# A Framework for the Development of Multi-Agent Architectures

Moises Lejter  Thomas Dean[1]
Department of Computer Science, Brown University
115 Waterman Street, Providence, RI 02912, USA
Phone: (401) 863-7600  Fax: (401) 863-7657
Email: {mlm,tld}@cs.brown.edu

**Abstract**

This paper describes a high-performance software system that supports distributed computing and multi-agent coordination. Our system provides the facilities necessary for experimenting with a variety of choices at different levels of abstraction. At each level, we provide a set of abstractions, tools to manipulate those abstractions, and the design rationale behind those abstractions. This paper presents the abstractions that make up our system, along with an empirical evaluation of their usefulness.

At the base level, we define a set of abstractions for communications among a collection of agents. At this level, the key advantages of our system are twofold: design flexibility and scalability. We also provide additional abstractions that provide functionality equivalent to that found in other well-known communications formalisms. At higher levels, we define a set of abstractions that describe the most widely used distributed control protocols. As a result, we also gain portability.

We also provide an empirical evaluation of the utility of our framework for the construction of distributed systems. The low-level abstractions are evaluated in the context of their efficiency and scalability for likely distributed architectures; the high-level abstractions are evaluated in terms of their expressive power.

[1] This work was supported in part by the Air Force and the Advanced Research Projects Agency of the Department of Defense under grant No. F30602-95-1-0020, by the National Science foundation in conjunction with the Advanced Research Projects Agency of the Department of Defense under grant No. IRI-9312395.

# 1 Introduction

This paper describes a high-performance software system that supports distributed computing and multi-agent coordination. We built the system "from the bottom up", providing the facilities necessary for experimenting conveniently with a variety of choices at different levels of abstraction. At each level, we provide a set of abstractions, tools to manipulate those abstractions, and the design rationale behind each of those abstractions.

At the base level, we define a set of abstractions for communications among a collection of agents. Our abstractions are based on the message-passing paradigm of communications. At this level, the key advantages of our system are twofold: *design flexibility* (both architectural and execution-time), and *scalability*. Section 2.1 will discuss these abstractions in more detail. At this level, we also define a set of abstractions that provide functionality equivalent to that found in other well-known communications formalisms, such as remote procedure calls [Birrell and Nelson, 1984], and tuple-space, or *generative*, communications [Gelernter, 1985].

The next layer of abstractions builds on this communications infrastructure. It provides abstractions that ease the task of constructing distributed systems based on common control protocols, such as client-server, subsumption layers, or pipelines of agents. These protocols can be characterized by their "static" nature: system designers can exactly determine at design time the responsibilities and interactions among all agents in the system. Section 2.2 will discuss these abstractions in more detail.

At the highest level of abstraction, we define a set of abstractions that describe a number of different coordination protocols. By *coordination protocols* we mean the protocols used by the agents in a distributed multi-agent system to agree on the behaviors the system will actually exhibit. These protocols can be characterized by their more "dynamic" nature: the details of the responsibilities and interactions among agents may not be known until run-time. A simple example of a coordination protocol could be the "contract net" [Smith, 1980], where the actual responsibilities of the individual agents depend on circumstance. Section 2.3 will discuss these abstractions in more detail.[2]

We also provide an empirical evaluation of the utility of our framework for constructing distributed systems. Our framework includes abstractions

[2] The distinction between the middle- and high-level layers is really a matter of degrees, rather than a clear static/dynamic. We will discuss this in more detail in Section 2.

at multiple levels of complexity, and the evaluation reflects it. The higher-level abstractions are evaluated in terms of their expressive power, by describing a variety of systems and how they could be built using our abstractions. The lower-level abstractions are evaluated in terms of their efficiency, and their performance is compared against that of other comparable systems.

# 2 Protocols for Multi-Agent Architectures

The construction of a multi-agent system requires that developers choose an architecture for the system. This includes a specification of how the responsibilities of the system are to be distributed among its component agents, along with a specification of how they will interact with each other to achieve those responsibilities. At a minimum, some simple communication scheme will be necessary to coordinate the activities of the different agents or for the agents to exchange information. As the architecture becomes more complex, it may require more flexible communications support, or more sophisticated algorithms to coordinate the activities of all the agents in the system.

Our framework assumes that these two issues must always be considered when designing a multi-agent architecture: the capabilities of the communications infrastructure that the system will use, and the strategies to be used to coordinate the activities of the agents within the system. As a result, our framework includes two fundamental components. The first is a set of low-level abstractions, providing communications in a way that is both flexible and efficient; it will be discussed in Section 2.1. The second is a set of high-level abstractions that represent the underlying control structure common to all distributed systems. This second set of abstraction we will present as two different layers, to emphasize the difference between static and dynamic coordination strategies: Section 2.2 will introduce the abstractions corresponding to static strategies, while Section 2.3 will concentrate on those corresponding to the dynamic ones.

## 2.1 Communications Protocols

The architecture for a distributed agent system assumes an underlying communications protocol that determines the possible relationships among all agents. The literature includes a variety of communications protocols that

fit this role: message-passing, remote procedure call (RPC), and tuple-space communications are the most popular protocols.

Message-passing protocols assume that communications among the individual agents take the form of *messages* routed from a particular agent to another agent (or set of agents). This exchange of messages is the only means by which information is passed among the agents in the system. Remote-Procedure-Call protocols assume that agents in a distributed system exchange information by means of two-way exchanges of information, where the sender agent invokes a function associated with the destination agent (who can then receive information from the sender in the form of function arguments). The return value produced by that function would then be information "passed back" from the destination agent to the original sender. Tuple-space communications assume a central data store available to all agents. Agents can communicate with each other in this model by placing and retrieving items (called *tuples*) from this central store.[3]

Our system introduces a set of abstractions (collectively known as the CoRaCLe library) based on the standard message-passing communications protocol. The individual agents of a system built using the facilities of CoRaCLe can exchange messages with others, using either point-to-point (directed) addressing, or multicasting when addressing a set of agents at once. In addition to this, individual agents are also able to "eavesdrop" on message traffic not directly addressed to them, based on criteria specifiable by each agent in the system.

CoRaCLe shares with other systems the specification of its message exchange facilities. CoRaCLe differs from these other systems in that the abstractions it specifies describe more than just the actual message exchange facilities. CoRaCLe also describes as part of its abstractions those that provide the overall support structure that is internal to the message transport layer. Designers using CoRaCLe then also have control over the particulars of the message transport to be used.

Message-based protocols specify what capabilities the individual agents in a multi-agent system will use to exchange messages. These protocols often specify such capabilities in such a way that the internal structure of the message-passing subsystem is hidden from the rest of the application. Any design decisions related to that internal structure are taken by those who provide the protocol; they are fixed, or at best rigidly constrained, from the

[3] This approach was introduced as a programming-language paradigm by Linda [Gelernter, 1985]; it was also introduced as an architecture for AI systems by [Hayes-Roth, 1985].

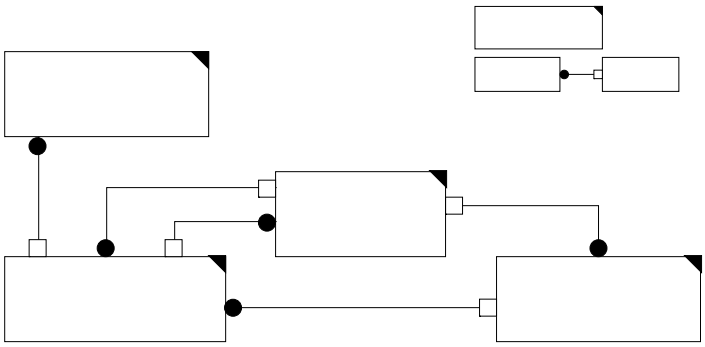


Figure 1: The CoRaCLe framework

perspective of the multi-agent systems using them.

CoRaCLe was designed with this kind of design flexibility in mind. CoRaCLe describes generic abstractions that any message-passing protocol can be implemented on; the relationships between these abstractions are illustrated in Figure 1. It also specifies how message-passing protocols are implemented in terms of those abstractions. However, it leaves the actual decisions as to what the proper implementation for each of those generic abstractions will be. This gives designers the freedom to choose the implementations that will be best-suited for particular applications. To illustrate the power behind this approach, we distribute along with CoRaCLe three different realizations of the abstractions it defines. The three illustrate different ways in which the abstractions described by CoRaCLe can result in actual message-based protocols with specific features. They also illustrate how applications can take advantage of the isolation CoRaCLe provides between the semantics of the message passing protocol to use and its implementation to gain scalability. All the applications must do is specify that a different implementation of some or all the CoRaCLe abstractions are to be used.

CoRaCLe is strictly a message-based communications protocol. In order to ease the transition to the facilities our system provides, we also provide a set of abstractions built on top of CoRaCLe that emulate other traditional communications protocols (in particular, remote procedure call and tuple-space protocols are so provided).

## 2.2 Distribution (Control) Protocols

Once a decision is made to construct a system as a collection of cooperating agents, a decision has to be made as to what the control structure of the system as a whole will be.

Most distributed systems evolve a control structure during their design that is appropriate for the tasks that system is assigned. A few different strategies can be identified, across the distributed systems described in the literature (see [Andrews, 1991] for a survey of distributed processing paradigms):

- **pipeline** In pipeline strategies, the individual agents in the system are seen as producer/consumers of information. They each receive information from an "earlier" agent in the system, process that information, and send it on to a "later" agent in the system.

- **request-response** In request-response strategies, the individual agents in the system are organized in a hierarchy. In this hierarchy, agents "higher up" the hierarchy are given complex tasks. They are individually responsible for breaking those tasks down into subtasks, and distributing them over the set of agents in the next level in the hierarchy directly under their control. The *client-server* and *master-slave* distributed processing paradigms are examples of what we call request-response strategies.[4]

- **subsumption** In subsumption strategies, the individual agents that make up the system are connected in a fixed network. Instead of having individual agents partition responsibility among others, subsumption agents are each built to assume they can take overall control of the system, by suppressing communications for others with lower priority than themselves.

- **peer-to-peer** Peer to peer strategies are *dynamic*, unlike the strategies described above. In peer-to-peer strategies the determination of which of the different agents in the system is to actually control the overall behavior of the system is reached through some kind of agreement among all the peer agents that participate in that system. That decision is delayed until the system is actually running.

[4] *master-slave* and *client-server* differ on details of the semantics associated with the individual agents in the system.

