triggers certain actions, *i.e.*,computation or sending *parameter messages* to other *nodes*.

Since ColorCSP is built for solving the distributed coloring problem, the *Constraint Factory* stores the topology of the graph that is to be colored. And each *node* represents a vertex of the graph. The *CSP Solver* we used in the system implements DBA [24] that is one of the most effective existing DCSP algorithms. It should note that an *agent* of DCSP only has a partial knowledge of the problem, *i.e.*, variables and constraints, and there normally doesn't exist a central node like the *Constraint Factory*. In our system, the *Constraint Factory* component is used only for simplifying configuration, *i.e.*, if each *node* can be configured with its CSP variables and constraints before run time, the *Constraint Factory* component can be removed. Although our system aims to solve *distributed graph coloring* problem, it can be extended to solve various DCSPs. In the following discussion, we focus ourself on the *distributed graph coloring* problem.

The sequence of events that happen in the system is as follows:

1. *Constraint Factory* loads a graph definition file in DIMACS standard format and reads the topology of a graph for which we need to find a valid color assignment. Then it will accept the registrations from *node* processes.

2. *Nodes* register with *Constraint Factory*, which stores the IDs and addresses of the *nodes*.

3. *Constraint Factory* returns constraint information to each *node*. For graph coloring problem, an edge of the graph represents a constraint in the corresponding CSP. Thus the constraints of a *node* in graph coloring problem can be represented by a set of edges incident to the *node*. *i.e.*, for solving a CSP, a *node* only needs to know the information about the adjacent *nodes* in the graph. In this sense, the CSP algorithms used in the system are *local algorithms* in which each *node* only has the knowledge of neighboring *nodes*.

4. *Nodes* run the *CSP Solver* until a termination condition, *e.g.*, any two connected vertices have different color assignments.

Furthermore, the processing of *CSP Solver* consists of a number of *cycles*. Each *cycle* involves the following steps:

1. Receiving all relevant *Parameter Messages* from neighboring *nodes*.

2. Performing one step of algorithm-specific evaluation based on the current state and neighbors' states that can be retrieved from incoming *Parameter Messages*.

3. Sending the result of evaluation(*e.g.*, a new value assignment or improvement message) to all neighboring *nodes*.

The *CSP Solver* module will be discussed in greater detail in Section 4.3.

4.2 Design of Communication Channel

Depending on the middleware platforms on which our system is built, the *Communication Channel* could employ one of the following mechanisms:

- Socket communication API: In the ACE implemention of our system, the socket API is used by a *node* to communicate with other *nodes*.

- Remote object function invocation: In the nORB/TAO implementations of our system, when a *node* wants to communicate with another *node*, it sends an one-way object request encoding a *parameter message* to the target.

- Event channel: In our implementation based on TAO Real-Time Event Service [9], each *node* is associated with some Event Channel via which the *node* can send or receive a *parameter message* to a target *node*.

As illustrated in Figure 4.2, three subclasses are derived from the common Comm_Channel abstract class, namely Comm_Socket that implements the socket network communication for ACE version of ColorCSP, Comm_Obj that sends messages by marshaling them into oneway object function calls, and Comm_EC that sends messages by invoking *push* operations of event channel. The attributes and methods of *Communication Channel* component are illustrated in Figure 4.3.

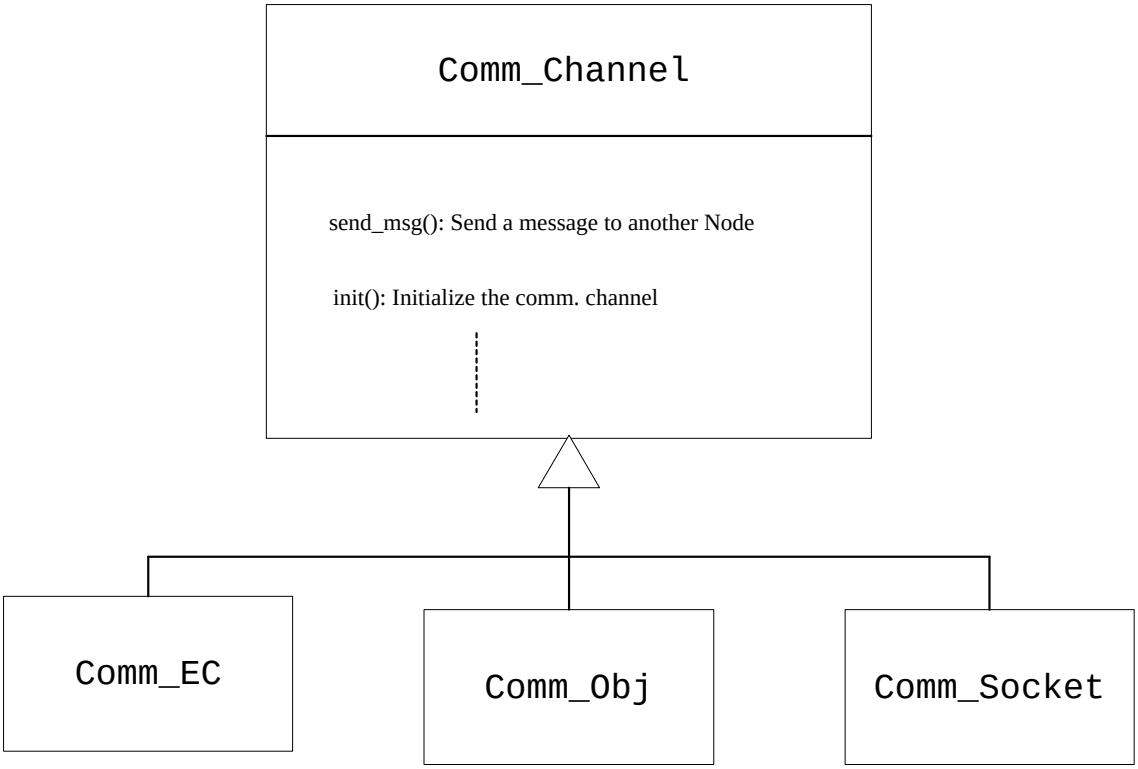


Figure 4.2: ColorCSP Communication Class Hierarchy

Attributes/Methods	Description
constraints_	A vector storing information about the neighbors, *e.g.*, the addresses etc.
init()	Initialize the communication channel
send_msg()	Send a *parameter message* to all neighboring *nodes*

Figure 4.3: The Attributes/Methods of Communication Channel

4.3 Integrating DCSP Algorithms

Different DCSP algorithms can be applied in *CSP Solver* component of ColorCSP. The class hierarchy is depicted in Figure 4.4.

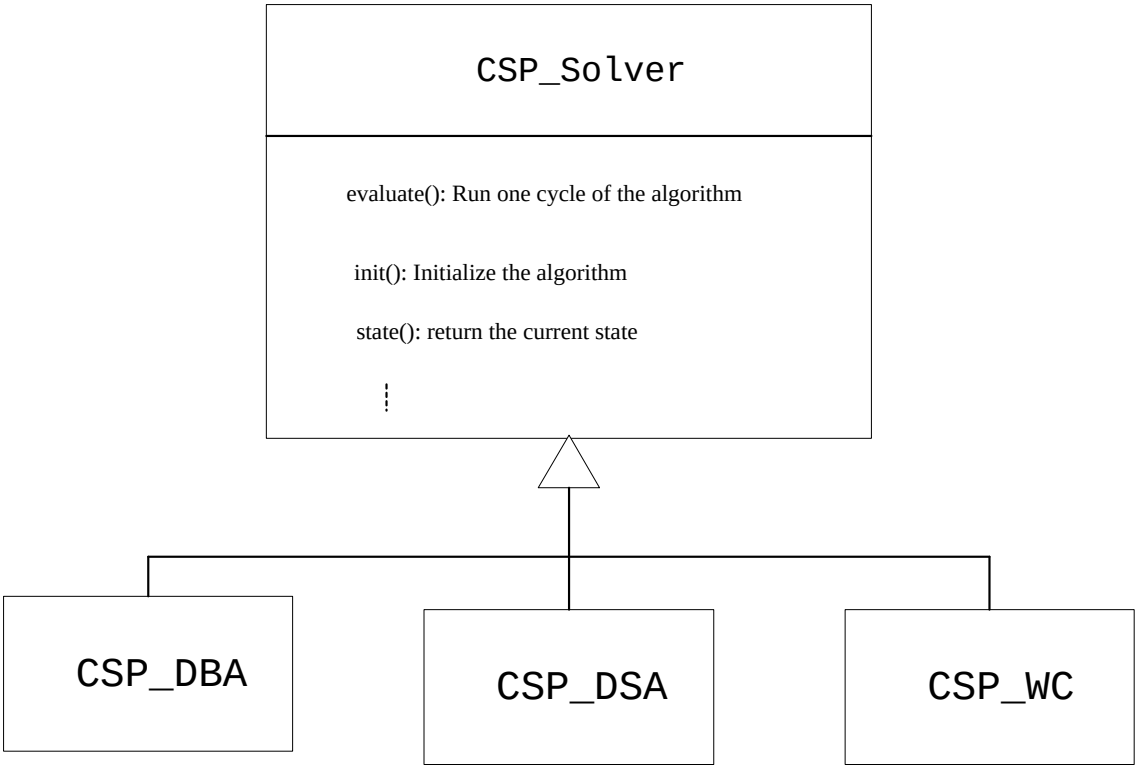


Figure 4.4: Class Hierarchy for *CSP Solver* Component

The subclasses shown in Figure 4.4 implement three DCSP algorithms: Distributed Breakout Algorithm(DBA) [26], Weak-commitment Algorithm [25] and Distributed Stochastic Algorithm(DSA) [27]

The main attributes and methods defined in *CSP Solver* are illustrated in Figure 4.5.

Currently, only *Distributed Breatout Algorithm* is supported by ColorCSP. The details of the algorithm is discussed in [26].

Attributes/Methods	Description
attrs_	The attributes of the node, *e.g.*, the current color assignment etc.
dynamics_	The dynamic parameters needed by DBA, *e.g.*, the current value of evaluation, the value of termination counter etc.
init()	Initialize the DBA algorithm
evaluate()	Perform a step of DBA evaluation based on the states of the node and all neighboring nodes.

Figure 4.5: The Attributes/Methods of CSP Solver

4.4 Generic *Node* Framework

To maximize the reusability of the design when the system is ported to different middleware platforms, *e.g.*, ACE,nORB,TAO etc., the following design forces need to be addressed:

1. The distributed CSP algorithm used in the system is subject to change. Thus the *CSP Solver* should be decoupled from the other components and be independent with the underlying platforms.

2. The communication mechanism is platform specific. Thus the functionality of *Communication Channel* component should be well encapsulated such that the other components of the system are decoupled from the low-level platform specific communication mechanisms.

3. Since the *node* component embeds the *CSP Solver* and *Communication Channel* components, the interfaces between them should be well defined to reduce the complexity of composition.

In our design, three abstract classes are defined to represent the system components, namely Node_Base, Comm_Channel and CSP_Solver. For each platform, a new subclass is defined to for every component to implement platform specific functionality, while the common abstract classes are reused. For example, in the TAO version of ColorCSP, Node class needs to inherit the CORBA stub supper class, while in the ACE version of ColorCSP, Node class needs to inherit the ACE_Task class to

implement the functionality of thread. As depicted in Figure 4.1, the *node* component needs to compose the the functionality of communication channel and CSP solver. To achieve the composition, we consider the following two approaches:

1. Parameterization: Templates bring a considerable flexibility to object-oriented programming by parameterizing the type definitions. *Communication channel* and *CSP Solver* can be template parameters for Node class. By this way, the Node can adapt different type of *Communication channel* and *CSP Solver components*. In addition, this approach provides better runtime performance by instantiating the templates at compile time.

