Piecing Together the Magic Mirror : a Software Framework to Support Distributed, Interactive Applications

by

Diane E. Hirsh

B.A., Boston University (2004)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

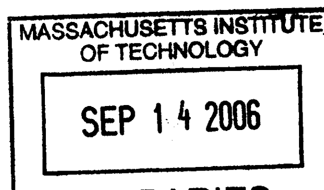
September 2006

© Massachusetts Institute of Technology 2006. All rights reserved.

Author ____
Program in Media Arts and Sciences
August 4, 2006

Certified by ____
Dr. V. Michael Bove Jr.
Principal Research Scientist
MIT Media Laboratory
Thesis Supervisor

Accepted by ____
Andrew B. Lippman
Graduate Officer
Departmental Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
SEP 14 2006

ROTCH

Piecing Together the Magic Mirror : a Software Framework to Support Distributed, Interactive Applications

by

Diane E. Hirsh

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 4, 2006, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

Developing applications for distributed platforms can be very difficult and complex. We have developed a software framework to support distributed, interactive, collaborative applications that run on collections of self-organizing, autonomous computational units. We have included modules to aid application programmers with the exchange of messages, development of fault tolerance, and the aggregation of sensor data from multiple sources. We have assumed a mesh network style of computing, where there is no shared clock, no shared memory, and no central point of control. We have built a distributed user input system, and a distributed simulation application based on the framework. We have demonstrated the viability of our application by testing it with users.

Thesis Supervisor: Dr. V. Michael Bove Jr.
Title: Principal Research Scientist, MIT Media Laboratory

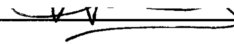
Piecing Together the Magic Mirror : a Software Framework to Support Distributed, Interactive Applications

by

Diane E. Hirsh

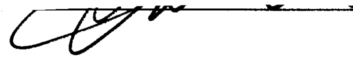
The following people served as readers for this thesis:

Thesis Reader



Dr. Pattie Maes
Associate Professor of Media Technology
MIT Media Laboratory

Thesis Reader



Dr. David Cavallo
Research Scientist
MIT Media Laboratory

Contents

Abstract **3**

1 Introduction **17**

2 Patterns for Managing Data Flow and Control **25**

2.1 Guaranteeing Agreement 26
2.2 Creating Unique Identifiers 26
2.3 Broadcast 27
2.4 Spewing 28
2.5 Handshaking 30
2.6 Handshaking with Timeouts 31
2.7 Queries for specific data 32
2.8 Queries for streams of data 33
2.9 Keep-alives 34
2.10 Semaphores 36
2.11 Semaphores with Timeouts 37

3 Framework Description **39**

3.1 Messaging Subsystem 39
3.1.1 Primary application interface 41
3.1.2 Networking details and persistent connections 42
3.1.3 Keeping track of other hosts 45
3.2 Data Backup/Retrieval and Process Management Services 45
3.2.1 Process Management 46
3.2.2 Primary application interface for the Process Manager 46
3.2.3 How it works 47
3.2.4 Data Backup Services 49
3.2.5 Primary application interface for the Data Backup services 49
3.2.6 How it works 50
3.3 Data Aggregation Subsystem 51
3.3.1 Primary application interface 55
3.3.2 How it works 57

4 Applications **61**

4.1 Distributed Simulation 62
4.2 Data Aggregation Instance : Distributed Ball Tracking 66
4.3 A Distributed User Input System 70
4.4 An Interactive Application 76

5 Evaluation **81**

6 Conclusion and Future Work **89**

A System and Services Summary **93**
A.1 Is this system for you? 93
A.2 Summary of Services 94

B PointerList API **97**
B.1 PointerList <YourTypeHere*> 97

C Messaging API **105**
C.1 Message 105
C.2 Message Types 108
C.3 Messenger 110
C.4 MachineNode 116
C.5 MessageQueue 120
C.6 MachineNodeMgr 121
C.7 Connection 123

D Network Client Interface **129**
D.1 Internal Commands 129
D.2 Network Commands 130

E Telephone **131**

F Process Manager / Backup Server Interface **133**
F.1 Program Management 133
F.2 Backup Service Interface 136
F.3 Package 138
F.4 PackageQuery 139
F.5 DataBackup 139

G Data Aggregation API **143**
G.1 Sensor 143
G.2 SensorDatum 145
G.3 SensorDataStrand 147
G.4 SensorDataAggregate 151
G.5 Aggregator 154