We consider peer-to-peer strategies to be qualitatively different from the other strategies described (pipeline, request-response, and subsumption). The element that distinguishes peer-to-peer strategies is the need for mechanisms to achieve some consensus among the participating agents while the system executes. Furthermore, a variety of mechanisms to achieve that consensus are possible. The organization of our framework will reflect this difference: the first three strategies correspond to the "middle layer" of our framework, while the last strategy is given a layer of its own.

The AI literature includes examples of systems that span the spectrum of strategies just described. There are systems which have a fixed (hierarchical) organization of agents, along with systems where the hierarchical organization interacts with the runtime behavior of the agents (Brooks's subsumption architecture [Brooks, 1985]). There are some where the set of agents and the determination of control rely on agreement among the agents involved (voting schemes). Figure 2 illustrates the architecture of Huey, one of our early attempts [Dean et al, 1990] at designing a robot architecture along hierarchical lines. Huey was controlled by a collection of modules organized in a hierarchy in terms of their responsibilities: modules higher up the hierarchy could control the modules below them; modules at the same level were independent of each other. In the case of Huey, this defined a hierarchy of 5 levels (where we chose to number them with 0 being the lowest); the paths of communication and control were those illustrated by the arrows in Figure 2: control always "top-down", communications as indicated by the arrows.

Instead of dictating that a particular strategy is best suited for distributed applications, our system provides a set of abstractions (collectively known as the DisPeL library) that realize the set of strategies mentioned above. Designers building distributed applications can simple specialize from the abstractions provided by DisPeL, without having to concern themselves with the details of implementing the specific strategies they chose for their applications. DisPeL, discussed in this section, includes support for the fixed distribution strategies described above. CAF, discussed in Section 2.3 will discuss our support for peer-to-peer strategies.

DisPeL provides a set of core abstractions that describe the different kinds of agents that can participate in distributed systems that use some variant of a fixed strategy, including the *pipeline*, *request-response*, and *subsumption* strategies describe above. The set of abstractions presented by DisPeL is an extension of the basic framework proposed in [Andrews, 1991]. Figure 3 illustrates the icons we use when drawing the architecture of sys-

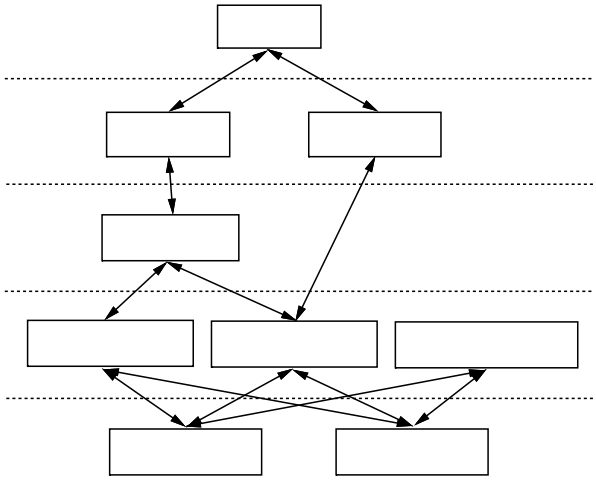


Figure 2: Architecture of Huey

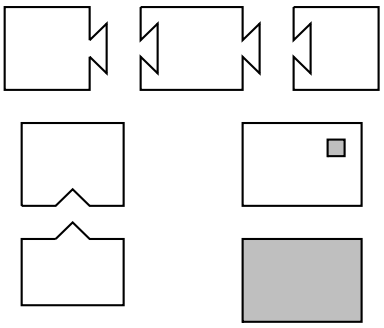


Figure 3: DisPeL icons

tems built using those DisPeL abstractions.

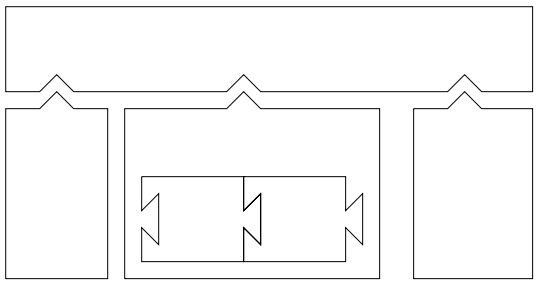
DisPeL support for pipeline strategies is based on the `PipelineAgent` abstraction, along with the three special cases of `Source`, `Filter`, and `Sink`. A `Filter` is an agent that receives messages from its supplier (the filter before it in the pipeline), processes them, and produces messages it delivers to its consumer (the filter after it in the pipeline). `Source` and `Sink` are simply special cases for the endpoints to the pipeline; any number of instances of `Filter` could be placed sequentially between an instance of a `source` and one of a `Destination`. Developers who wish to write a distributed system based on a pipeline strategy can simply derive their own

system-specific abstractions from the three abstractions DisPeL provides. The DisPeL abstractions provide the basic machinery necessary to connect the individual agents into a pipeline. They also specify the semantics of the behavior of the individual agents developers will write; in deriving their agents from the DisPeL abstractions, the developers customize those behaviors to match the goals required by the system at hand.

DisPeL support for request-response strategies differs between the two cases of *client-server* and *master-slave* strategies. The differences between these two cases have to do with the specific semantics DisPeL associates with the abstractions involved. A generic abstraction `ReqRespAgent` is provided, to describe at a high-level all agents participating in a system organized using such strategies.

To support *client-server* strategies, DisPeL includes the two abstractions `Client` and `Server`. As in [Andrews, 1991], a *client* is a triggering agent, making requests of servers; a *server* is a reactive agent, responding to those requests it receives. Clients initiate activities; they do not necessarily have to wait for their requests to be fulfilled before continuing on their own execution. Servers wait for client requests; servers can handle requests from any number of clients, and need not be dependent on any specific client. To support *master-slave* strategies, DisPeL includes the two abstractions `Master` and `Slave`. Like above, a *master* is a triggering agent, while a *slave* is a reactive agent. However, the association between a master and its slaves is somewhat different from that between a client and its servers. Slaves serve a single master, and are dependent on that master they serve. Again, developers need just derive from these abstractions new ones to represent the agents they require for the systems the developers wish to construct. The DisPeL abstraction provide a framework that provides all the basic functionality and specifies what services developers need customize for their own purposes. When designing a distributed system as a multi-layer hierarchy, it is easy to imagine agents within that hierarchy playing multiple roles at the same time. The DisPeL abstractions that support *request-response* strategies can be freely combined to provide the basis for any such agent that developers may require.

The strategies described above can be used to implement the different DAI architectures in the literature. Some, like Brook's subsumption architecture [Brooks, 1985], motivate the strategies proposed. Others are examples of those strategies, like Ferguson's TouringMachines [Ferguson, 1992] (master-slave strategy), or Brattman, Israel, and Pollack's IRMA [Brattman, Israel and Pollack, 1988] (master-slave and pipeline strategies). In partic-



The TouringMachines system has an overall controller (the ControlFramework) running the system. This controller decides which of three independent modules (Reactive, Planning, or Modeling) is to run. The Planning module is implemented as a two-step process: a Focus task decides what needs to be done, then a Planner task figures out how to achieve what Focus decided on achieving.

Figure 4: The TouringMachines system

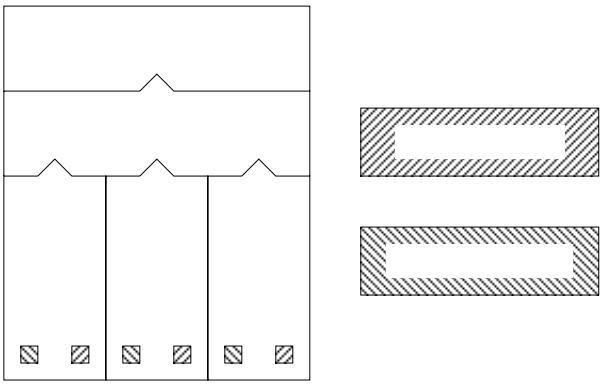
ular, Figure 4 shows how we could build the architecture of the TouringMachines system; Figure 5 shows how we could build the architecture of Huey (our early robot) using the abstractions presented here.[5]

### 2.3 Coordination Protocols

A particular kind of a Distribution Protocol common in AI is the one based on voluntary coordination among multiple agents. CAF is our cooperating agent framework. CAF includes standard AI coordination protocols, including voting, contract, and negotiation protocols based on game-theoretic ideas. As in the case of DisPeL, each of these protocols motivates its own small set of abstractions.

Voting protocols have been used in several distributed networking applications. They have been the target of some interest on the part of researchers interested in applying constraints derived from social behavior to the domain of intelligent distributed agents. They are all based on the notion that a collection of interacting agents can achieve useful common behaviors by simply voting on their favorite over a collection of compet-

[5] The architecture was somewhat simplified for illustration purposes.



The Geographer module has overall control of Huey. Geographer relies on the services of the LDPclassifier module to navigate, and to identify the robot's current location so as to add it to the map the Geographer is constructing. The LDPclassifier module, in turn, relies on three independent modules FeatureRecognizer, CorridorFollower, and ObstacleAvoidance to navigate through the world and recognize what kind of locations the robot traverses. All three modules FeatureRecognizer, CorridorFollower, and ObstacleAvoidance share access to the Sonar and Motor controllers that are in charge of the robot's sensors and motion.

Figure 5: The Huey architecture, revised

ing proposals, then choosing the most popular alternative. For example, Urken [Urken, 1990] describes the application of a voting scheme to distributed network routing on a phone network. A similar class of protocols discussed in the literature are the *market-oriented* protocols [Wellman, 1993], where the behavior of the agents of the system is controlled by the principles of market economy: the designers of the system define an artificial economy. The agents participate in this economy as either producers or consumers; Their behavior is determined by their decisions in this artificial market economy.

The support CAF provides for voting protocols revolves around the `Peer` and `Council` abstractions. Each agent that uses a voting protocol is an instance of a `Peer`. All peers involved in a particular control decision belong to the same instance of a `Council`, which is responsible for the mechanics of voting. Decision-making is a two-step process: whenever a decision is to be made, each peer is expected to provide some of the alternatives to be voted on. Once all peers have submitted their alternatives, the complete list of alternatives is given to all peers, who must then vote for their preferred alternative. Several kinds of `Council` are available, with different strategies to use in case of ties.

Another coordination strategy is based on the notion of bidding for a contract. In this strategy, each of the agents is able to receive a specification for a task to perform and partition it into subtasks. Agents have no information about which other agents should be assigned the resulting subtasks. Instead, the protocol calls for the agents that partition a task to open each of the resulting subtasks to bidding. Agents able to fulfill those tasks will bid for them, based on their own expectations of how well that agent can fulfill the tasks. The highest bidder always "wins" each subtask and becomes responsible for achieving it. Smith's *Contract Nets* [Smith, 1980] originally introduced the bidding protocol strategy. A recent example of a system built using contract nets is Sen's distributed meeting scheduler [Sen, 1996].