2. Inheritance: Node class can inherit the interfaces of both *Communication channel* and *CSP Solver* components. Polymorphism is obtained if the actual types of objects are unknown at compile time. This approach avoids the complexity of organizing multiple subclasses.

To achieve the runtime efficiency and the flexibility of adapting different type of components, the parameterization approach is favored over inheritance. As depicted in Figure 4.6, in our system framework, the Node class takes two template parameters: Comm_Channel and CSP_Solver.

As illustrated in Figure 4.6, the Node class is derived from Node_Base class that defines the common interfaces for *Node* component. Figure 4.7 shows the attributes and methods defined in *Node* component.

4.5 Design with ACE

In this section, we present the design of ColorCSP on ACE.

4.5.1 Applying Event Handling Design Patterns

As discussed in Section 4.1, each *node* process in ColorCSP has to support the following functionalities:

1. Register itself with *Constraint Factory*. This step involves making connection to *Constraint Factory* actively, sending relevant registration information and receiving the information about its neighboring *nodes*.

2. Accept incoming connections from neighboring *nodes*.

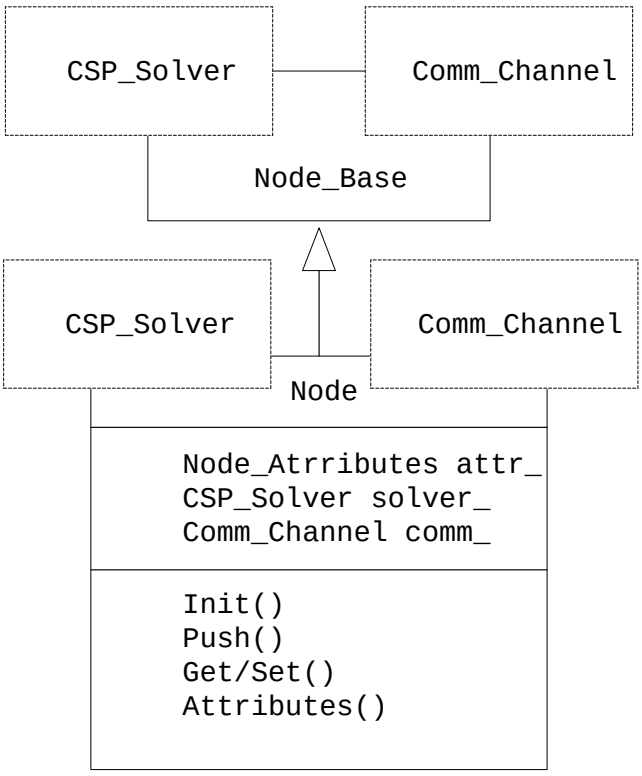


Figure 4.6: ColorCSP System Framework

Attributes/Methods	Description
attr_	A vector storing information about the Node, *e.g.*, the ID, current color assignment etc.
solver_	An object of *CSP Solver* is defined to implement the functionality of the CSP algorithm used in ColorCSP
comm_	An object of *Communication Channel* is defined to implement the functionality of the *Communication Channel*
init()	Initialize the *Node*
push()	Invoked by *Communication Channel* component to notify the arrival of an incoming message.
get/set_attributes()	Access the attributes of *Node*

Figure 4.7: The Attributes/Methods of Node

3. Create connections actively to neighboring *nodes*.

4. Exchange *parameter messages* with neighboring *nodes*.

5. Run *CSP Solver* until a termination condition is met.

From above discussion, we can see each *node* process in ColorCSP has to implement quite complex communication tasks. To reduce the complexity in designing and implementing such kind of distributed application, *event-driven* model is used as our central design paradigm in ColorCSP. Furthermore, various event-handling design patterns provided by ACE are used as our basic building blocks, *e.g.*, Reactor, Connector, Acceptor etc. We will briefly discuss these design patterns before we present our system architecture.

Reactor [18]: The *Reactor* architectural pattern allows event-driven applications to demultiplex and dispatch one or more client requests. *Reactor* pattern includes following two layers [17]:

1. Event infrastructure layer that is responsible for implementing application-independent strategies for demultiplexing indication events to event handlers and then dispatching the relevant event handler hook methods. This layer includes various implementations of the ACE_Reactor and the ACE timer queue.

2. Application layer that implements concrete event handlers that implements application-defined processing in their hook methods. All application layer classes should be derived from ACE_Event_Handler class of ACE.

Acceptor-Connector [18]: The *Acceptor-Connector* design pattern decouples the connection and initialization of communication peers from the application-specific processing after the connection is established and initialized. ACE implements the following classes in this pattern [17]:

1. *ACE_Svc_Handler*: This class template represents the local end of a connected service and contains an IPC endpoint that communicates with the peer service handler.

2. *ACE_Acceptor*: This class template factory initializes an ACE_Svc_Handler object passively in response to an active connection request from a service peer.

3. *ACE_Connector*: This class template factory makes a connection to a peer acceptor actively and then initializes an ACE_Svc_Handler object that will communicate with the connected peer.

Acceptor-Connector design pattern normally collaborates with *Reactor* pattern to perform application-independent event dispatching. In application layer, the generic strategies of these patterns should be implemented(normally by inheritance or template instantiation) to perform application-specific processing.

As illustrated in Figure 4.8, two classes of ColorCSP, Node_Acceptor and Node_Svc_Handler subclass ACE_Acceptor and ACE_Svc_Handler respectively. To accept incoming connections, the *node* application first calls the initialization method of Node_Acceptor. This method creates a passive mode transport endpoint and binds it to a transport address. Then the Node_Acceptor listens on this transport address for incoming connection requests. In addition, Node_Acceptor registers itself to dispatcher(since this step is completed by ACE_Acceptor internally, the relevant classes, *e.g.*, dispatcher, are not shown in the figure). When another *node* makes an active connection request, the dispatcher notifies the associated acceptor. The Node_Acceptor then creates a new service handler, Node_Svc_Handler and stores the connection handle into it. Finally, Node_Svc_Handler will read the incoming data from the connection and completes the communication with the remote *node*.

4.5.2 Design Choices on Concurrency Architecture

Multithreading has been widely used in distributed applications to improve the overall system throughput and scalability. However, choosing an appropriate multithreading model for a application needs careful analysis of the workload conditions and system requirements. In this section, we discuss two design choices on our concurrency architecture for ColorCSP. Both approaches are based on widely used ACE concurrency patterns, namely Half-sync/Half-async and Leader/Followers [18].

Half-sync/Half-async Pattern: The Half-sync/Half-async architectural pattern decouples asynchronous and synchronous service processing in concurrent distributed systems, while simplifies programming without impairing performance. This pattern includes two layers: asynchronous and synchronous service processing. A queuing layer exists between the two layers and buffers the events that are passed from upstream asynchronous layer to synchronous layer.

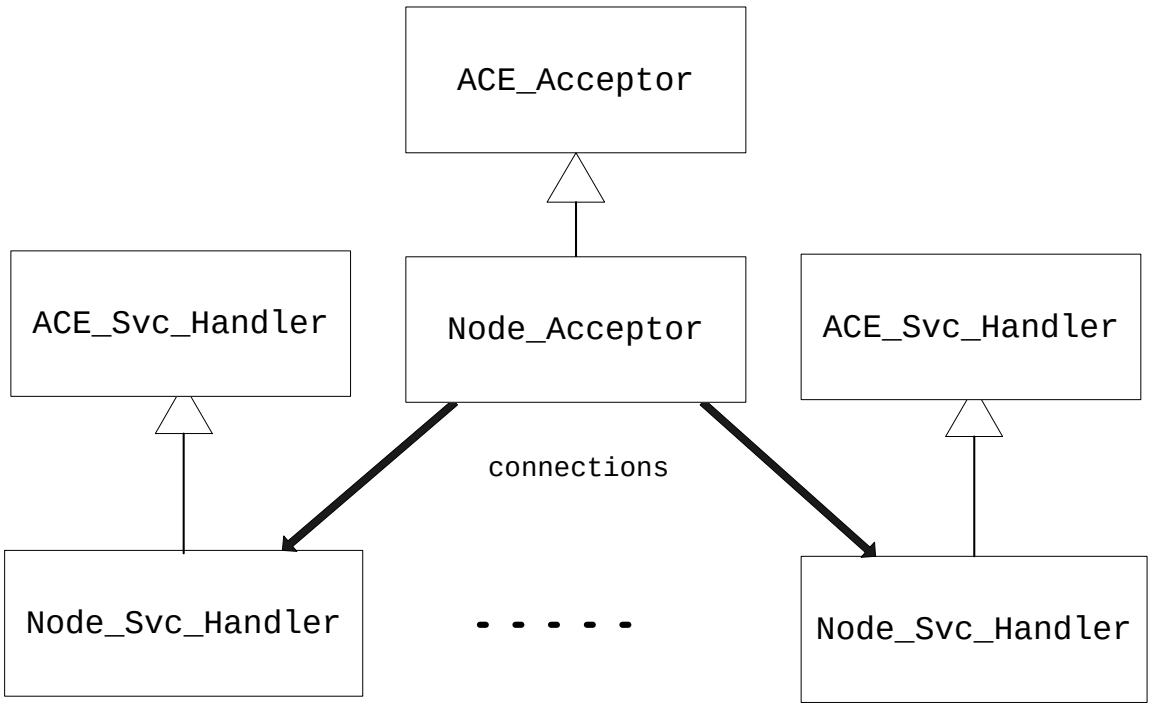


Figure 4.8: Applying ACE Event-handling Patterns

ACE_Task includes a message queue that can be accessed by the external method invocations as well as the internal thread. Thus the queue of ACE_Task serves the queuing layer in Half-sync/Half-async pattern.

Figure 4.9 depicts the *node* architecture based on Half-sync/Half-async pattern. The *parameter messages* arrive asynchronously from multiple neighboring *nodes* and then are buffered in a message queue. A thread then retrieves the *parameter messages* from message queue and process them synchronously. The sequence of events in processing of a *parameter message* is:

- *Node* creates an instance of Node_Task that implements the functionality of thread by subclassing ACE_Task.
- A neighboring *node* makes a connection.
- The Node_Acceptor accepts the incomming connection request and creates a Node_Svc_Handler that completes the subsequential communication process.
- Within the handle_input() method, Node_Svc_Handler reads a *parameter message* from the connection and enqueue it to Node_Task's message queue.

- In the entry method of Node_Task thread, a *parameter message* is read from the message queue.
- When *parameter messages* from all neighboring *Nodes* are read, node_Task will invoke *CSP Solver* to perform one step of DBA algorithm.

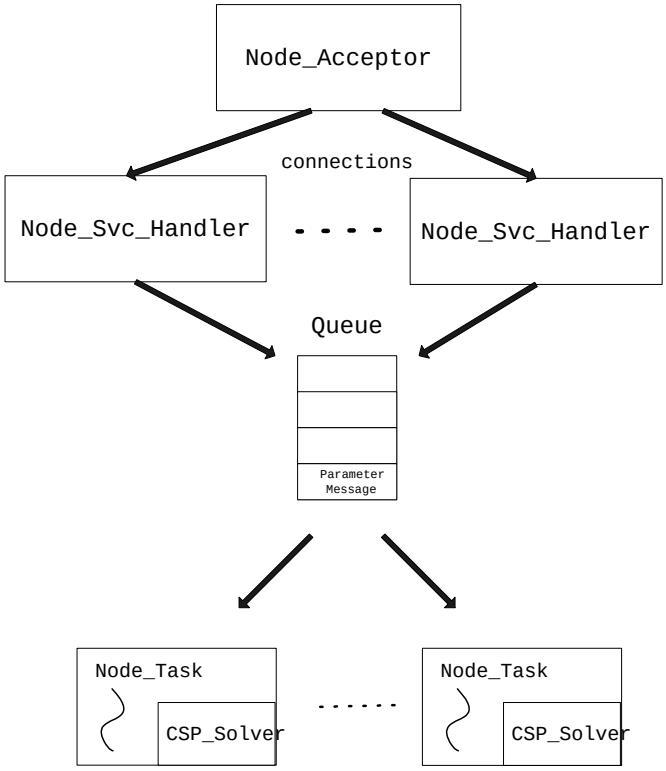


Figure 4.9: Half-sync/Half-async Concurrent Architecture

The Half-sync/Half-async concurrent pattern simplifies the programming in synchronous processing layer while preserving the high-performance of asynchronous lower-level layer. In addition, the separation of concerns provides more flexibility of choosing concurrency policies at different layers.