H Ball Tracking API **163**

H.1 TrackingMgr 163
H.2 Ball 165
H.3 BallTrack 166
H.4 User 168

I Cursor Input API **171**
I.1 CursorCoordinator 171
I.2 CursorPosition 176
I.3 Cursor 177
I.4 Stroke 179

J Paint **181**

List of Figures

1-1 The tiles are shown running the distributed simulation. There are a number of instances where pieces of different agents appear across multiple tiles. . . 19

2-1 Summary of properties, advantages and disadvantages of broadcast. 28
2-2 Summary of properties, advantages and disadvantages of spewing. 29
2-3 Summary of properties, advantages and disadvantages of handshaking. . . . 30
2-4 Summary of properties, advantages and disadvantages of handshaking with timeouts. 31
2-5 Summary of properties, advantages and disadvantages of using queries for specific data. 33
2-6 Summary of properties, advantages and disadvantages of using queries for streams of data. 34
2-7 Summary of properties, advantages and disadvantages of keep-alives. 35
2-8 Summary of properties, advantages and disadvantages of semaphores. . . . 36
2-9 Summary of properties, advantages and disadvantages of semaphores with timeouts. 37

3-1 The basic structure of a program using the Messenger and its associated services. 42
3-2 The Messenger has two MessageQueues, which serve as the interface between it and the networking threads that do the sending and receiving (the client and server threads). Arrows show the flow of data through the Messenger. . 43
3-3 The Messenger has two MessageQueues, which serve as the interface between it and the networking threads that do the sending and receiving. Here, the sending and receiving are done both by the main client and server threads, and by auxiliary sender and receiver threads, which are dedicated to connections with a particular host. Arrows show the flow of data through the Messenger. 44
3-4 This figure shows the flow of control when the Process Manager receives a request to start an application. The Process Manager starts a wrapper program, which in turn starts the application. The wrapper program waits for the application's exit code, and sends a message to the Process Manager. 48
3-5 This figure shows the flow of data through the Process Manager when an application wishes to back up data. The Process Manager randomly selects two other hosts to receive copies of the data. 51

3-6 When the Process Manager (re)starts, it sends out a query to other hosts on the network. Nodes that receive the query, which have data sent by the host, send a copy to the Process Manager. The information is then available for the application to retrieve. 52
3-7 This figure shows the structure of the various objects inside the Aggregator. The Aggregator contains SensorDataAggregates (hexagons), which are comprised of SensorDataStrands (octagons). SensorDataStrands, in turn, are comprised of SensorDatum (circles). The different colors represent data from different hosts/sensors. 53
3-8 This figure shows how the Aggregator functions over the network. The colors represent data from different sources. Each of the three Aggregators has a sensor that is a different color, but by exchanging data, they are all able to aggregate data from all of the possible sources. 54
3-9 The basic structure of a program using the Data Aggregation system. This program uses the callback registry to receive regular updates. 56

4-1 This figure shows the simultaneous state of four tiles participating in data aggregation, with the distributed ball tracking. The hexagon with the brightest color represents the local view of the ball. The darker colored hexagons represent remote views of the ball. 68
4-2 This figure shows the state of the cursor, where the position is determined by integrating the velocity of observed balls. The rectangle shows the field of view of the camera. The cursor is able to move outside of the field of view of the camera because the tiles are cooperating. 69

5-1 This figure shows the agents moving around in the simulation. 81
5-2 This figure shows the two modes for editing interactions, in the first version of the application. The selection mode is shown on top. The interaction mode is shown on the bottom. 83
5-3 Faulty data aggregation, due to inadequacies in the data association function, leads to multiple aggregates created for a single object, which leads to multiple cursors for the same ball. SensorDataStrands in the same SensorDataAggregate are represented by hexagons of the same hue. 87

C-1 The dependency diagram for objects in the Messaging subsystem. 106

E-1 This figure provides pseudo-code for a simple telephone receiver. 132
E-2 This figure provides pseudo-code for a simple telephone sender. 132

F-1 The dependency diagram for objects in the Process Management and Data Backup subsystems. 134

G-1 The dependency diagram for objects in the Data Aggregation subsystem. . 144
G-2 This figure shows the basic structure of a program using the Data Aggregation system. This program uses the callback registry to receive regular updates. Modified from figure 3.3.1. 155

H-1 The dependency (and inheritance) diagram for objects in the Ball Tracking subsystem. 164

I-1 The dependency diagram for objects in the distributed Cursor Input subsystem. 172

Acknowledgements

This work was sponsored by the Things That Think, Digital Life, and CeLab consortia. I'd like to thank my advisor, Mike Bove, and my committee members Pattie Maes and David Cavallo. I'd also like to thank present and previous members of the Object-Based Media group, who helped in putting this system together, and who helped by being a sounding board for ideas : Jacky Mallet, Ben Dalton, Arnaud Pilpre, James Barabas, Jeevan Kalanithi, Dan Smalley, and Quinn Smithwick. I'd also like to thank the many people who participated in my user study. Their help was invaluable.