The support CAF provides for contract net protocols revolves around the abstractions `Bidder`, `Client`, and `Contract`. Each agent that wishes to submit a task to bidding must be an instance of a `Client`. Clients are responsible for putting out requests, accepting bids, and selecting a winning bidder, if there are any, or rejecting all bids and either terminating the bidding or requesting new bids. Agents who wish to participate in bids must be instances of a `Bidder`. They are responsible for accepting requests, submitting bids when appropriate, and then fulfilling those bids they are granted. Bidders should register with the clients for whom they

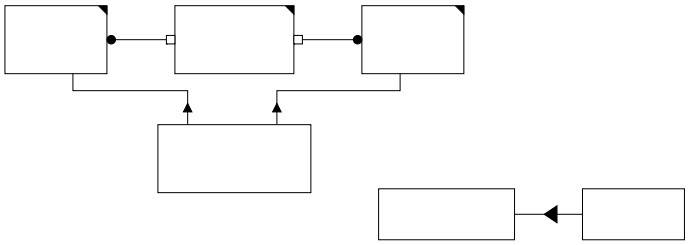


Figure 6: The Distributed Meeting Scheduler agent class

are willing to bid, but the framework provides a mechanism whereby bidders can be notified of contracts they may be interested in bidding for. To implement Sen's distributed meeting scheduler using CAF, we could create a DMS agent class as a subclass of both `Client` and `Bidder`, as shown in Figure 6.

### 2.3.1 A game-theoretic protocol

CAF also provides support for a new kind of negotiation protocol inspired by game-theoretic ideas. Essentially, agents interested in coordinating their behaviors construct a game-tree in which they each describe their choices on every step of the evolving plan represented by the game tree, until a mutually satisfiable solution is found. The approach taken here in broad terms is similar to that described in [Lesser, 1991] – we build a graph of the options available to the set of agents to coordinate, then choose the most appropriate action. The details on how the graph is constructed and then later evaluated differ from those Lesser describes. Lesser's original formalism did not consider dependencies across different tasks that different agents placed on the graph. The approach taken in CAF addresses this issue by incrementally updating the game tree and letting all agents criticize it as it expands. An updated presentation of Lesser's formalism, along with an alternative approach to address this issue, is presented by [Jennings, 1996].

The overall structure of a single agent in this protocol is illustrated by Figure 7: the *arbiter* part of the agent is the one responsible for receiving messages from other agents that represent how they would grow the game tree, incorporating them into this agent's game tree, and evaluating the result; the *planner* part of the agent is the one responsible for generating the requests to grow other agents' game trees that this agent, in turn, would forward to other agents in the system.

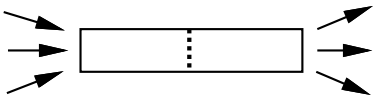


Figure 7: The structure of the game-theoretic agent.

Figure 8: An initial position in the game.

We designed a simple benchmark, both to illustrate the coordination procedures available to CAF and to test the performance of this protocol. Our benchmark based on the Towers of Hanoi puzzle. In it, there are three holders on a table, each capable of holding a stack of pieces. On a second table, off to a side, there is a single holder. One of the holders on the first table contains a collection of red and green pieces, ordered by size with the smallest piece on top. The goal of the game is to transfer all the red pieces from the initial holder to another holder on the same table, and to transfer all the green pieces to the holder on the second table. Figures 8 and 9 illustrate the initial and final positions of such a game.

All the usual rules of Towers of Hanoi apply to this game. In particular:

- a single piece can be moved at a time.
- a piece can only be placed at one of the pegs available, on top of a larger piece already at that peg (unless the peg was initially empty).

The two tasks of moving the red pieces and moving the green pieces can be given varying priorities. It is expected that when the priority of

Figure 9: A final position in the game.

moving the red pieces is larger than that of moving the green pieces, the robot will preferentially deal with red pieces, moving only green pieces when absolutely necessary. The reverse should be true if the two priorities are reversed. If both tasks have low priority, then the robot should try to minimize total travel time of the arm. If both tasks have high priority, the robot may fail at one of the tasks, in order to finish the second one. It ought not to fail both tasks, if it is at all possible to solve at least one of them. The relative priorities to be assigned to both tasks will be specified as a maximum time available for completion of that task.

The simulated robot consists of an arm that can move horizontally across the different pegs only when fully retracted (and perhaps holding a piece). When horizontally static, the arm can move up and down, so as to pick up a piece from the peg or release a piece at a peg. The arm moves vertically up or down from its fully retracted position to the height of the piece to be picked up or dropped in one control cycle. The horizontal motion between adjacent pegs in the same table takes one control cycle, as well, while the motion between the tables takes a few control cycles. This setup allows the robot to naturally reconsider what action to perform on every cycle, since the arm will be in position to move in any desired direction at that point.

The control system will be made up of three processing agents: one in charge of moving pieces between holders (call it mover), and one for each of the two subproblems of moving the red pieces to their holder (call this agent red) and of moving the green pieces to their holder (call this one green). Both red and green will make requests of mover to achieve their purposes, so it will be necessary for mover to coordinate the requests made by the two other agents; this coordination will be the task of the arbiter module of mover. This is the kind of coordination CAF is designed to achieve - the coordination needed when multiple agents have competing plans for a common resource, and where the goal is to agree upon an executable plan mutually agreeable to all participants.

The arbiter module of a CAF agent can take the following actions:

- accept requests from its clients to execute a plan;
- merge together requests from its clients to construct a composite plan that incorporates all requests received;
- submit a composite plan to the evaluation of the clients involved;
- accept an evaluated plan from each of its clients, and determine whether

the evaluations of all clients can be used to construct a mutually agreeable plan;

- execute a plan, by passing it along to the planner module of that same agent.

The plan constructed by the arbiter is a game tree that contains as its "moves" the actions desired by the individual clients, and where the value for a move sequence for each client is its evaluation of the plan represented by that sequence.

The planner module of a CAF agent is responsible then for determining how to achieve the plan accepted, and negotiating with the arbiter modules of other agents it is a client of.

# 3 Performance Considerations

Two important considerations went into the design of our framework: first, it had to offer *flexibility* to the designers of distributed applications; second, it had to offer reasonable run-time performance. As far as performance is concerned, there are two aspects to consider, when evaluating frameworks such as ours: the performance overhead of using the framework proposed, against that of a dedicated implementation; and the degree of scalability offered by the framework. We begin by studying the scalability of our framework in Section 3.1, followed by discussions of its performance overhead in Section 3.2, and a comparison with other frameworks well-known in the literature in Section 3.3. [6]

## 3.1 Scalability

Discussions on the performance of communications protocols often rely to a large extent on comparisons of a single number: the time taken to transmit a single message from its sender to its destination. This view is too simplistic. There are in fact another three important factors to consider when evaluating communications protocols: the number of agents

[6] The results provided here are somewhat biased, since the set of test runs presented could not factor out the load on the network and computers used for the experiments that were independent from the experiments; the load was fairly light (if at all present), as these test runs were run on our classroom lab, once classes were over. For the purposes of the presentation, we assume throughout that the bias applies uniformly to all experiments.

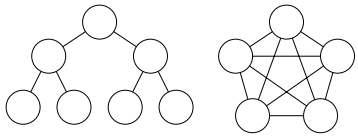


Figure 10: Simple Scalability Tests

in the distributed system, their arrangement within the network, and volume of messages transmitted among the agents in that system. All three of these factors affect the performance and scalability of any communications framework. To understand the influence of these factors on the performance of a communications framework, we consider two broad categories of frameworks: centralized, and point-to-point based.

### 3.1.1 Centralized frameworks

Centralized communications frameworks are built around a central *message clearinghouse*, a central message repository. All agents who use that framework to communicate with each other in fact communicate only with that central clearinghouse; the clearinghouse then delivers all messages to their appropriate destinations. The FIELD environment [Reiss, 1990], Sun Microsystems' ToolTalk [Julienne and Holtz, 1994], and the Central realization of CoRaCLe are all examples of this kind of a framework. We can make certain predictions about the performance of a distributed system based on a centralized message-passing architecture:

- All agents must communicate with the clearinghouse on each request to send or receive each message, regardless of the actual arrangement of agents within the system. As a result, the arrangement of agents within the system for a given number of agents, may not be all that important a factor.
- The number of messages transmitted within the system per unit time affects its performance. All messages must be funneled through the centralized message server on their way from the sender to the receiver of each message. As a result, the performance of the system will depend on how quickly that central server can dispatch each message. If the volume of messages per unit time is larger than can be accommodated by the response time of server, messages will begin to pile up on the server's incoming queues. The result, from the

perspective of the outside observer (or the individual agents) will be a slowdown in the speed with which messages arrive at their destinations.

- Message travel times are increased, since all messages must travel from the sender to the clearinghouse, and then on to their destination.

### 3.1.2 Point-to-Point frameworks

Point-to-point, or distributed, communications frameworks are built around the notion that each pair of agents that wish to communicate with each other within the system share a dedicated communications link. This approach avoids the pitfalls of the centralized approach, since each communications link is independent of all other such links. As a result, large message volumes through particular links do not affect other, unrelated links: the overall volume of messages in the system may not affect any of the individual links significantly.[7] The tradeoff involves additional complexity at each of the individual agents: the communications code at each agent may need to be more complex, and therefore slower, than that of an agent in a centralized system. In these systems, the arrangement of the agents in the system is important. Each additional link placed on an agent increases its complexity, perhaps slowing it down. On the other hand, the overall number of agents in the system should not affect the individual agents.

### 3.1.3 Experimental results

We provide several realizations of the CoRaCLe framework, along with the framework itself. Two of them we call the Central and Distributed realizations. The Central realization of CoRaCLe is a communications subsystem built using the CoRaCLe abstractions that implements a centralized communications subsystem. The Distributed realization of CoRaCLe implements a point-to-point based subsystem. To test the arguments made above about the tradeoffs between the different approaches to implementing a communications subsystem, we ran a series of experiments that test the performance of both realizations under a variety of conditions related to the factors mentioned above: number of agents in system, arrangement of those agents, and message volume. In all experiments, we measured

[7] We ignore for the moment the fact that the underlying communications hardware may also act as a "bottleneck" in the sense discussed above.