Leader/Followers Pattern: The Leader/Followers architectural pattern provides a thread pool mechanism that allows a number of threads to coordinate themselves and share the workload of detecting, demultiplexing and processing the service requests. In Leader/Followers pattern, one thread at a time waits for an event to occur on a set of event sources. This thread is the *leader* and other threads - the *followers*

- wait for their turn to be the leader. When the current leader detects an event on a set of event sources, it first promotes a follower to be the leader and then dispatches the event to a *event handler* that performs the application-defined event processing.

The ACE_TP_Reactor class in ACE implements the Leader/Followers architectural pattern. Multiple threads in a thread pool take turns calling select() on sets of I/O handles to detect, demultiplex, dispatch, and process incoming service requests. When an event happens on a connection handle, *e.g.*, the data is available to read, in the handle_events() method of current *leader*, the event is dispatched to the handle_input() method of associated event handler.

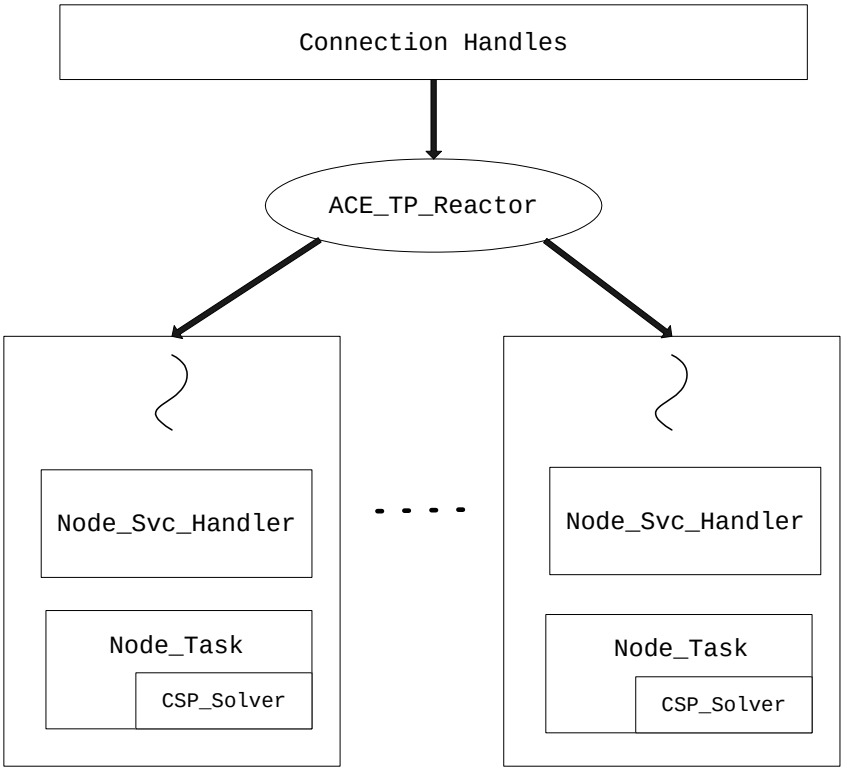


Figure 4.10: Leader/Follower Concurrent Architecture

Figure 4.10 depicts the *node* architecture based on leaderfollower pattern. A number of threads are spawned by ColorCSP application to process incoming messages. One thread will serve as *leader* and wait for incoming connections by registering a Node_Acceptor(not shown in Figure 4.10) with ACE_TP_Reactor. The sequence of events in processing of a *parameter message* is:

- A neighboring *node* makes a connection.

- The current *leader* thread will promote a new *leader* from thread pool and a new Node_Svc_Handler is created to complete the subsequential communication.
- Within the handle_input() method, Node_Svc_Handler reads a *parameter message* from the connection and pass it to Node_Task.
- Node_Task buffers the *parameter message*.
- When *parameter messages* from all neighboring *nodes* are read, Node_Task will invoke *CSP Solver* to perform one step of DBA algorithm.

In the design with Leader/Followers pattern, the processing of a *parameter message* happens within one thread, the context switch between threads is not needed thus event dispatching latency is reduced. Furthermore, the Leader/Followers concurrent pattern can minimize locking overhead by eliminating the data exchange between threads.

4.6 Design with nORB and TAO

The nORB version and TAO version of ColorCSP share the same design, since nORB and TAO use the same programming model.

4.6.1 Defining the Events

There are two types of events in ColorCSP:

- *Node registration*: When a *node* registers itself with *Constraint Factory*, the *node* sends the information about itself and its constraints to *Constraint Factory*. A CORBA sequence *NodeAttr* is defined to store the registration information. The IDL definition is shown as follows:

```
struct NodeAttr_s
  {
    // The IOR of the node
    string IOR;
    // The symbol of the node
    string symbol;
    // The current color value
```

```
  long color_val;
  // The impr of the node
  long impr;
  // The id of the node
  long src_id;
  // The weight of the constraint with which the node is
  // associated
  long weight;
  // The my_termination_counter of this node
  long mtc;
};

typedef sequence < NodeAttr_s > NodeAttrSeq;
struct NodeAttr{
  // The characteristics of itself
  NodeAttr_s self;
  // The constraints list of the node
  NodeAttrSeq constrns;
};
```

- *parameter message*: Each *node* of ColorCSP runs a *CSP Solver* and intermediate evaluation parameters of *CSP Solvers* are exchanged between neighboring *nodes* in the format of *parameter message*. The IDL definition of *parameter message* is shown as follows:

```
enum _EventType
{
  Value,
  DomSize,
  Impr,
  Terminate
};

struct Event
{
```

```
    // The type of event
    _EventType type;
    // The data according to the type
    long data;
    // The status of sending node
    long flag;
    // The my_termination_conuter
    long mtc;
    // The id of the node
    long src_id;
  };
```

4.6.2 Defining *Constraint Factory* and *Node*

The only interface provided by *Constraint Factory* is *register_node* that is called by a *node* to register itself. As discussed in 4.6.1, a *node* passes a structure that includes its Interoperable Object Reference(IOR) to *Constraint Factory*. When all *nodes* complete registrations, *Constraint Factory* calls the *start_eval* interfaces of registered *node* CORBA objects to start the executions of *nodes*.

To send a *parameter message* to a neighboring *node*, a *node* calls the *push* interface with the *parameter message* as CORBA *in* parameter. Within *push* method, the *node* buffers the incoming *parameter message* and invokes *CSP Solver* to perform a step of DCSP algorithm after a *parameter message* is received from each neighboring *node*.

The IDL definitions of *Constraint Factory* and *Node* are shown as follows:

```
//A factory class for the Color Node interfaces
interface Constraint_Factory
{
  /**
   * Node will call this to register itself.
   * @param NodeAttr attr The attributes struct
   * the node registers with NodeFactory
   */
  oneway void register_node (in NodeAttr attr);
```

```
};

interface Node
{
  /**
   * Called by peers to push a message
   * @param Event msg Input msg from peer
   */
  oneway void push (in Event msg);
  /**
   * Call back by constraint factory and start
   * the evaluation
   * @param long steps The steps of evaluation
   * @param NodeAttr attr The attributes struct
   *  that will be set to the node
   */
  oneway void start_eval (in long steps,
                          in NodeAttr attr);
};
```

4.7 Design with TAO RT Event Channel

In this section, we present the design of ColorCSP based on TAO Real-time Event Service [14].

4.7.1 OMG Event Service and TAO Real-time Event Service

The OMG Event Service provides a decoupled communication model rather than the traditional client-server synchronous request invocations. It defines the basic interfaces for consumers and suppliers to subscribe to an event channel that serves as a *mediator* for the subsequential communicatations between consumers and suppliers.

However, OMG Event Service lacks several key features for large-scale applications with real-time requirements, *e.g.*, real-time guarantee for event delivery and scheduling, event filtering and correlation etc. The TAO Real-time Event Service

extends standard OMG Event Service by providing many key features to satisfy the QoS requirements of real-time applications.

4.7.2 System Architecture Based on TAO RTEC

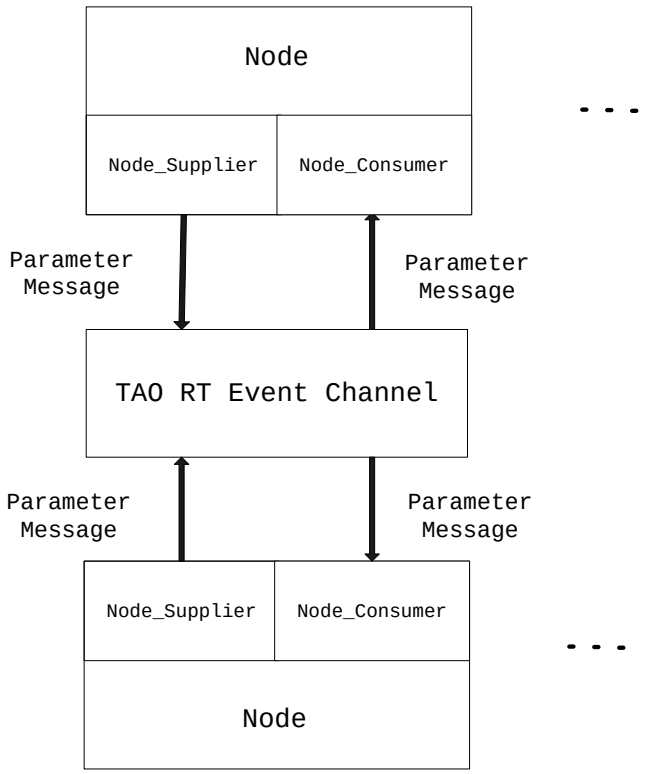


Figure 4.11: System Architecture Based on TAO RTEC

Figure 4.11 illustrates the system architecture based on TAO RTEC. *Node* includes two classes Node_Supplier and Node_Consumer, which are derived from TAO RTEC supplier and consumer classes respectively.

When a *node* initializes, it looks up the object reference of RTEC from TAO Naming Service. It then invokes the *connect* method of Node_Supplier to connect to TAO RTEC as event supplier. In connecting to TAO RTEC, the Node_Supplier can specify a set of QoS parameters including the source ID, event type etc. Then *node* invokes the *connect* method of Node_Consumer to connect to the TAO RTEC as event consumer. Since a *node* only needs to communicate with neighboring *nodes*, a consumer *node* can take advantage of TAO RTEC's event filtering by specifying

the event sources when connect to event channel. TAO RTEC then only delivers the interest data to the consumer *node*. This feature can reduce the overhead of message delivery.

4.7.3 Configurations of TAO RTEC

As discussed in the previous section, only one event channel is used in the system. Although this configuration doesn't incur the complexity of managing multiple event channels, the system performance doesn't scale well when the the number of *nodes* increases. To address this limitation, TAO RTEC also supports other configurations.

- Multiple Event Channels: Since a *node* only communicates with neighboring *nodes*, one RT event channel can be configured for one *node* or a neighborhood of adjacent *nodes*. This configuration will improve the system throughput when the number of *nodes* is small. However, the complexity of managing multiple event channels might be very high for a large-scale application. In addition, a *node* must communicate with separate event channels, thus extra overhead is introduced.

- Federating Event Channels: Federation of event channel could avoid some above problems. When separate event channels are joined into a federation, the resulting federated event channel acts as one logical event channel. This configuration allows the communication particants to interact with only one logical event channel. It also improve the system performance by delivering events locally when possible. TAO Real-time Event Service provides several mechanisms for federating event channels, including CORBA gateway, UDP and IP multicast.