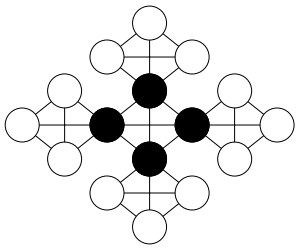


Figure 11: Another Scalability Test

the average time to deliver individual messages (including both processing and transmission delays), and the total time required to run the experiment to completion.

First of all, it was necessary to test a variety of arrangements for the agents in the network. A common practice is to choose a random arrangement of agents who will exchange messages, then measure the performance of that system. Such a technique has the drawback that it need not mirror "real-life" experience – actual agent architectures are not random; the particulars of their organization might well influence the performance of the system.

For our experiments, we chose a series of fixed architectures that would be likely candidates for real-world distributed systems. Figure 10 illustrates the first two architectures: a fixed complete N-ary tree of agents, which mirrors systems with traditional hierarchical control strategies; and a completely connected network of agents, which mirrors coordination strategies such as voting schemes. Figure 11 illustrates our last test architecture: a collection of fully-connected groups of agents that interact with each other via a chosen representative for every group; the representatives themselves are also connected to each other in a fully-connected graph, forming a "council". (Figure 11 illustrates the particular case where there are 4 groups of 4 agents each; the circles represent individual agents, while the black circles represent the chosen representatives for each group.)

The tree, "council", and fully connected networks of agents represent a progression towards ever more complex arrangements of agents within the system. Given our discussion above, we would have expected that the results of the tests runs on the Distributed realization of CoRaCLe to reflect this progression. The test runs for the Central realization of CoRaCLe should not have been affected by the arrangements. The performance of

the Central realization, on the other hand, should be sensitive to the size of the system, independently of its configuration. These were in fact the results obtained, as we shall see below.

The arrangement of the agents of the system is not the only parameter of interest. It was also necessary to test varying numbers of agents for each of the architectures chosen, as well as varying message volumes for the different numbers of agents for each architecture. Both of these parameters affect the overall message traffic flowing through the system. That traffic can also affect the performance: system bottlenecks are typically manifest only in conditions of high traffic load. To model message volume, our experiments had each agent in the test systems send or receive messages at random, at several fixed intervals. [8]

As we mentioned before, one of the design goals behind CoRaCLe was to achieve design flexibility. As the architecture of a distributed system evolves, the relative weights of the factors discussed here will change. Systems built using CoRaCLe can customize the actual implementation to use for the communications subsystem transparently, to best match the current design requirements for the system.

### 3.1.4 The test agent

The same test agent was used for each of the individual nodes in all test runs for all configurations, to eliminate the test agent itself as a variable in all experiments. This test agent is given a list of the names of the neighboring nodes it is to interact with, plus the number of messages it is to exchange with each of its neighbors. On startup, the test agent registers itself with the CoRaCLe server, provides its list of acquaintances to the server, then enters a loop in which it decides randomly whether to send or receive a message until all messages to be sent or received from each of its neighbors are processed. On every iteration, each agent will choose randomly whether to send or receive a message. Each will also chose randomly one of the neighbors still active (that is, those who have not yet sent or received, as appropriate, the number of messages specified) to communicate with, and then issue the send or receive request. In addition to originating the specified number of messages for each of its neighbors, each agent will also

[8] Given the test agent's algorithm discussed in Section 3.1.4, the intervals between successive messages between any given pair of agents follow a normal distribution. The fixed interval mentioned here is the interval between iterations of the test agent's main loop.

reply to each original message received, as soon as they receive that message.

One of the parameters we are interested in testing is the effect of the load on the individual agents on the behavior of the system as a whole. To model this parameter in our testbed, we can specify what we call the *agent cycle delay*, the number of milliseconds that the test agent will pause on every iteration through its main loop. This number will be fixed for all agents in any given test run. The delay between successive requests between any two agents in the system will obey a normal distribution, given the algorithm described above.

Each of the messages contains the sender's time at which the message was sent. With this information, the receiver of the message can compute message transit time as the difference between the receiver's time at which the message was received and the time encoded in that message. Because the time at all of the agents may not be synchronized, the results presented below will focus on elapsed round-trip transit time - the sum of the transit times for a message and its reply. This sum cancels out the effects of variations in agent clocks.

Each agent produces periodic updates on the average transit time for all messages received from each of its neighbors, plus final average transit times per message received from each neighbor. CoRaCLe allows clients to send messages to other clients who may not be connected to the system yet. This could artificially inflate the elapsed transit times for messages when recipients's entrance to the net is delayed. To avoid skewing the results due to these differences, each agent is structured to send an initial message to each of its neighbors and to wait for its response, before beginning to average message transit times for all messages sent and received from that neighbor.

#### 3.1.5 Fully-connected architecture

The fully-connected architecture is a simple arrangement of agents, shown to the right in Figure 10. In this architecture, each agent establishes a connection with every other agent in the system. Table 1 displays the message round-trip times for a set of test runs in which the agents connect to each other and then begin to send/receive messages. It includes the results from three different sets of runs: when individuals exchange messages as fast as they can, and then when delays of either 50 msec or 100 msec are introduced after every send/receive request.

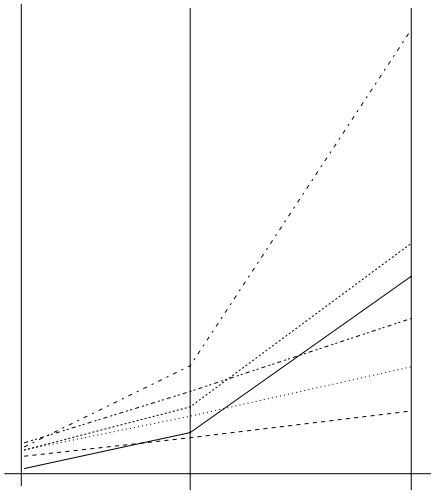


Figure 12: Fully-Connected Results By # Agents

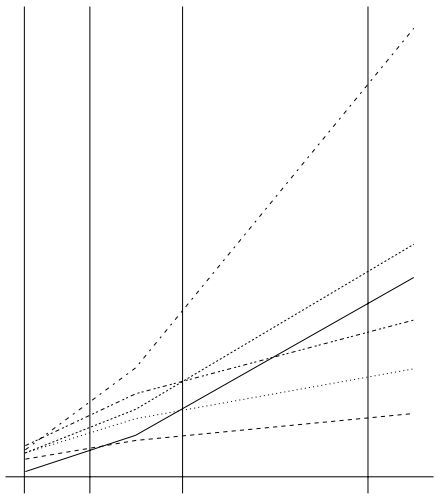


Figure 13: Fully-Connected Results by # Connections

| Delay | # Nodes | # Conn | Message Round Trip (msecs) | |
|---|---|---|---|---|
| | | | **Central** | **Distributed** |
| 0 | 3 | 3 | 287.136 | 976.817 |
| | 6 | 15 | 2330.945 | 2035.839 |
| | 10 | 45 | 11406.251 | 3584.217 |
| 50 msec | 3 | 3 | 1366.809 | 1318.975 |
| | 6 | 15 | 3813.976 | 3285.391 |
| | 10 | 45 | 13289.015 | 6172.110 |
| 100 msec | 3 | 3 | 1535.421 | 1730.029 |
| | 6 | 15 | 6197.208 | 4759.786 |
| | 10 | 45 | 25693.062 | 8985.121 |

Table 1: Results: Fully-Connected Architecture

Table 1 shows how the performance of the centralized system seems to be proportional to the size of the system, either in terms of number of agents, or number of connections established in the system, when under conditions of heavy load; the degraded performance seems to be worse than linear with respect to the number of connections established with the system. Table 1 also shows how the performance of the distributed system seems to be largely independent of the size of the system, measured in terms of number of connections between agents in the system. It would seem to be linear, however, in terms of the overall number of agents in the system. Figure 12 shows a graph of performance as a function of the number of agents in the system; Figure 13 show it as a function of the number of connections.

Our prior discussion suggested that neither measure (overall number of agents, or overall number of connections ) should have affected the performance of the distributed version of the test system. This discrepancy can be explained by examining once again the logic of the test agent used, described above. Notice that on each iteration the test agent must choose whether to send or receive, then whom to send to or receive from. As the number of agents in the system goes up (and hence the number of neighbors for each agent in this fully connected architecture), the higher the chance that agents will find themselves sending messages to others who will not themselves get around to receiving those messages immediately.

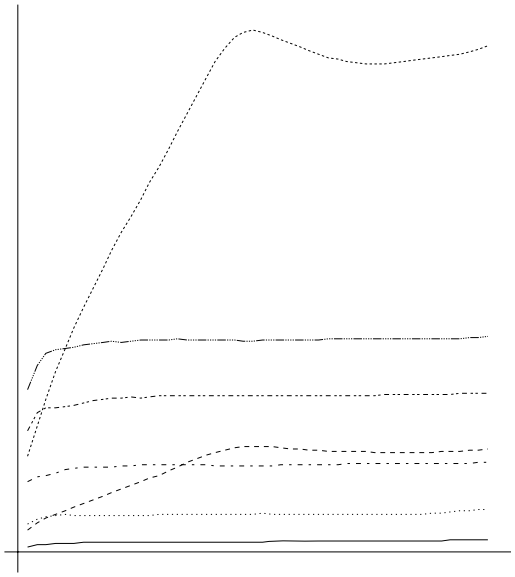


Figure 14: Fully-Connected Result Trends (no delay)

This additional delay until the receivers actually attempts to read messages from specific senders would be part of the elapsed time our experiments measure; that delay will be proportional to the number of neighbors of each agent, which in this case is tightly linked to the number of agents in the system. We can confirm this explanation by looking at the results for the "council" architecture, described below.