Chapter 5

Experiments

This chapter presents our experimental results. Our goals are to

1. Quantify the implementations of ColorCSP on various DOC middleware platforms.

2. Validate the design of small footprint ORB implementation -nORB- by comparing the performance of nORB with other existing middlewar platforms. ColorCSP implementations are used as benchmark applications to evaluate the performances of nORB and other DOC middleware platforms.

We first present the experimental setup. The experiments we performed fall into two categories, namely application-level and fine-grain experiments. Application-level experiments aim to compare the overall performances of ColorCSP implementations on various DOC middleware platforms. Fine-grain experiments are performed to investigate the performances of low-level system mechanisms(*e.g.*, ORB-level operations), and explain the results observed in application-level experiments.

5.1 The Experimental Platforms

Taking the advantages of platform-independency provided by DOC middlewares, *e.g.*, ACE, nORB and TAO, the implementations of ColorCSP can be ported to different OS platforms easily. All experiments were conducted on a 4-machine cluster of Pentium 4 2.53GHz CPUs, each with 512MB RAM. The configurations of OS/DOC middlewares/compilers are listed in Figure 5.1.

KURT Linux is chosen as our experimental platform because of its support for micro-second-level high-resolution timing capacity.

OS	Compiler	DOC middlewares
KURT Linux 2.4.18	gcc 3.2	ACE 5.2.8 nORB TAO 1.2.8

Figure 5.1: Configurations

The high-resolution timer of ACE is used to measure the elapsed time of critical operations of ORB and ColorCSP.

5.2 Application-level Experiments

5.2.1 Experimental Setup

Figure 5.2 shows the setup that we used for 100 node experiments.

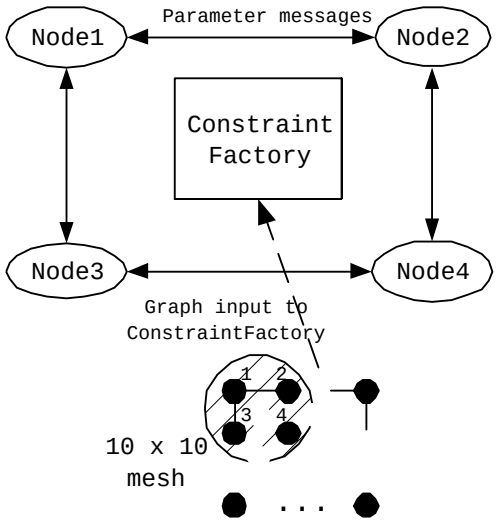


Figure 5.2: Performance Measurement Infrastructure

ConstraintFactory process opens a graph definition file and reads the topology of a graph for which we need to find a valid color assignment. In our experiments we used a 10x10 mesh of 100 Nodes, a representative sensor network topology for the Boeing OEP. Each Node represents a distributed vertex of the graph.

A group of 25 Node processes was executed on each of the 4 machines and the *ConstraintFactory* was executed on one of the machines. In Figure 5.2 only the interactions between nodes in a four node region are shown. A Node communicates with its neighbors by sending *parameter messages*. There are two types of *parameter messages*: 1) Value messages, containing the current color assignment of the sending node, and 2) Improvement messages, containing the maximal reduction in conflicts that could be achieved by a color change at the sending Node.

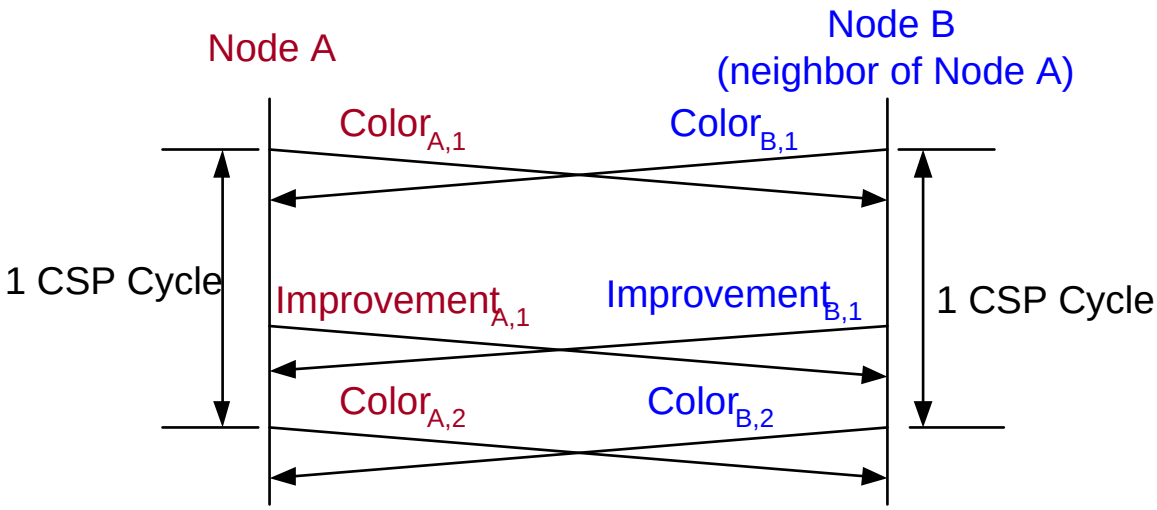


Figure 5.3: One DBA Timing Cycle

To meet the stringent performance requirements of real-time applications, we explored various optimization techniques for TAO and nORB. In particular, the following versions of nORB and TAO are used in our experiments: 1) unoptimized (default), 2) with static compiler optimizations, and 3) with runtime optimizations. For the runtime optimized versions, TAO is optimized for ColorCSP implementation via single-thread and one way function call settings, whereas nORB is optimized using the critical path techniques discussed in Section 5.2.3.

5.2.2 Performance Metrics

The following metrics were used to evaluate the performance of nORB in comparison to TAO and ACE.

Elapsed cycle times: The elapsed time for one cycle of *CSP Solver* running the DBA algorithm is the fundamental measurement in our experiments. A node has to wait for *parameter messages* from *all* its neighbors in each cycle of the DBA algorithm before it proceeds to the next cycle. Thus, a small delay in one cycle of a node will be amplified and propagated to its neighbors in the following cycles. This metric's sensitivity to delay was a major factor leading to identifying the performance variations discussed in Section 5.2.4.

Convergence times: Convergence of the DBA algorithm for a given network topology is quantified by the number of cycles needed before a global solution is reached. However as noted above, variation in timing of individual algorithm steps can have a significant impact on the overall performance. Therefore, we measure the total time for the algorithm to converge over multiple repeated runs, to assess the relative overall impact of using ACE, nORB or TAO in the ColorCSP.

5.2.3 Critical Path Optimization

To achieve remote method invocation performance that is comparable with TAO, we first identified the critical path through the ORB while making a remote call. On the client side, the critical path consists of the following actions in sequence:

1. marshaling the remote call parameters into a request message
2. sending the request message through a TCP socket
3. receiving the reply from the server
4. demarshaling the return values from the reply message

Similarly on the server side, the critical path consists of the following:

1. receiving the request from the client
2. demarshaling the parameter values from the request message
3. making the operation upcall, marshaling the call return values into a reply message
4. sending the reply message through TCP socket back to the client

Next, we took timing measurements at key checkpoints along the critical path in both nORB and TAO. The check points were:

1. when the client makes a remote call
2. when the connection to the server is established
3. when the server receives the request
4. when the server dispatches the request to the remote object

Problem: Unnecessary system calls in the critical path result in reduced performance. We found that on the server side, reading data from different client connections was unnecessarily interleaved in nORB and hence the latency between the first and the last read operation for one request was very high. Normally only one read operation is necessary to receive a request from the client. The ORB dispatches an upcall to a servant only after the whole request is read, and hence the delay incurred by multiple read operations hurts the overall performance of the server. In comparison, TAO does not exhibit this overhead and receives each request in a single read operation.

After further investigation, we traced the problem to the inefficient manner in which requests were being sent on the client side of nORB. When nORB handled a request from the client side application, two write operations were being used to send a GIOP request to the server. The data written by the second write operation could be buffered in TCP layer while the first data chunk of the request has already been delivered to the server. Thus the GIOP request was segmented and its receipt at the server spanned multiple read operations. This effect worsened when other requests came to the server before the arrival of the later chunk of a segmented GIOP request.

We also found that reading a request on the server was done using two *recv* system calls on the connection stream - first for reading the request header and the second one for reading the request body, whereas the read operation could be done using a single *recv* call for relatively small payloads. This was done despite the request body already being present in the TCP buffer.

Solution: Use single read/write operation We achieved a single write operation on the client side by applying the well-known *gather-write* [19] technique. On the server side, we optimized reading a request so that if possible the request header and

body are read using a single *recv* call. While this solution is obvious in hindsight, the problems were not detected during the initial development of nORB. Rather, this problem first came to light during careful timing measurements performed to compare nORB and TAO in our ColorCSP application. Given the myriad features within even a reduced middleware framework, it is therefore essential to perform significant performance testing along the critical path of special purpose middleware and of related general purpose middleware for comparison, *during the development process itself.*

5.2.4 The Experimental Results

Footprints

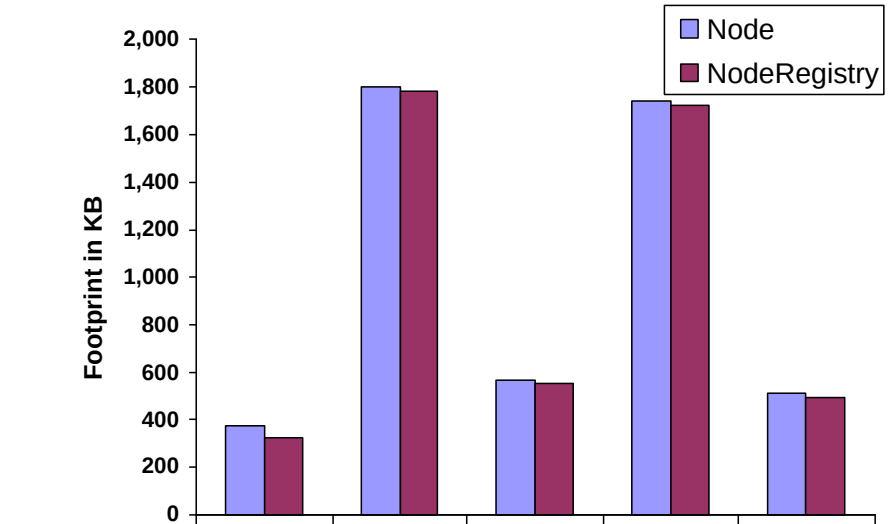
To validate the nORB's design goal of achieving the footprint reduction by only adopting the necessary set of features, the footprints of ColorCSP implementation on various DOC middleware platforms were measured.

Figure 5.4 [21] shows the footprint reductions achieved by nORB. All measurements were taken using the *size* command. The application specific code in Node and *ConstraintFactory* take about 164KB and 146KB respectively. From our measurements, we found that ACE introduces an footprint overhead of about 212KB, and unoptimized versions of TAO and nORB introduce an additional overhead of 1424KB and 1911KB respectively. Compile optimization of TAO and nORB reduces the added overhead of the ORB layer to 1362KB and 133KB respectively.

Performances of Concurrent Design Patterns

Figure 5.5 shows the the distribution of measured cycle times over ~500,000 cycles of ColorCSP implementations using ACE Leaders/Followers and Half-sync/Half-async concurrent design patterns. Single thread is used in Leaders/Followers pattern.

We can see that the implementation using Leaders/Followers pattern takes $\sim 6,500$ miro-seconds to complete a CSP cycle while Half-sync/Half-async pattern implementation takes $\sim 10,000$ micro-seconds. This result is consistent with our expectation since Half-sync/Half-async pattern has the extra overhead of a thread context switching in processing a *parameter message*. Furthermore, since the system throughput is very high, the Half-sync/Half-async pattern incurs buffering delay because of the queue layer between synchronous and asynchronous layers.