Figure 14 shows the average message round-trip time as a function of the number of messages processed so far per agent during each trial run, for the three different sets of experiments described in this section (fully-connected agent graphs, under no delay, 50 msec, or 100 msec agent-cycle delays). The graphs for the centralized tests show the point at which the central server in the centralized message-passing scheme hits overload, and how average message delay increases rapidly after that. In comparison, the graphs for the distributed tests show how the distributed solution avoids this problem. In particular, the graph of average message round-trip time for a system that is keeping up with the message load should be flat: a flat line means that there is no message deliver bottleneck within the system. Positive slopes mean that the more messages are processed by the

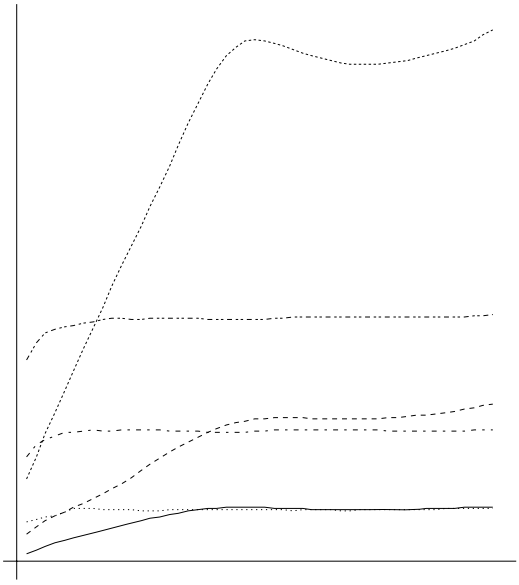


Figure 15: Fully-Connected Result Trends (50ms delay)

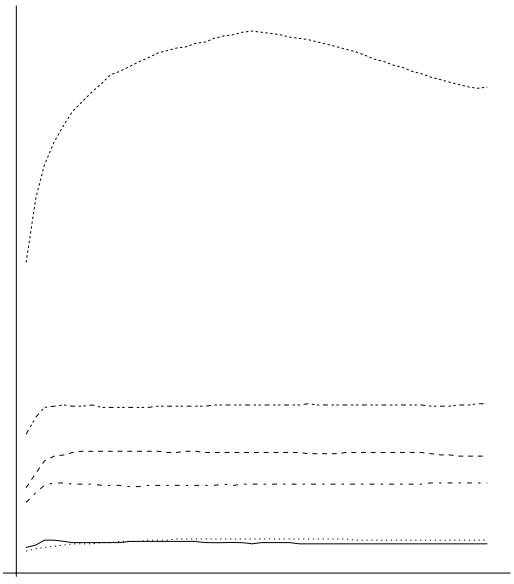


Figure 16: Fully-Connected Result Trends (100ms delay)

system, the more each message takes to deliver. This could be explained by imagining a message delivery bottleneck, where messages queue up for delivery. If messages arrive at the bottleneck faster than they can be processed, then messages will accumulate at the bottleneck. Assuming constant time delivery of messages, later messages will have to wait longer in the queue, and hence their round-trip delivery time will be higher. Note that a larger positive slope will correspond to a system that is worse off, in terms of keeping up with the message traffic traveling over it. (Negative slopes would mean the system becomes ever more efficient at delivering messages, as a function of the number of messages delivered so far.)

Figure 14 shows how the centralized system comprised of 3 agents (labeled "Centralized-3") is able to keep up with its message traffic, while the centralized systems comprised of 6 and 10 agents labeled "Centralized-6" and "Centralized-10", respectively) fail to keep up. The distributed systems, on the other hand, have no trouble keeping up with the message load in all three scenarios (3, 6, and 10 agents fully connected). Figure 15 shows what happens when a 50 msec agent cycle delay is added to all agents in the system. We can see how the 50 msec delay introduced had little appreciable effect on the centralized testbeds: messages still arrive at the central server faster than they can be processed, and so the message round-trip travel times increase as the number of messages processed by the system increases. The distributed testbeds are still insensitive to the size of the agent configuration tested: the only effect of the delay is to uniformly delay all results, since there is an additional delay element introduced that affects all transactions equally. (Not surprisingly, the additional message delivery delay corresponds to the agent cycle delay introduced.) Finally, Figure 16 shows what happens when a 100 msec delay is introduced, in every iteration of the agents' main loop. Here we can see how a delay of 100 msec is in fact sufficient to allow the message server in the centralized tests to keep up with the message traffic passing through it. All results are now flat, indicating that message travel times are no longer affected by the configuration and size of the agent networks tested. The distributed implementations are still more efficient than their centralized counterparts, and the difference between the two implementations increases as the size of the agent networks tested increases.[9]

[9] The ramp-up in all results is due to the fact that the agents start communicating with each neighbor as soon as each connection is set up, without waiting for the others to appear. Since agents do not start up simultaneously, message traffic gradually rises initially until all agents are active, then shrinks gradually as agents finish communicating.

### 3.1.6 "Council"-oriented architecture

The "council" architecture is a two-level arrangement of agents, as seen in Figure 11. First, a set of agents known as the *councillors* are connected to form a fully-connected graph. Then, each of the councillors is also connected to a second fully-connected network of *peers*. The arrangement is intended to represent *mixed* agent-oriented architectures, where a set of controllers cooperatively decide what to do, then each controller communicates those decisions as appropriate to their individual subnet. [10]

The experimental results obtained for the CoRaCLe implementations of this architecture for a variety of cases are presented in Tables 2, 3, and 4. Table 2 summarizes the results of a series of experiments run on systems with varying numbers of councillors and peers per councillor, where there was no agent cycle delay at any of the agents in the system. Tables 3 and 4 show the same set of experiments, except the agent cycle delay was set at 50 msecs and 100 msecs, respectively. In these tables, the configuration named $N$ – $M$ indicates a test system where there were $N$ councillors, each of which was fully connected to the other councillors and to $M$ peers. Each table presents 4 sets of numbers for each configuration[11]. In addition to the average message round-trip time for all messages exchanged in that configuration, we also present average round-trip times for messages exchanged in 3 different cases: when sender and recipient are both councillors ($c \rightleftharpoons c$), when sender and recipient are both peers ($p \rightleftharpoons p$), and when they are in opposite classes ($c \rightleftharpoons p$). The results were discriminated like this to discover whether the number of connections associated with particular agents had an effect of the message round-trip times that agent experiences, or whether it was in fact the case that message round-trip times depended on other variables.

When discussing the results for the fully-connected tests, we concluded that the performance of a centralized message-passing system would be related to the size of the system, measured in terms of the overall number of connections. The performance of a distributed system seemed to be related to the number of agents, or the number of connections per agent (which could not be distinguished from each other in the fully-connected case).

[10] We could imagine constructing other possible arrangements by using a strict hierarchical organization for either the councillors or the set of peers associated with each councillor. In this paper, we will concentrate on the "council" architecture we describe.

[11] Table 4 only includes averages, to preserve space. Complete results can be found in [Lejter, 1996]

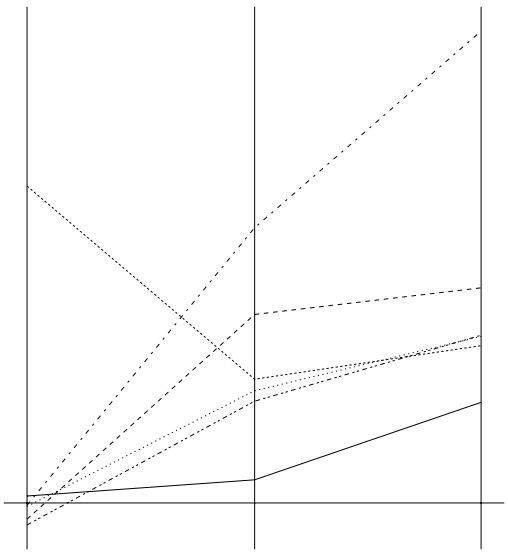


Figure 17: Detailed Results: Council architecture, 3 councillors, varying number of peers per councillor.

In the council tests, the configuration of the system can be described as a collection of fully-connected subnets hooked together. Each councillor participates in a subnet, fully connected with its peers; each councillor also participates in a subnet made up only of the other councillors, also fully connected.

Our fully-connected experiments suggest that the performance on the council networks, when discriminated into these individual subnets, should be similar to that of the equivalent stand-alone subnets. Any differences would be due to the interaction between the subnets, at each of the councillor agents. Some of the results from this discriminated analysis are shown in Figures 17 and 18. Figure 17 shows a graph of the results obtained for all test runs where the council set included three agents, while the number of members of each councillor's circle varied over three, six, or nine agents. Figure 18 shows the opposite graph, that is, one where the number of agents in each councillor's circle is fixed at three, while the number of councillors varies over three, six, or nine councillors.

These results provide additional evidence for some of the conclusions we had stated earlier: the performance of the distributed system tends to

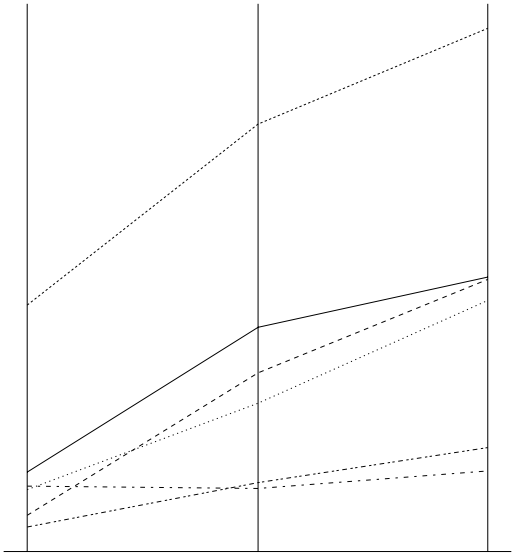


Figure 18: Detailed Results: Council architecture, 3 peers per councillor, varying number of councillors. (Y-axis uses log scale.)

be independent from the overall size of the system, all other things being equal. In Figure 17, we see how the results for the distributed case suffer far less of an impact as the size of the system (and in particular, the size of the individual councillor subnets) increases, (In particular, performance tends to level out.). As was the case in the fully-connected tests, performance in the centralized tests is far more affected by the size (or number of connections) of the system being tested.

The result listed for the centralized case with 3 councillors, each with a subnet of size 3, looks somewhat anomalous: the plot of message delay for the communications between the councillor and its peers seems to go counter to all the other plots. This is an indication of our observation above that the interactions between subnets have an effect on the performance of the system. By discriminating the resulting data into the three kinds of connections possible we can see how the different kinds of connections are not affected equally, as the size of the system increases. This provides additional evidence for the observation that it is necessary to consider the "shape" of the system, and not just its size, when analyzing its performance.

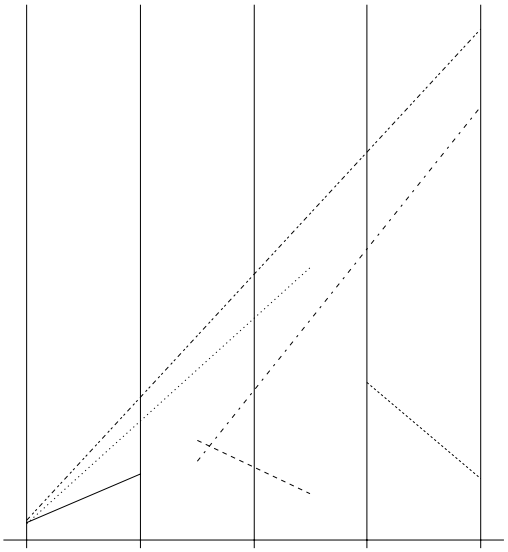


Figure 19: Council results by number of connections: Central. Each line links the results for the kinds of agents present in a given architecture.