	ACE	TAO	nORB	compile optimized TAO	compile optimized nORB
Node	376	1800	567	1738	509
NodeRegistry	324	1778	549	1725	492

Figure 5.4: Footprints Comparison

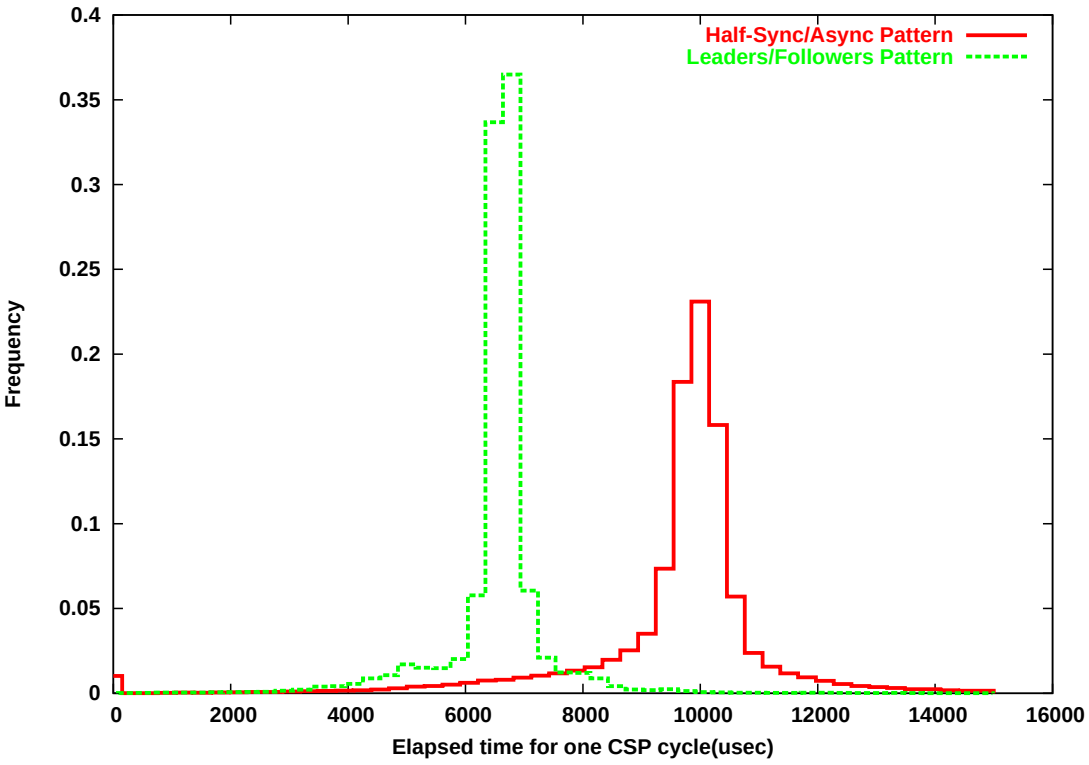


Figure 5.5: Performance of Different Concurrent Patterns

In the following experiments for comparing ColorCSP implementations on various DOC middleware platforms, the ColorCSP implementation using ACE Leaders/Followers pattern is used.

Results of ACE, nORB and TAO

Elapsed cycle times: Figure 5.6 andFigure 5.7 show the distribution of measured cycle times over ∼500,000 cycles of the *CSP Solver* using ACE, nORB and TAO, up to a 50msec limit that includes 95% of all samples in each case. Measurements over 50msec are considered in the next metric.

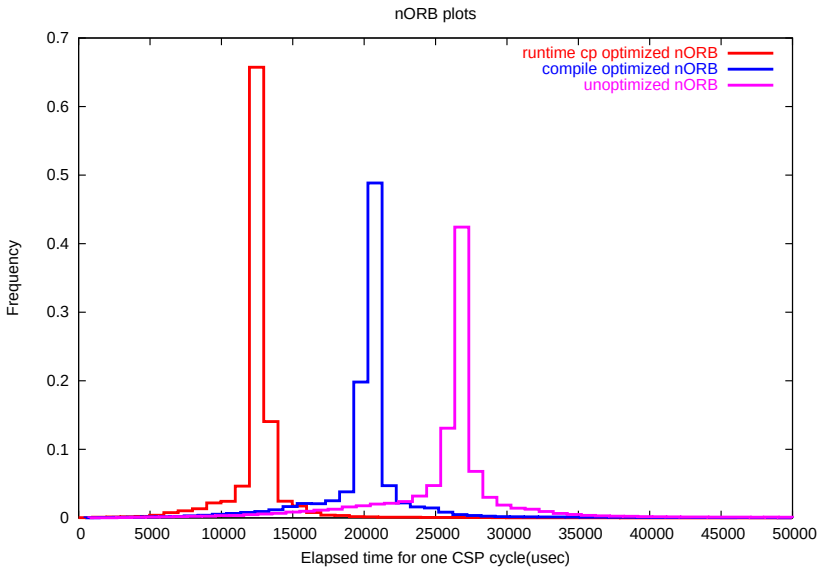


Figure 5.6: Performance of ACE and nORB Configurations

We see that each optimization is effective in improving performance. Furthermore, the effects of the optimizations are more obvious for TAO, since the default configuration of TAO is aimed for multi-threaded general purpose applications. In addition, we see that one cycle of *CSP Solver* takes similar time on average, ∼12 msec, using the runtime optimized versions of nORB or TAO with average performance marginally better for TAO.

Convergence times: Finally, Figure 5.8 shows the total convergence times for the *CSP Solver*, running on ACE, TAO, and nORB. As may be expected from the previous cycle time and time bound figures, ACE outperforms both nORB and TAO,

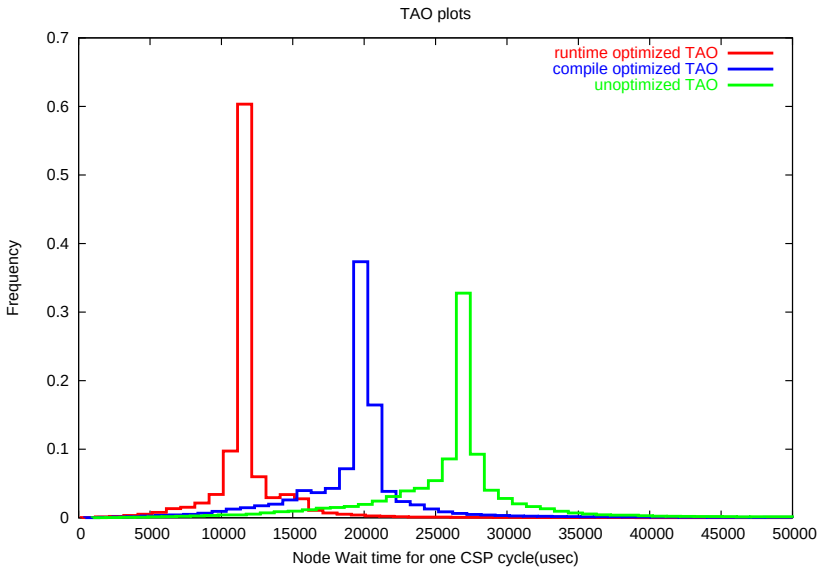


Figure 5.7: Performance of TAO Configurations

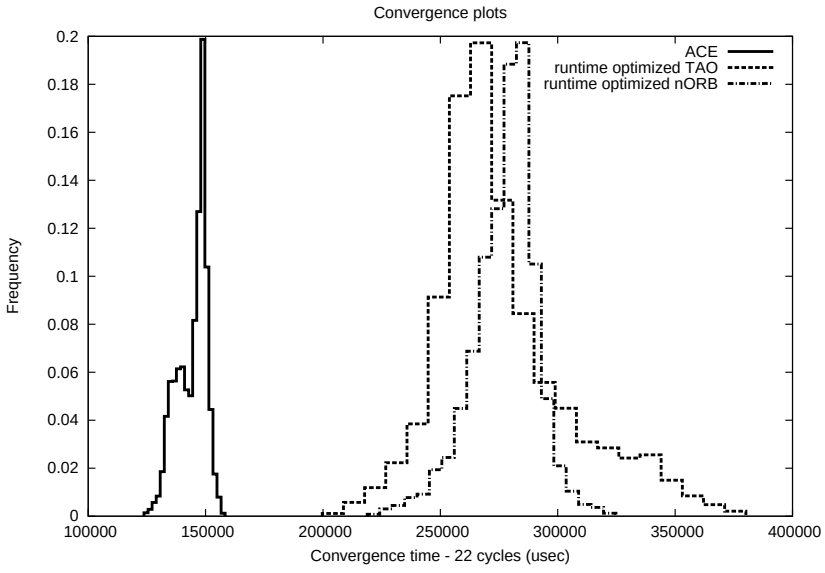


Figure 5.8: *CSP Solver* Convergence on 10x10 Mesh

performing much better in both the average and worst cases. TAO slightly outperforms nORB in the average case, while nORB performs better in increasingly worse cases.

The application-level experiments performed in this section have shown that the application-level performance of ColorCSP implementation using nORB is comparable with that using TAO after applying various optimization techniques. ACE implementation outperforms both nORB and TAO. To understand the implications from these performance results, we perform various ORB level fine-grain experiments in the next section.

5.3 Fine-grain Experiments

In this section we describe a set of experiments conducted to quantify fine-grain middleware performance.

5.3.1 Sequence of ColorCSP Operations

Figure 5.9 illustrates the sequence of operations on each of two connected nodes in ColorCSP, and shows the messages exchanged between them. Both application-level and ORB-level operations are shown in the figure.

Each *Node* performs the following sequence of steps in each *CSP Solver* cycle:

1. it marshals its *color* value,
2. sends a color value message to each neighbor,
3. waits for color value messages from its neighbors,
4. receives each neighbor's color value message,
5. looks up and dispatches each message's method,
6. demarshals the color value from each message,
7. decides its best improve value by running *DBA*,
8. marshals its new *improve* value,
9. sends an improve message to each neighbor,

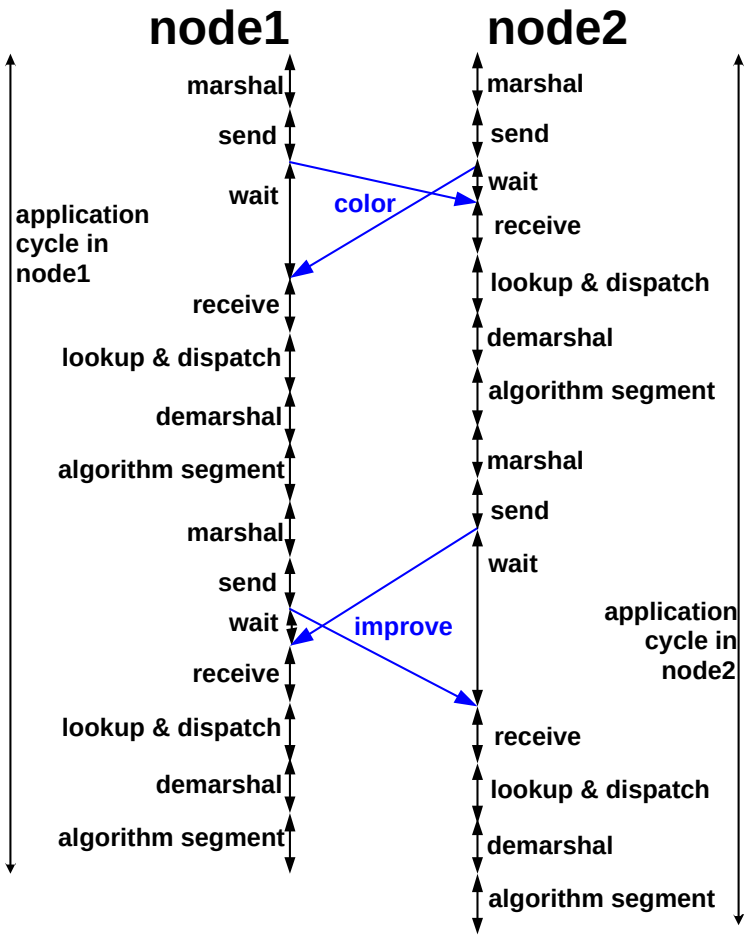


Figure 5.9: *ColorCSP* Timing Sequence

10. waits for improve value messages from its neighbors,
11. receives each neighbor's improve value message,
12. looks up and dispatches each message's method,
13. demarshals the color value from each message, and
14. decides its new color value by running *DBA*.