In Figure 18, we vary the sizes of the councillor subnet, for a fixed subnet for each councillor of size 3. We can clearly see how the specific connections affected by the change (the increased size of the councillor subnet) are specifically only those connections between councillors and the individual members of each of their subnets. (Yet more evidence for the effect of the actual arrangement of agents within the network on its performance.) In fact, the results for the those connections for the centralized test case were so affected that the graph is shown in log scale, to more easily distinguish the other plot lines against those results: plots towards the bottom of the graph, or flat plots, indicate little effect; plots towards the top of the graph, or significant slopes, indicate more of an effect. The results for the connection between councillors and their peers appears highest in the graph, and with the steepest slope of them all.

Figures 19 and 20 present the same data as Table 2, except sorted this time by the number of connections established by the agents in the system. We can see how this measure can be used for coarse predictions (increased number of connections will affect the performance of the system, and dis-

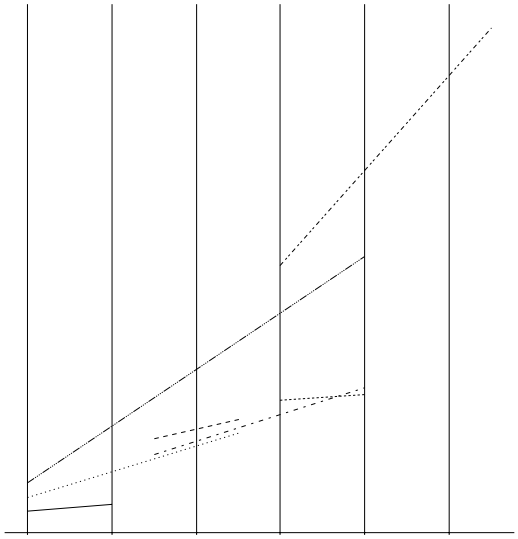


Figure 20: Council results by number of connections: Distributed. Each line links the results for the kinds of agents present in a given architecture.

tributed systems are affected less than the corresponding centralized systems) [12]. We can also see how this measure alone is not sufficient - it does not take into account the "shape" of the system.

### 3.1.7 Tree-oriented architecture

The results from our experiments on the tree-architecture systems can be seen in Table 5. We ran a series of tests with a 3-level complete tree with either two, three, or four children per node, along with a 4-level tree with either two or three children per node and a 5-level tree with two children per node. These results also appear in graph form in Figure 21.

These results confirm our conclusions based on the fully-connected and council architectures, without adding any additional detail. Once again, we can see how the centralized system outperforms the distributed one for small test systems, but suffers more as the size of the system increases.

[12] Note that the y-axes of these two graphs, which measure message round-trip delays, are not to the same scale

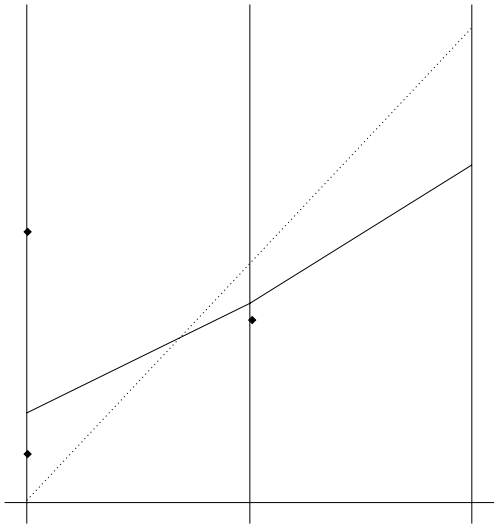


Figure 21: Results for Tree architecture

### 3.2 Overhead of the Framework

Even after choosing to use our system, designers would be justifiably curious to know what kind of overhead the different features of our communications framework incur in actual usage. This information would be necessary to find the best trade-offs between the functionality available in our framework and the requirements of the system to be constructed. Table 6 presents our measurements of the overhead incurred by the different abstractions involved in the Central and Distributed realizations of CoRaCLe discussed in this paper, respectively. There are two kinds of overhead involved: the fixed overhead incurred regardless of which realization it is, which is due to the generic CoRaCLe abstractions; and the realization-specific overhead incurred by each realization above and beyond the cost that would have been incurred by a dedicated implementation that directly used the same implementation represented by that realization.

### 3.3 Comparison with Other Systems

Initiatives to provide developers with the support necessary for designing and implementing distributed applications can be categorized into three

different groups, according to the level at which such support is to be incorporated.

First, we have those who believe that this support should be incorporated at the operating system level. The key advantage to this approach is that of efficiency: by incorporating support for distributed applications at such a low level, designers can tune operating system services for optimal performance. The disadvantages have to do with portability and compatibility with existing operating systems, programming languages, and hardware: all these become a load for the designers of that new operating system. Nonetheless, several attempts have been made to develop operating systems specifically for distributed computation: the *x*-kernel[Peterson et al, 1990], for example, or the V[Cheriton, 1984], Amoeba[Tanenbaum, 1995] and Sprite[Ousterhout et al, 1988] operating systems (or see [Tanenbaum and Renesse, 1985], [Tanenbaum, 1995]).

Second, we have those who believe that this support should be incorporated at the programming language level, providing the best set of abstractions for distributed processing within the language to be used for developing these applications. Ideally, such languages would be built on top of any distributed-processing-friendly operating systems, described above. However, the fact that the facilities are defined at the programming language level means that it is possible to provide them, even at the cost of implementing in the language's runtime whatever the underlying system services leave out. This approach is somewhat more portable than the first one, since we are moving away from the hardware level. It also results in a better environment for developers, since the programming language can be designed to provide ideal abstractions for distributed processing. The principal drawback to these programming languages is that they are *new*, typically incompatible with existing languages. This increases the initial effort required, until developers are familiar with the new language. Languages such as Linda [Gelernter, 1985], Argus [Liskov, 1988], *Obliq* [Cardelli], and E [EC, 1996] are examples of this second approach.

Third, we have those who attempt to incorporate this support into existing programming languages and operating systems - among which we place our own efforts. The advantages to this approach are that we can leverage off existing expertise, smoothing the transition to distributed development paradigms. The drawbacks are that the abstractions to be introduced are constrained to fit within the language's model (along with support that might have to be added to the language's runtime). These constraints introduce potential "impedance mismatches" between the exist-

ing language's underlying model and the requirements placed on it by the new distributed processing abstractions. Remote procedure calls [Birrell and Nelson, 1984], Sun Microsystems' ToolTalk [Julienne and Holtz, 1994], FIELD's communications framework [Reiss, 1990], ARPI's CPE testbed initiative, the Transis system[Amir et al,1992], PVM [Sunderam, 1990], [Geist et al, 1994], and our CoRaCLe framework, are examples of this third approach. In this section, we will concentrate on comparing our framework's performance against that of some of these other systems: Sun Microsystems' ToolTalk, FIELD's communications framework, and ARPI's CPE testbed environment.

ToolTalk and FIELD's framework are both communications frameworks based around a centralized message server, just like CoRaCLe's Central implementation. However, their implementation varies: FIELD's communications are built on top of straight TCP/IP, while ToolTalk is built on top of RPC. All three frameworks support *message-pattern based* delivery of messages. Message-pattern delivery based systems are logically broadcast systems in which the sender of a message need not specify its destination: agents interested in receiving a certain class of messages must register a pattern that describes that class of messages with the central message server. The server will forward to each agent only those messages that match any one of the patterns registered by that agent. In the case of CoRaCLe, this mode of delivery is provided in addition to the standard addressed delivery.[13] The message patterns can be

- regular expressions to apply to the source, target, or text of messages transmitted (in the case of CoRaCLe),
- agreed-upon values for components of a message (in the case of ToolTalk),
- format patterns to match against the text of each message, like the format patterns used by the standard C library's `scanf()` routine (in the case of FIELD).

The developers of ToolTalk were aware of the limitations on centralized systems demonstrated in the previous section. ToolTalk supports the ability to run multiple synchronized message servers, so as to distribute the load of delivering messages to several agents. This strategy allows them

[13] All our tests in the previous section used addressed delivery only. Pattern-based delivery would only have slowed things down, since pattern matching would have been necessary on each message sent.

to increase the number of simultaneous agents/messages they can manage before hitting overload. The developer of FIELD chose a different approach - the FIELD message server is far simpler than either ToolTalk's or CoRaCLe's central servers. This allows the FIELD message server to be implemented efficiently enough to negate much of the impact of having a centralized server.

FIELD also supports several implementation for the underlying communications subsystem in a way that makes them transparent to the users of FIELD. The main difference between the abstractions presented by FIELD and those presented by CoRaCLe is that in the case of FIELD, the communications subsystem is effectively a "black box": developers can choose which of the message transfer mechanisms supported by FIELD they wish to use, but that is all they can choose. There is no notion of modifying or extending any of the aspects of the communications subsystem without rewriting portions of the system. [14] CoRaCLe, on the other hand, exposes all the abstractions used to define the communications layer to the applications using it. This makes it possible for developers to modify or extend aspects of CoRaCLe when appropriate, by simply providing alternative implementations of the abstractions to modify that continue to obey the semantics defined by the CoRaCLe abstractions. CoRaCLe takes advantage of the object-oriented nature of its implementation language to make this process straightforward.

ARPI's Common Prototyping Environment (CPE) is a technology integration infrastructure designed to test the interoperability of various technologies being developed under the Arpa/Rome Labs Planning Initiative (ARPI), a project to advance research into knowledge based planning and scheduling sponsored by ARPA and Rome Labs. The CPE testbed includes as a component a communication framework designed to ease communications among the different modules that participate in this initiative. Unlike the two frameworks mentioned above, the CPE testbed communications framework is built around a point-to-point communications model. Like those other two, the CPE framework is also designed as a library built on top of an existing programming language - in this case Common Lisp.

[14] The implementation of FIELD is such that even at the programming level its communications facilities appear as a black box: there are no provisions built into FIELD to provide the programmer feedback on their use of FIELD's communications facilities.

### 3.3.1 Performance results

The developers of the CPE testbed constructed a set of benchmark tests, to evaluate the performance of their communications framework under varying conditions. For consistency, the first set of results we will present are thus that same set of benchmarks, applied to the other three frameworks under discussion: ToolTalk, FIELD's framework, and CoRaCLe's Central and Distributed implementations. The measures of performance tested by the developers of the CPE testbed to evaluate their communications framework was the message round-trip delay, under different network conditions, for a single pair of agents, and the actual CPU time consumed to transmit the message. This is a good measure of the efficiency of the implementation of the communications framework and the underlying operating system services and hardware implementation. It is not a good measure of the performance of the system once it scales up, as we have seen, but it is a starting point.

The test setup for this set of experimental results is different from the one discussed in Section 3.1.4; these experiments were modeled on the benchmarks presented by the CPE researchers. To gather data on the raw performance of the different frameworks, we designed two processes, a produced and a consumer: the producer would send a message to the consumer and await its reply, before repeating; the consumer would wait for a message from the producer and then reply to that message, before repeating. Each message contains the time at which the sender originated the message. Each receiver, on receipt, can compute the time the message spent in transit. The time reported for each message includes both processing at the sender and receiver processes and the network propagation time itself; if we assume that the network propagation time is consistent for any message sent under similar conditions, then a comparison based on this message transit times would be valid - the numbers are biased to include that propagation time, but the bias is uniform for all results.

Table 7 presents the results of the CPE testbed benchmarks for all the systems tested; it shows both the average running time (real time, user time, and system time) per message, and the average message round-trip time per message. The CPE testbed results measured message delay in two different situations: when the sender and receiver agents were in the same host, but using the network interface, and when the two agents were in different hosts, but connected to the same subnet. We also consider a third case, where the two agents are in different hosts, connected to different

subnets of the same net. In all cases, "real time" is the average real time taken to transmit a message from sender to receiver, while "user time" and "sys time" are the average user and system time (CPU time spent within the agent, and CPU time spent by the operating system on behalf of the agents) required to transmit a message.

### 3.3.2 Scalability results

The second set of results we will present are *scalability* results: the results obtained when running a subset of the experiments described in Section 3.1 on test systems built with these other frameworks.[15] This will allow us to compare the different frameworks under the same conditions used to evaluate the performance of CoRaCLe's implementation in Section 3.1. For this comparison, we concentrated on the fully connected system described in Section 3.1.5. For testing, we simply replicated the configurations used then. For the individual nodes in each network, we ported the original peer agent used in all the CoRaCLe tests to the other frameworks (thus eliminating again the actual implementation of the test agent as a factor).

It was necessary to introduce one change in the behavior of the test agent for these experiments, however. In the experiments described in Section 3.1, our test agent would randomly choose whether to send or receive a message, then randomly choose a specific source or destination for that operation. This was intended to model best the behaviors of interacting agents, which presumably would be interested in information from specific sources, or commands to specific targets. Neither ToolTalk nor FIELD have convenient mechanisms for specifying that only messages from a particular source are desired on any one read request. In the case of ToolTalk, patterns can be registered (to receive messages matching that pattern) and unregistered (to cease receipt of such messages) - but unregistering a pattern discards all queued but unread messages associated with that pattern. In the case of FIELD, messages that do not match any existing patterns are simply discarded - thus our agents would potentially lose messages every time a pattern was temporarily unregistered. To get around this limitation, the test agent used for the experiments discussed below was modified to match: each agent still chooses at random whether to send or receive a

[15] We will only present results for ToolTalk and FIELD's framework, since we do not yet have access to the CPE testbed. [Lejter, 1996] provides comparisons with these and other systems.

message; outgoing messages are still sent to a target selected randomly for each message, but incoming messages are accepted regardless of source.

This change should only improve on the performance of both of the CoRaCLe realizations discussed: message reads are now guaranteed to succeed as long as there is some message available, from any source – in our previous experiments, message reads would only succeed if there were messages from the sender explicitly requested on each call. This increases the likelihood that any one message read will succeed, thus reducing the average time to respond to messages. Table 8 presents the results obtained for both CoRaCLe realizations under the original, "directed read" algorithm, versus the new, "read from anywhere" algorithm. We can see that indeed the change in the agent's algorithm resulted in improved performance for our system.

Table 9 presents the results of the scalability benchmarks for all the systems tested, using the new "read from anywhere" algorithm. For comparison purposes, Figure 22 includes the results shown in Table 9, along with the original results we obtained for the two realizations of CoRaCLe using the directed read algorithm for our test agent. We can see how the performance of CoRaCLe improved as we changed the test agent's algorithm. We can also see how FIELD pays the performance penalty associated with centralized servers we discussed earlier: performance drops down as the size of the system increases, under conditions of heavy load. FIELD suffers this effect since its implementation must examine each incoming message against all patterns registered, for simplicity of implementation. On the other hand, we see how it would seem that ToolTalk does not suffer this problem – even though it, too is a centralized system.

ToolTalk and the Centralized realization of CoRaCLe use different implementation strategies. In the case of CoRaCLe, the increasing performance hit as a function of system size as described above is due to the fact that agents must query the central server for incoming messages, or even for discovering whether they have messages to retrieve. ToolTalk, on the other hand, is implemented so that it uses two independent communications channels to communicate between the clients and the ToolTalk central server: an RPC-based channel is used to transmit data to/from the server, but a socket-based channel is used for the server to notify each client that it might have incoming messages available at the server [16]. The implemen-

[16] ToolTalk does not guarantee that a notification will imply there is actually a message queued to be read

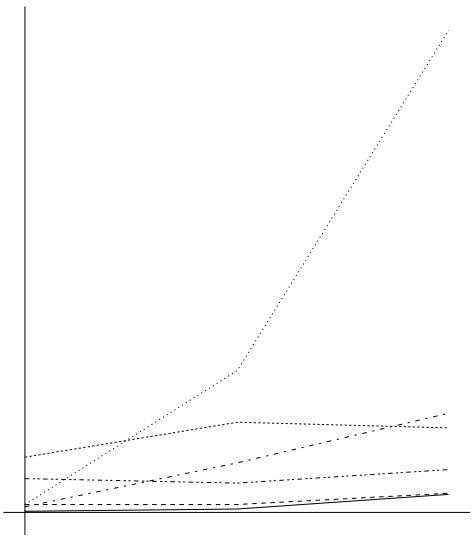


Figure 22: Scalability Results

tation of the socket mechanism allows clients to detect whether they have messages queued at the server in a way that does not involve the server on each query, thus reducing the load on the server to only actual message traffic. This explains why ToolTalk performance remains flat for fully-connected networks of sizes up to 10, while CoRaCLe's Central realization is impacted with fully connected systems of as little as 6 agents. ToolTalk does not avoid this drawback of centralized systems, but merely postpones it.

# 4 Implementation of the System

Our system was implemented as a series of class libraries, written in C++. Three class libraries were written: CoRaCLe (providing the low-level communications abstractions), DisPeL (providing the middle-level distributed control abstractions), and CAF (providing the high-level coordination abstractions). They were all developed using Sun SparcStations running Solaris 2.4. However, the code itself should be portable to any other version of UNIX that also includes a C++ compiler that implements the ANSI C++ Draft Standard for the language. To date, a port to Linux 1.3.45 running on

IBM 486-class machines is also available.

In addition to the core C++ libraries, we also provide code that allows developers working in C, Scheme, and Common Lisp to take advantage of the facilities described here. We also provide a set of tty-based and GUI-based tools to help trace and debug systems built on top of our abstractions.

To ease the task of developers interested in our system, we have written user and reference manuals for the different parts of our system, along with example applications in several domains.

The three different layers (CoRaCLe,DisPeL, CAF) are available as three individual UNIX `tar` files from `ftp://ftp.cs.brown.edu/pub/dai`. Each tar file contains the code corresponding to that layer, plus the corresponding reference and user manuals. There are also a few sample applications, to illustrate the use of CoRaCLe, as an independent `tar` file.

## 5 Conclusions and Future Work

This paper describes a high-performance software system that supports distributed computing and multi-agent coordination. Our system provides the facilities necessary for experimenting with a variety of choices at different levels of abstraction, to find the best match for the system under development.

We have so far provided only two realizations of the CoRaCLe framework. One of the areas for future work is the development of other realizations, to provide a greater choice of message transport and agent directory services within the CoRaCLe framework. In particular, directory services with more powerful search capabilities, and hierarchical directory services, would be welcome additions.

As far as the higher-level control and coordination abstraction are concerned, we need to gain more experience with them, implementing a wider variety of distributed systems using them. This should lead us to even higher-level, more convenient frameworks to use to construct distributed systems. An important feature lacking from our high-level abstractions so far is a common agent language; this should be one of the first additions to our abstraction hierarchy.

## Bibliography

[Amir et al, 1992] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki, "Transis: A Communication Subsystem for High Availability,", Hebrew University of Jerusalem Tech Report CS91-13, April 30, 1992.

[Andrews, 1991] Gregory R. Andrews, "Paradigms for Process Interaction in Distributed Programs," ACM Computing Surveys, 23: 49–90.

[Birrell and Nelson, 1984] A. D. Birrell, and B. J. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, 2:39–59.

[Brattman, Israel, and Pollack, 1988] M. E. Brattman, D. J. Israel, M. E.Pollack, "Plans and resource-bound practical reasoning, " Computational Intelligence, 4:349–355.

[Brooks, 1985] Rodney A. Brooks, "A Robust Layered Control System for a Mobile Robot," MIT A.I. Memo 864.

[Cardelli] Luca Cardelli, "A Language with Distributed Scope".

[Cheriton, 1984] D. R. Cheriton, "The V kernel: a software base for distributed systems.", IEEE Software, April 1984, pages 19-42.

[EC, 1996] Electric Communities, "The E Programming Language", http://www.communities.com/e/index.html

[Dean et al, 1990], Thomas Dean, Kenneth Basye, Robert Chekaluk, Seungseok Hyun, Moises Lejter, Margaret Randazza, "Copying with Uncertainty in a Control System for Navigation and Exploration". Proceedings of AAAI-90, 1990.

[Ferguson, 1992] I. A. Ferguson, "Towards an Architecture for Adaptive,

Rational, Mobile Agents,: E. Werner, ed., Decentralized AI 3 - Proceedings of the Third European Workshop on Modeling Autonomous Agents and Multi-Agent Worlds, 249–262, Elsevier.

[Geist et al, 1994] A. Geist, A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam, "PVM: A User's Guide and Tutorial for Networked Parallel Computing" Cambridge, MIT Press, 1994.

[Gelernter, 1985] D. Gelernter, "Generative communication in Linda," ACM Transactions on Programming Languages, 7: 80–112.