As illustrated by Figure 5.9, some steps are synchronous within a process, *i.e.*, steps 1, 5-8, and 12-14, other steps are asynchronous, *i.e.*, steps 2-4, 9-11. Furthermore, as Figure 5.9 illustrates this asynchrony can lead to variations in cycle times, both between and within nodes.

5.3.2 Experimental Setup

Four fully connected nodes are used in the experiments. Each node is on a separate machine and the other 3 nodes are its neighbors. We also performed the same experiments with 2 nodes, each on its own machine and with the other node as its sole neighbor, to study fine-grain communication phasing effects between nodes.

5.3.3 Experimental Methodology

To analyze the performance of *ColorCSP* using ACE, TAO, and nORB, we measure the time in each of the steps enumerated in Section 5.3.1 along the end-to-end messaging path for ACE, TAO, and nORB, in addition to *elapsed cycle time* used in application-level experiments.This kind of fine-grain measurements allows us to construct a complete timing profile of combined application and mechanism level operations. It proves that this approach is very helpful in understanding the performance variations of different middleware platforms.

Two timers are used in the fine-grain measures:

- an application-level timer to measure the time taken for one application-level cycle, and
- a mechanism-level timer to measure the time taken for the different stages of the middleware layer.

The following paragraphs describe our approach to each of several crucial issues, which we addressed to ensure accuracy and reproducibility of our results.

Application-level Instrumentation: We used each node's application-level timer to measure each application *cycle* as described in Section 5.3.1. Immediately before a node started to send messages to its neighbors, its application-level timer was started, along with its mechanism-level timer for the first middleware segment. Specifically, both timers were started in each node just before the *marshal* stage commenced at the beginning of the timeline shown in Figure 5.9. The application-level timer was stopped and restarted after each complete application cycle. To achieve full timing coverage across all cycles with no intervening gaps, we stopped the application-level timer just before starting it.

Mechanism-level Instrumentation: At each mechanism-level checkpoint, *i.e.*, between each of the segments shown in Figure 5.9, the mechanism-level timer was stopped, the elapsed time logged in-memory and the timer was started again. As with the application-level timer, we stopped, measured and started the mechanism-level timer contiguously in our measurements, thus ensuring that we did not leave any gaps in the mechanism-level measurements. Hence we ensured reasonably full accounting of the time spent during an entire application-level cycle.

To further verify that our timing measurements offered full coverage of a message path, we computed the total time taken for a cycle based on summation of the individual measurements and compared these results to the actual observed cycle time measurements. We verified that those two results matched closely, which indicated that our timing measurements offered reasonably full accounting. We also tagged each of the mechanism-level measurements with a segment identifier, so that related sequences of mechanism-level measurements could be correlated with the overall application-level measurements. This helped immensely in pinpointing the cause of middleware behavior causing higher cycle times in TAO, as described in Section 5.3.8.

Timer operations were all inline and used pre-allocated memory to avoid instrumentation overhead interfering significantly with actual system performance. Figure 5.12 in Section 5.3.5 shows performance of the 100 node configuration with full timer instrumentation, which is nearly indistinguishable from the similar plot without instrumentation originally presented in 5.2. This result indicates that the effect of instrumentation was minimal, as designed.

Scale of Experiments: We conducted identical experiments with the *ColorCSP* using ACE, TAO, and nORB, and across several different node configurations. The

first, using only two nodes on two separate machines was designed to minimize the coupling between nodes, with each node having only a single neighbor and with contention both minimal and limited to only the network resource. The second node configuration used four nodes, each again on its own machine, but with three neighbors each. Finally, we ran the original 10 by 10 hundred node mesh, with 25 nodes running on each of 4 machines as described in Section 5.3.2. The point of varying the node configurations was to isolate factors of node interaction and resource contention. To avoid misinterpreting artifacts of network, platform or other experimental noise, we repeated each experiment over roughly 500,000 cycles of the *CSP Solver*, producing consistent and repeatable distributions of measured timing.

5.3.4 Overall Performance Results

Figure 5.10 shows the mean and median performance values for the application cycle times and individual segments, for each implementation running in 100, 4, or 2 nodes. Of particular interest is that even with the TAO optimization described in Section 5.3.8, the mean and median application cycle times for nORB show a slight performance improvement over TAO. Furthermore, this effect is weaker with more nodes and stronger with fewer nodes. This effect correlates most strongly with differences in the mean and median wait times, which also show larger differences with fewer nodes, and smaller differences with more nodes. Section 5.3.8 discusses the implications of these wait time results in detail.

	ACE 100 nodes	nORB 100 nodes	TAO 100 nodes	ACE 4 nodes	nORB 4 nodes
cycle	6928/6978	13015/13210	13687/12667	228/227	385/384
marshal	4/5	13/10	18/9	5/3	11/11
send	40/5	75/7	90/7	6/3	8/5
wait	906/534	1703/1190	1764/788	16/16	15/16
receive	11/9	12/11	9/7	10/9	9/6
lookup &dispatch	0/0	15/12	12/20	0/0	17/15
demarshal	1/0	0/0	0/0	1/0	0/0
algorithm execution	1/0	1/1	1/1	0/0	0/0

Figure 5.10: Fine-grain Mean/Median Timing (in μsec)

	ACE 2 nodes	nORB 2 nodes	TAO 2 nodes	TAO 4 nodes
cycle	128/128	190/189	165/164	411/410
marshal	2/2	13/16	12/12	11/9
send	3/1	5/5	5/5	8/7
wait	51/52	52/52	50/46	29/20
receive	5/5	6/6	7/5	9/8
loopup &dispatch	0/0	15/13	4/4	7/4
demarshal	1/1	0/0	0/0	0/0
algorithm execution	0/0	0/0	0/0	0/0

Figure 5.11: Fine-grain Mean/Median Timing (in μsec) Contd.

ACE performs better on average (both mean and median) than either nORB or TAO in overall cycle times, marshaling, send, and wait times. TAO's highly optimized receive mechanism outperforms those for both ACE and nORB. TAO's lookup and dispatch mechanisms are similarly optimized and outperform nORB: the implementation using ACE does not perform these functions but rather demarshals directly from the socket receive. Finally, demarshaling and algorithm segment times are negligible due to the relative simplicity of these mechanisms. We now turn our attention from average performance values to detailed performance *distributions* in the rest of this section.

5.3.5 Cycle Times

Figure 5.12 shows the distribution of measured cycle times over ~500,000 cycles of the *ColorCSP* using ACE, nORB and TAO, up to a 25 msec limit that includes 98% of all samples in each case.

While These measurements were taken with the additional fine-grain timing instrumentation in place , they match very closely with the results presented in our previous application-level measurements without the fine-grain instrumentation in place. These results thus give evidence that our instrumentation of the middleware had only a limited effect on the performance of the system.

The results shown in Figure 5.12 is also consistent with the overall performance obtained from Figure 5.10.

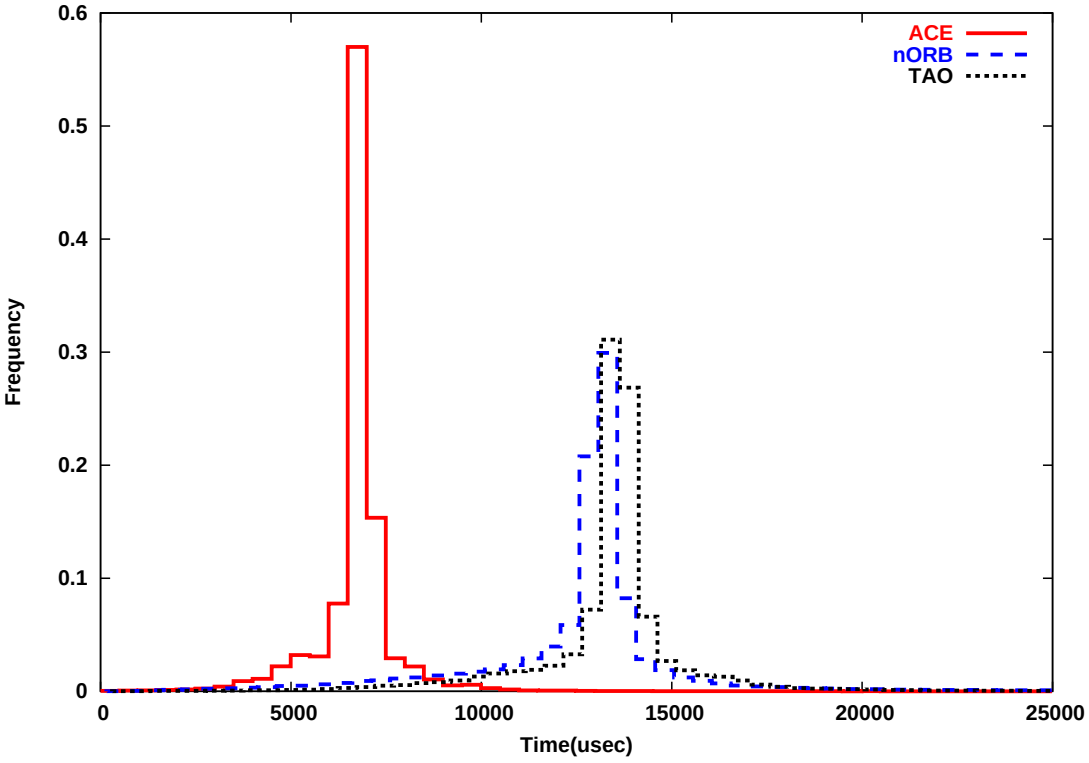


Figure 5.12: Cycle Times: 100 Nodes

5.3.6 Marshaling

Figure 5.13 shows the Marshaling times for ACE, TAO, and nORB configurations with 100 nodes. The distributions shown in Figure 5.13 reinforce the overall impression we get from Figure 5.10.

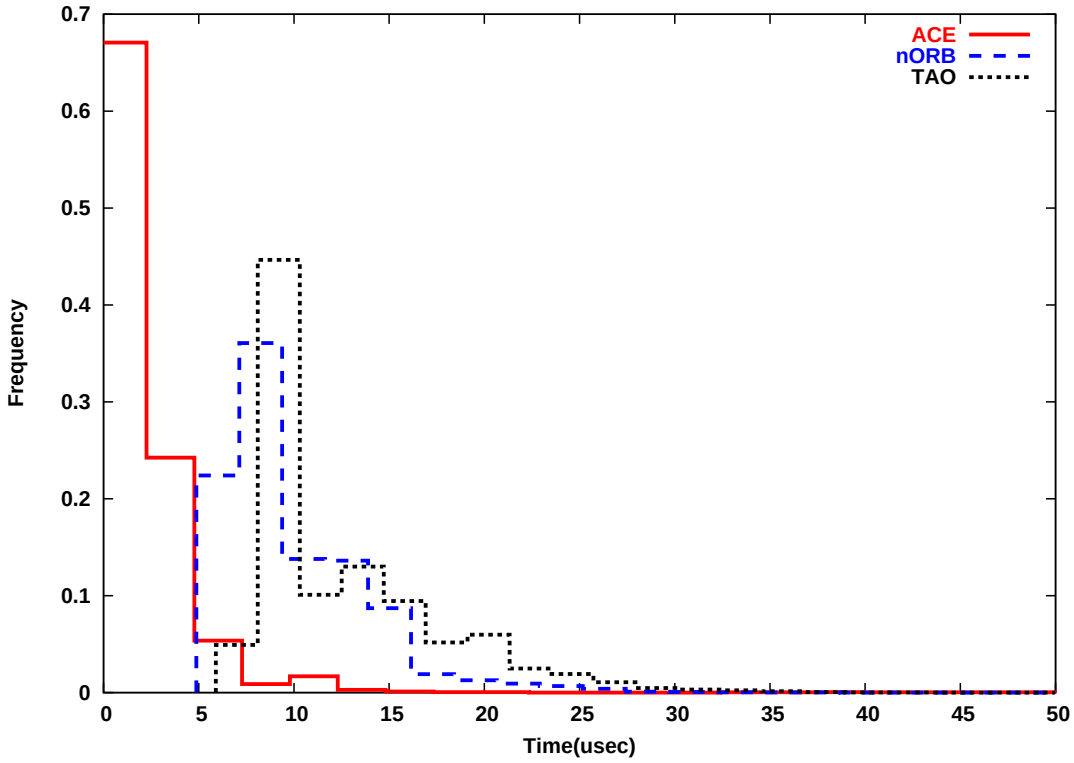


Figure 5.13: Marshaling Times: 100 Nodes

We can see that the overhead of marshaling a request and writing it to the connection are very similar for nORB and TAO, which is an expected result because both nORB and TAO use ACE's Common Data Representation (CDR) class to marshal their requests. Marshaling in the ACE implementation of *ColorCSP* outperforms that for both nORB and TAO, which can be explained by two reasons:

1. In the ACE implementation, a node only marshals each message itself, whereas nORB and TAO marshal both the message header and message body.

2. The ACE implementation only constructs one message in each cycle, even if it will be sent to multiple neighbors.

The results shown in Figure 5.13 thus correlate strongly with our understanding of the underlying marshaling mechanisms in ACE, TAO, and nORB.

5.3.7 Lookup and Dispatching

Figure 5.14 shows the times for servant lookup and method dispatching on the server side in TAO and nORB, running on 4 nodes.

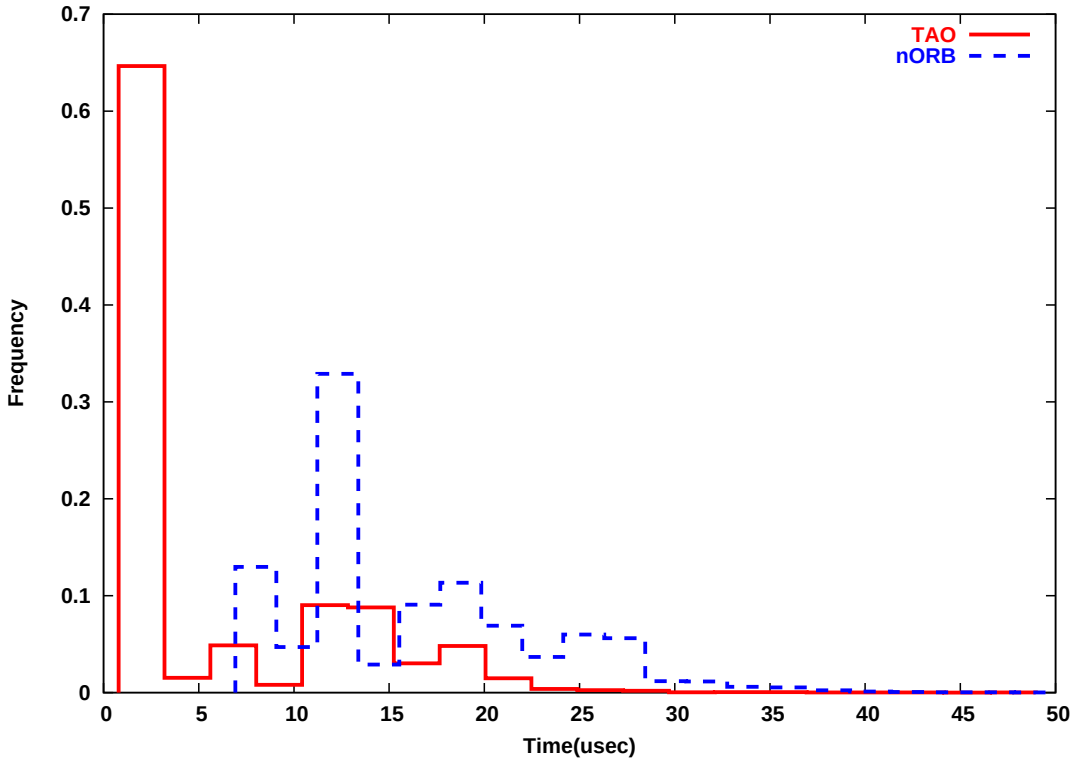


Figure 5.14: Lookup and Dispatching Times: 4 Nodes

We show this case in detail because it reflected the greatest difference in lookup and dispatching times between nORB and TAO, as shown in Figure 5.10. Times for ACE are not shown in Figure 5.14, as a node in ACE implementation does not perform lookup or dispatching and the times measured were negligible as expected, which is consistent with the results of Figure 5.10.

5.3.8 Wait Times

Figure 5.15 shows the wait time distributions for TAO with 2-nodes, first with the `SYNC_NONE` messaging policy, and then with the better-suited `SYNC_TRANSPORT` policy. `SYNC_NONE` was originally used by the TAO *CSP Solver* implementation because its use was intended to improve message throughput. However, our experiments showed that for lightly loaded networks with stringent performance requirements, TAO's default `SYNC_TRANSPORT` policy is strongly preferred.

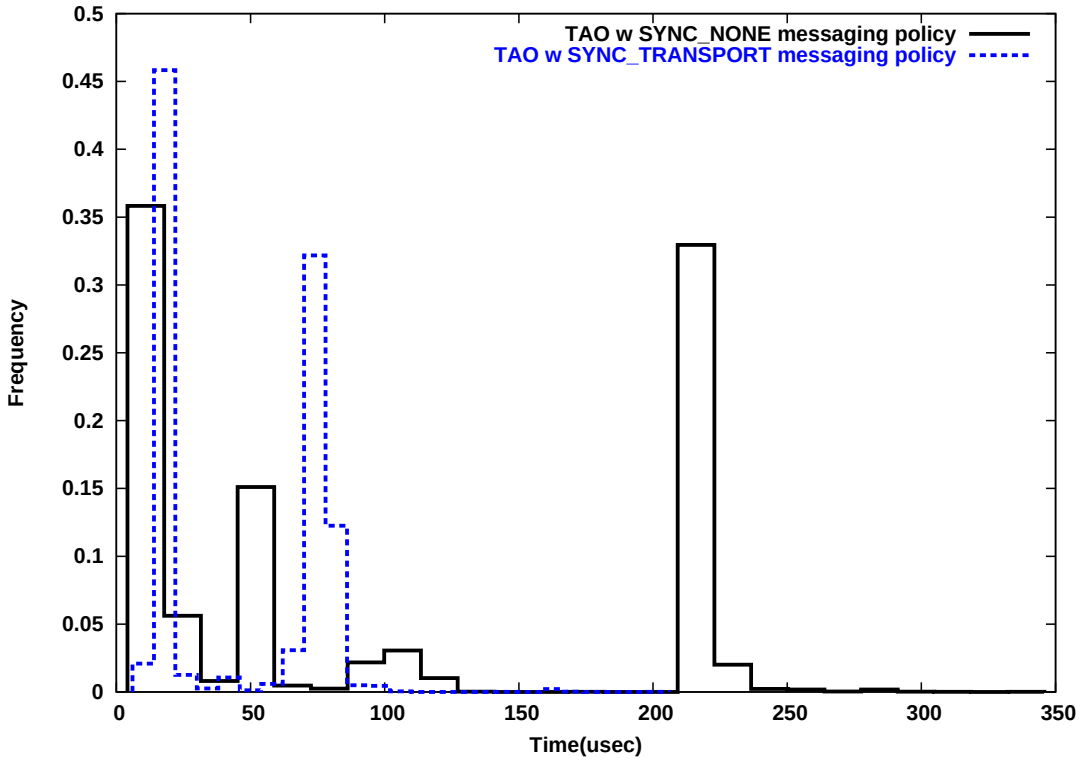


Figure 5.15: TAO Wait Times: 2 Nodes

We found the `SYNC_NONE` policy lead to slower application performance than `SYNC_TRANSPORT` policy in TAO during our performance experiments using 2 nodes. Because we had tagged the individual timing data with ids for the segments being measured, we were able to observe that reasonably frequent cases of ORB-level message buffering were occurring for TAO with 2 nodes. We then examined the code path in TAO and identified the mechanism configured by the `SYNC_NONE` policy as the cause of the observed message delays.

5.4 The Effect of Graph Topology

From the experiments of previous section, we can see the *cycle time* is influenced by the the topology of the graph. In one CSP cycle, each node has to wait *parameter messages* from all its neighbors before it can proceed to next cycle. This synchronization leads to the coupling behavior between a node and its neighbors. A delay encountered in one node could be propagated to other nodes in following cycles. Thus the overall application performance is affected by the graph topology. To investigate the effect of graph topology on overall application performance, we measured the *cycle time* of different implementations for some typical graphs with various number of nodes and topologies.

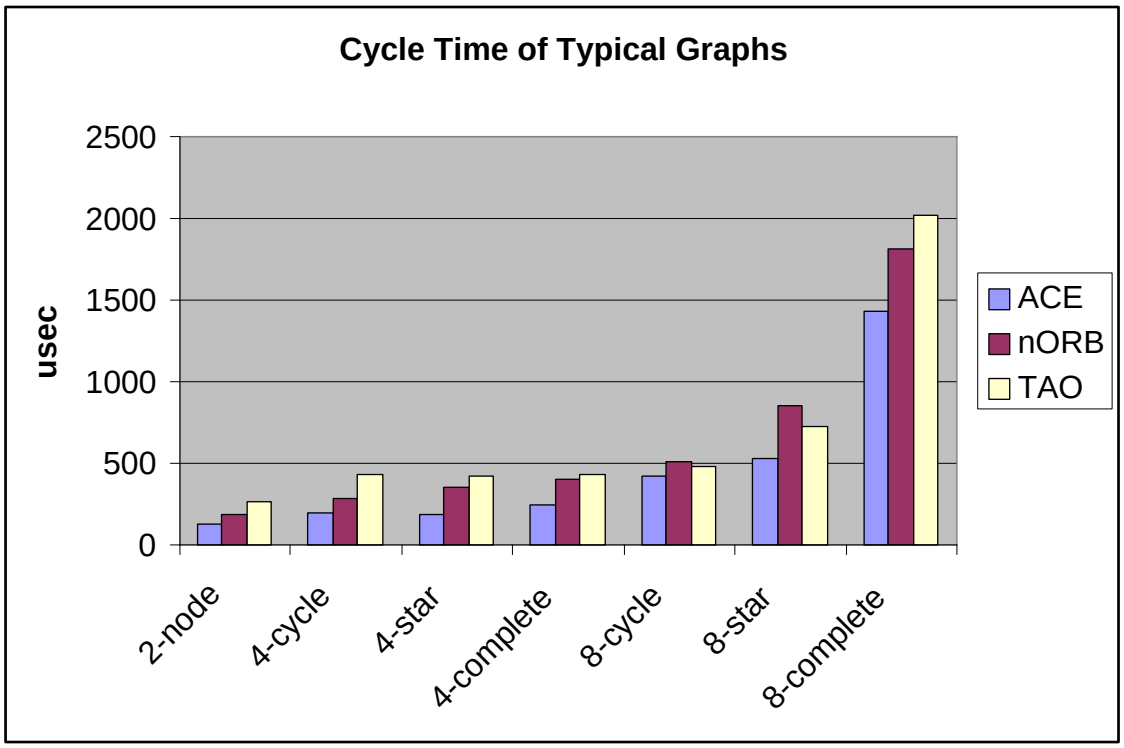


Figure 5.16: Wait Time in One CSP Cycle

Figure 5.16 shows the average *cycle time* for 3 graph topologies: star graph composed of one 'central' node and all other nodes connected to it, cycle graph in which all nodes are chained together, complete graph in which there is a link between any two nodes. The number of nodes varies from 2 to 8 for each topology. This

configuration allows us to measure the application performance under different node density and graph topology. For each graph, The average *cycle time* is measured for 3 ColorCSP implementations including ACE, nORB with runtime critical path optimization and TAO. Each data is the average of 30 rounds. From the figure, we can see for each topology ACE performs best and nORB outperforms TAO in most configurations. This conforms to the previous results from 100-node experiments. However, TAO performs similarly as nORB for cycle graph with 8 nodes and outperforms slightly better than nORB for star grah with 8 nodes. This is partly because the marshaling overhead of nORB is higher than TAO and the central node in star graph incurs high delay and slows down the overall application performance.