[Hayes-Roth, 1985] Barbara Hayes-Roth, "A Blackboard Architecture for Control", Artificial Intelligence, Vol. 26, 1985, pages 251-321.

[Jennings, 1996] N. R. Jennings. "Coordination Techniques for Distributed Artificial Intelligence," in Foundations of Distributed Artificial Intelligence (eds. G. M. P. O'Hare and N. R. Jennings), Wiley, 1996, pages 187-210.

[Julienne and Holtz, 1994] Astrid M. Julienne and Brian Holtz, "ToolTalk and Open Protocols: Inter-Application Communication," Sunsoft Press, 1994.

[Lejter, 1996] Moises Lejter, "A Framework for the Development of Multi-Agent Architectures." Brown University Department of Computer Science, Ph.D. Thesis, 1996 (forthcoming).

[Lesser, 1991] Victor R. Lesser "A retrospective view of FA/C Distributed Problem Solving" IEEE Transactions on Man, Machine, and Cybernetics, Vol. 21, pages 1347-1363.

[Liskov, 1988] Barbara Liskov, "Distributed Programming in Argus", Communications of the ACM, Vol. 31, Num. 3, March 1988, pages 300-312.

[Ousterhout et al, 1988] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent W. Welch, "The Sprite Network Operating System," IEEE Computer, February 1988, pages 23-36.

[Peterson et al, 1990] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao, "The *x*-kernel: A Platform for Accessing Internet Resources," IEEE Computer, May 1990, pages 23-32.

[Reiss, 1990] Steven P. Reiss, Connecting Tools using Message Passing in the FIELD Program Development Environment, IEEE Software, July, 1990.

[Sen, 1996] Sandip Sen, "An automated distributed meeting scheduler," accepted for publication in IEEE Expert.

[Smith, 1980] Reid G. Smith"The Contract Net Protocol: High-Level Communications and Control in a Distributed Problem Solver", IEEE Transactions on Computers, Vol. C29, No. 12, December 1980.

[Sunderam, 1990] V. S. Sunderam, "PVM, A Framework for Parallel Distributed Computing", Practice and Experience, Vol. 2, Num. 4, December 1990, pages 315-339.

[Tanenbaum and Renesse, 1985] Andrew S. Tanenbaum and Robbert van Renesse, "Distributed Operating Systems", ACM Computing Surveys, Vol. 17, N. 4, December 1985, pages 419-470.

[Tanenbaum, 1995], Andrew S. Tanenbaum, "Distributed Operating Systems" Prentice Hall, New Jersey, 1995.

[Urken, 1990] Arnold B. Urken, "Coordinating Distributed Action via Agent Voting" ACM, OIS90, 1990.

[Wellman, 1993] Michael Wellman, "A market-oriented programming en-

vironment and its application to distributed multicommodity flow problems." Journal of Artificial Intelligence Research, 1:1-23, 1993.

| Config | Case | Message Round Trip (msecs) | |
|---|---|---|---|
| | | Central | Distributed |
| 3-3 | c ⇌ c | 2252 | 1455 |
| 3-3 | c ⇌ p | 12750 | 1897 |
| 3-3 | p ⇌ p | 1957 | 1288 |
| 3-3 | **Average** | 7427 | 1634 |
| 3-6 | c ⇌ c | 2803 | 8419 |
| 3-6 | c ⇌ p | 6209 | 5814 |
| 3-6 | p ⇌ p | 11324 | 5470 |
| 3-6 | **Average** | 9193 | 5762 |
| 3-9 | c ⇌ c | 5428 | 9315 |
| 3-9 | c ⇌ p | 7360 | 7666 |
| 3-9 | p ⇌ p | 17977 | 7698 |
| 3-9 | **Average** | 15342 | 7735 |
| 6-3 | c ⇌ c | 10124 | 6308 |
| 6-3 | c ⇌ p | 83652 | 4616 |
| 6-3 | p ⇌ p | 1904 | 2042 |
| 6-3 | **Average** | 35367 | 4917 |
| 6-6 | c ⇌ c | 23253 | 17332 |
| 6-6 | c ⇌ p | 75751 | 9697 |
| 6-6 | p ⇌ p | 9010 | 4544 |
| 6-6 | **Average** | 30113 | 7843 |
| 6-9 | c ⇌ c | | 31206 |
| 6-9 | c ⇌ p | | 28054 |
| 6-9 | p ⇌ p | | 15529 |
| 6-9 | **Average** | | 19123 |
| 9-3 | c ⇌ c | 17078 | 16678 |
| 9-3 | c ⇌ p | 223285 | 13395 |
| 9-3 | p ⇌ p | 2302 | 2910 |
| 9-3 | **Average** | 73883 | 13773 |

Table 2: Experimental Results: Council Architecture

| Config | Case | Message Round Trip (msecs) Central | Distributed |
|---|---|---|---|
| 3-3 | c ⇌ c | 2018 | 3364 |
| 3-3 | c ⇌ p | 8766 | 12855 |
| 3-3 | p ⇌ p | 712 | 602 |
| 3-3 | **Average** | 4928 | 7419 |
| 3-6 | c ⇌ c | 2745 | 4834 |
| 3-6 | c ⇌ p | 3074 | 11038 |
| 3-6 | p ⇌ p | 1971 | 7576 |
| 3-6 | **Average** | 2364 | 8486 |
| 3-9 | c ⇌ c | 5048 | 4461 |
| 3-9 | c ⇌ p | 5243 | 8701 |
| 3-9 | p ⇌ p | 4440 | 12789 |
| 3-9 | **Average** | 4630 | 11680 |
| 6-3 | c ⇌ c | 3059 | 8082 |
| 6-3 | c ⇌ p | 63124 | 59319 |
| 6-3 | p ⇌ p | 612 | 923 |
| 6-3 | **Average** | 24456 | 25412 |
| 6-6 | c ⇌ c | 8003 | 11565 |
| 6-6 | c ⇌ p | 47312 | 35353 |
| 6-6 | p ⇌ p | 3943 | 5260 |
| 6-6 | **Average** | 16914 | 14759 |
| 9-3 | c ⇌ c | 5923 | 12973 |
| 9-3 | c ⇌ p | 183951 | 104882 |
| 9-3 | p ⇌ p | 835 | 795 |
| 9-3 | **Average** | 56061 | 37493 |

Table 3: Experimental Results: Council Architecture - 50 msec delay

| Config | Case | Message Round Trip (msecs) | |
|---|---|---|---|
| | | **Central** | **Distributed** |
| 3-3 | **Average** | 5935 | 10029 |
| 3-6 | **Average** | 3785 | 10443 |
| 3-9 | **Average** | 5791 | 18540 |
| 6-3 | **Average** | 38621 | 46001 |
| 6-6 | **Average** | 12487 | 22550 |
| 6-9 | **Average** | | 22371 |
| 9-3 | **Average** | 65942 | 62092 |

Table 4: Experimental Results: Council Architecture - 100 msec delay

| # Nodes | # Conn | Message Round Trip (msecs) | |
|---|---|---|---|
| | | **Central** | **Distributed** |
| 3-2 | 3 | 10290.860 | 21906.331 |
| 3-3 | 3 | 0.000 | 36607.202 |
| 3-4 | 3 | 73364.593 | 55097.587 |
| 4-3 | 6 | 0.000 | 34317.682 |
| 5-2 | 10 | 46180.929 | 16492.103 |

Table 5: Experimental Results: Tree Architecture

| Reason | Time (msec/call) | |
|---|---|---|
| Central | Read: 0.1 | Write: 0.2 |
| Distributed | Read: 42 | Write: 6 |

Table 6: Overhead for the realizations of CoRaCLe

| Test | Framework | Time (msecs) Running Time real | user | system | Round Trip |
|---|---|---|---|---|---|
| Different subnets | Dedicated RPC | | 3.6 | 3.5 | |
| | Dedicated IPC | | 50.0 | 24.8 | |
| | CoRaCLe Central | 43.8 | 4.6 | 1.9 | 18.8 |
| | CoRaCLe Dist | 51.7 | 5.0 | 2.1 | 27.0 |
| | ToolTalk | 53.5 | 5.4 | 2.6 | 26.5 |
| | FIELD | 69.6 | 29.1 | 30.5 | 57.8 |
| | CPE Testbed | | | | |
| Same Subnet, 2 Hosts | Dedicated RPC | | 3.1 | 1.7 | |
| | Dedicated IPC | | 18.6 | 12.5 | |
| | CoRaCLe Central | 40.8 | 4.3 | 1.9 | 15.9 |
| | CoRaCLe Dist | 41.3 | 4.3 | 1.8 | 15.9 |
| | ToolTalk | 29.3 | 6.7 | 1.0 | 23.2 |
| | FIELD | 84.8 | 33.7 | 32.1 | 56.3 |
| | CPE testbed | 301.3 | 108.6 | 45.1 | |
| Same Host, Networked | Dedicated RPC | | 3.2 | 1.7 | |
| | Dedicated IPC | | 19.6 | 11.8 | |
| | CoRaCLe Central | 22.2 | 3.1 | 2.05 | 17.8 |
| | CoRaCLe Dist | 21.3 | 3.0 | 2.1 | 16.9 |
| | ToolTalk | 3.1 | 0.1 | 0.2 | |
| | FIELD | 26.5 | 13.1 | 12.3 | 51.2 |
| | CPE testbed | 331.9 | 85.3 | 17.1 | |
| Same Host, Memory | CoRaCLe Local | 4.9 | 2.3 | 0.1 | |

Table 7: Performance Results - CPE Testbed Benchmarks

| CoRaCLe | | Time (msecs) | | |
|---|---|---|---|---|
| Mode | algorithm | Full-3 | Full-6 | Full-9 |
| Central | directed | 287.1 | 2330.9 | 11406.3 |
| | anywhere | 62.8 | 166.8 | 1013.0 |
| Distributed | directed | 976.8 | 2035.8 | 3584.2 |
| | anywhere | 446.7 | 450.5 | 1058.7 |

Table 8: Scalability Results under both algorithms

| | Time (msecs) | | |
|---|---|---|---|
| Framework | Full-3 | Full-6 | Full-9 |
| CoRaCLe Central | 62.8 | 166.8 | 1013.0 |
| CoRaCLe Dist | 446.7 | 450.5 | 1058.7 |
| ToolTalk | 3173.1 | 5139.4 | 4954.9 |
| FIELD | 405.4 | 8131.0 | 27684.8 |

Table 9: Performance Results - Scalability Benchmarks