For each implementation, the pformance decreases with the increase of the number of nodes. For example, ACE implementation performs one CSP cycle in 128 *u*seconds for 2-node graph while 191 *u*seconds and 529 *u*seconds for 4-node star and 8-node star graph respectively. This is because the propagation effect is more significant when the number of nodes increases. In addition, the configuration with cycle graph performs best in 3 topologies and configuration with complete graph performs worst. This matches our expectation since the node density of complete graph is higher than star and cycle graph. Furthermore, the central node in star graph has high load and the performance of this node dominates the overall performance due to the synchronization between this node and the rest nodes. Nodes in cycle graph is more load-banlanced and perform better than node in star graph.

5.5 Discussion

In this section we discuss the results of application-level and fine-grain experiments. The effect of each mechanism-level operation to overall application performance is analyzed.

Application Cycle: The time taken by each cycle affects the total time the algorithm takes to converge. As illustrated in Figure 5.9, the application cycle stage is influenced by each of the middleware-level operations, as we explain in detail below. Because of the topology of the 100-node graph, a delay in any of these middleware-level stages has a potential ripple-effect, which then may increase the delay experienced in the wait stage and as a result the overall cycle time may be impacted significantly.

Marshal: When a remote call is made by application, the application data is marshaled to a common on-the-wire format by middle-level operations. All three implementations use the CDR format to marshal the data. In ACE implementation, since the data formats are decided at design time, no extra information is needed to identify the destination object in the server or the associated method on the destination object. On the contrary, TAO and nORB needs header information to identify the appropriate target object and its method. When the payload is relatively small, the overhead introduced by the header could impairs the overall performance.

Furthermore, marshaling may cause more significant amount of overhead if performed repeatedly. In the *ColorCSP* application, marshaling may be performed multiple times in one cycle based on the topology of the graph. For example, for the fully connected 4-node graph, this would be done 6 times per cycle at each node: once for each of the 3 neighbors for the color and improvement messages. For TAO and ACE, this amounts to doing the marshaling all the 6 times, whereas for ACE, the marshaling can be done once and then the marshaled data sent to all the neighbors. This forms one of the most significant sources of overhead for the Color-DBA implementations using TAO and nORB, when compared to the implementation using ACE.

Send: This stage measures the time incurred to send the byte stream assembled from the previous stage. This is the time it takes to hand over a byte stream buffer from the application buffers to the OS kernel buffers. The length of the byte stream buffer determines the amount of time it takes in this stage. TAO and nORB takes more time in this stage compared to ACE because of the header information sent with each request.

Wait: This stage accounts for the idle time in a node, while it is waiting for messages from its neighbors. This time is influenced by the other stages, the distributed nature of the algorithm, and the topology of the graph. The wait time is also sensitive to delays in network. The implementation using ACE shows the least delay in this stage, because of the lower time spent in the other stages.

We observed a potentially interesting property of the wait times in the 2 node TAO configuration using `SYNC_NONE`. Once a delay was set up, it was possible for the two nodes to lock into a fairly stable and synchronous pattern of message exchange, resulting in persistently long wait times. In the larger graphs this effect was not

observed, and we speculate that some elasticity property due to the relative degrees of freedom of the nodes in the 100 node, 4 node, and 2 node networks is involved.

Receive: Upon receipt of a message byte stream from a client, the server tries to read a header, which contains the total length of the payload. The payload in turn consists of the marshaled request header and application data sent from the client. Based on the information in the header, the request is assembled. In this stage, a very small amount of demarshaling is involved - demarshaling the header information. Since all three implementations do this and the observed times are uniformly small, this stage has little effect on the overall cycle time.

Lookup and Dispatch: Once a request is completely assembled by the previous stage, the request is parsed to dispatch the method to the appropriate server-side implementation object. This involves demarshaling the request header, looking up the servant object using the object key embedded in the request header, looking up the method to be called on the servant object and then making the upcall on to the skeleton object, which finally dispatches the data to the implementation.

The ColorCSP ACE implementation knows the target object of the incoming call at design time and hence does not go through the stage of lookup and dispatch, whereas in TAO and nORB this has to be done for each and every remote call on the server. This is a significant source of overhead and is one of the reasons for the lower cycle times observed for the implementation using ACE. For one remote call, this might not pose a significant overhead. But, as shown in Figure 5.9, this happens two times the number of neighbors in each cycle on each node. Furthermore, this effect increases the delay on each node as it waits to get data from its neighbors.

Demarshal: The time spent in this stage is the time taken by the skeleton to demarshal the application data payload from the incoming CDR stream, which contains the marshaled payload sent by the client. This does not include the time taken to demarshal the header and request header, since that time is included in the *Lookup and Dispatch* stage. Since the ACE, TAO and nORB implementations use the ACE CDR stream classes, the time taken to demarshal is the same for all of them, since the application message structure is the same across all the three implementations. We observed that the time taken is very small (sub-μsec) and hence has very little effect on the overall performance.

Algorithm Segment: After the target has been identified, the method encoded in the request is called on that object. This stage is solely determined by the application logic. In the *ColorCSP* application, each node evaluates the current color assignments locally and searches for a better assignment which would reduce the amount of violations in the coloring rules. The steps performed in this stage are both simple and exactly the same for the three implementations, and hence have relatively little effect on the overall performance.

Chapter 6

Conclusion and Future Work

In this thesis, we have developed a performance-driven middleware framework for supporting *DCSP* applications. The prototype system ColorCSP which is built on this framework has been shown to be highly portable across various middleware platforms. We have presented a methodology for combined application and middleware mechanism-level performance analysis. Using the prototype system, we performed extensive experiments to evaluate the performance of special-purpose middleware nORB. In investigating the problems revealed by empirical results, we explored various optimization techniques to nORB. The resulting performance of nORB is improved significantly and comparable with the high-performance CORBA implementation TAO.

The rest of this chapter is structured as follows. Section 6.1 presents key observations obtained from our eimperical results, and we discuss the possible solutions to the observed problems. Section 6.2 then describes future work.

6.1 Lessons Learned from Empirical Results

- *Redundant Marshaling:*In *DCSP* applications, a node sends the same *parameter messages* to its neighbors. Using a standard ORB data delivery mechanism causes the same data to be marshaled repeatedly, which incurs extra overhead to application. The ColorCSP implementation using ACE uses application-level knowledge and optimizes this by sending the marshaled *parameter message* to all neighbors. There are several different candidate mechanisms to avoid redundant marshaling when the same data is sent to multiple neighbors, including memoization, multicast, and Group IORs etc.

- High Time Complexity for Lookup: TAO performs better in lookup operation than nORB and provides bounded behavior using strategies like perfect hashing for operation lookup. Currently, nORB uses a linear search for operation lookup, which is appropriate only for the server with a few registered objects.

- Large Header Size:The size of the request header plays a significant role in the time of marshal and demarshal, when the payload is relatively small. Improved efficiency could be achieved by payload aggregation or header minimization. Fine tuning the contents of the header based on the application requirements may be necessary in some embedded systems.

6.2 Future Work

The future work include:

1. Integrating more efficient *DCSP* algorithms. One of the most effective algorithms we plan to integrate into our existing framework is Distributed Stochastic Algorithm*DSA* [27] which outperforms the *DBA* in many cases.

2. Extending our framework to support more general distributed message algorithms.

3. Exploring further optimization techniques to special-purpose middleware. In particular, we plan to apply our key observations obtained from empirical performance results to nORB.

References

[1] Ramon Bejar, Bhaskar Krishnamachari, Carla Gomez, and Bart Sellman. Distributed constraint satisfaction in a wireless sensor tracking system. In *IJCAI-01 DCR Workshop*, Seattle, WA, 2001.

[2] Center for Distributed Object Computing. The ACE ORB (TAO). www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[3] Center for Distributed Object Computing. The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html, Washington University.

[4] S.E. Conry, K. Kuwabara, V.R. Lesser, and R.A. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Trans. Systems, Man, and Cybernetics*, pages 1,462–1,477, 1991.

[5] DARPA IXO. Networked Embedded Software Technology (NEST). `http://www.darpa.mil/ixo/`.

[6] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[7] Douglas C. Schmidt et. al. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[8] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 20(2), March 2001.

[9] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[10] Timothy H. Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. *submitted to the Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 1998.

[11] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.

[12] Steven Minton, Mark D. Jonston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[13] Pragnesh Jay Modi, Hyuckchul Jung, Milind Tambe, Wei-Min Shen, and Shriniwas Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. In *Proceedings of Seventh International Conference on Principles and Practice of Constraint Programming*, Paphos, Cyprus, 2001.

[14] Carlos O'Ryan and Douglas C. Schmidt. Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation. In 5^{th} *International Workshop on Object-oriented Real-Time Dependable Systems*, Monterey, CA, November 1999. IEEE.

[15] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Applying Optimization Patterns to the Design of Real-time ORBs. In *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, San Diego, CA, May 1999. USENIX.

[16] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[17] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks.* Addison-Wesley, Reading, Massachusetts, 2002.

[18] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.

[19] W. Richard Stevens. *UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2nd Edition.* Prentice Hall, Englewood Cliffs, NJ, 1998.

[20] Venkita Subramonian and Christopher Gill. OMG Workshop On Embedded & Real-Time Distributed Object Systems. In *Experiences with Middleware for a Networked Embedded Software Technology Open Experimental Platform.* Object Management Group, January 2002.

[21] Venkita Subramonian, Guoliang Xing, Christopher Gill, and Ron Cytron. The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study. Technical report, Washington University in St.Louis, St.Louis,MO, 2003.

[22] Nanbor Wang, Douglas C. Schmidt, and Steve Vinoski. Collocation Optimizations for CORBA. *C++ Report*, 11(10):47–52, November/December 1999.

[23] L. Wittenburg Weixiong Zhang. Distributed breakout revisited. In *Proc. 18-th National Conf. on Artificial Intelligence (AAAI-2002)*, pages 352–357, Edmonton, Canada, July 2002.

[24] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[25] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998.

[26] Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problem. In *International Conference on Multiagent Systems*, 1996.

[27] Weixiong Zhang, Guandong Wang, and Lars Wittenburg. Distributed stochastic search for constraint satisfaction and optimization: Parallelism, phase transitions and performance. In *Proceedings of AAAI Workshop on Probabilistic Approaches in Search*, 2002. to appear.

Vita

Guoliang Xing

Date of Birth April 20, 1977

Place of Birth Zi Bo, Shan Dong, P.R.China

Degrees M.E. Computer Sci. and Engr., 2001,
from Xi'an JiaoTong University, Xi'an, China.

May 2003