My fiancé, Michael Theriault, has been wonderful and supportive through these trying times. Thanks to my parents, who have always put my education at the forefront of their priorities. I'd also like to thank Margrit Betke at Boston University for giving me the proverbial "chance."

Thanks everybody!

Chapter 1

Introduction

The purpose of this thesis is to present a software framework to support distributed, interactive, collaborative applications that run on collections of self-organizing, autonomous computational units. We have chosen to think in terms of very general mesh network style distributed computing. Units have no shared clock, no shared memory, and we assume that latency is unbounded. We have also avoided strict client/server models. We do assume that we are dealing with IP-enabled computers, but there is no reason that the framework could not be adopted to run with any network protocol where units can be identified uniquely. Important benefits of the mesh network model are that distributed nodes can gather more data and process more information than any single computer could, there is no single point of failure, and such systems are generally very extensible. However, important drawbacks are that there is no single point of control, and such systems can be very difficult and subtle to program and debug.

The purpose of the software framework presented in this thesis is to facilitate the development of distributed applications by providing services and software modules that will reduce the burden on the application programmer. The framework consists of four sets of services, packaged in three subsystems.

The Messaging subsystem is the basis for all other systems in the framework. It provides a suite of services to asynchronously exchange messages with other hosts on the network. Its

purpose is to allow the application programmer to deal with messages in an asynchronous, event-driven manner. This serves to relieve the programmer of the need to handle the details of sending and receiving data over the network.

The Process Management and Data Backup services, which are contained in the same software module, provide support for fault tolerance in applications. The purpose of these systems is to provide services that will help the programmer deal with faulty programs and systems. The Process Management services enable a host to react to critical hardware or operating system faults by rebooting itself automatically when applications report critical errors. The Data Backup services provide a way for applications running on nodes to recover their state if they are terminated or if the node they are running on is rebooted. Data sent to the backup server is propagated out to other nodes on the network. When the node restarts, the backup server gathers backed up data from other nodes on the network, and makes it available to the application.

The Data Aggregation system has been built to provide support for distributed sensing and sensor network applications. It provides a service that will produce, distribute, and gather data on behalf of the application. It provides a set of containers for sensor data, and an interface where applications can define criteria to associate pieces of data together. Its purpose is to help the application programmer by providing a pattern for data aggregation, which the application programmer can fill in with details.

The intended platform for our framework is the smart wall tiles of the Smart Architectural Surfaces project. The smart wall tiles have been built using handheld computers with 400 MHz XScale processors running the ADS version of debian Linux, version 2.4.7. The computers have been equipped with a 15" LCD screen, an array of sensors (including a camera and a microphone), and an 802.11 PCMCIA wireless card. The tiles are mounted in rails on the wall. Our goal has been to develop an interactive application for the tiles, where the tiles are able to work together. To build this interactive system, we wanted to be able to use the data from lots of sensors, together, to understand the intent of the user. It is also important to be able to dynamically add and remove nodes from the system, and handle hardware, operating system, and network failures.

Figure 1-1: The tiles are shown running the distributed simulation. There are a number of instances where pieces of different agents appear across multiple tiles.

The specific application we have developed is a distributed simulation where agents of various types move around and interact in the system. The tiles work together, so that the agents are not restricted to a life on any one particular tile. Users interface with the system using a set of brightly colored balls. Using the ball, users can work together to add agents of different types to the system, and set up attractions between different types. The simulation we have built is very simple but serves as a microcosm of the types of systems that could be built on top of our framework. The possibilities range from simulations of cars in traffic, to protein folding. We show the tiles, with the simulation running, in figure 1-1.

There are a number frameworks for supporting distributed applications. CORBA [22] and DCOM [21] are frameworks for creating and using interoperable distributed software com-

ponents in heterogeneous environments. Jini [23] is another system, which relies on Java. Jini provides a resource look up service and discovery protocol, in addition to a distributed event model, transaction processing, and resource leasing. The primary difference between Jini and CORBA and DCOM is that Jini uses the code mobility capabilities of the Java language to directly move components around the network, whereas in CORBA and DCOM, components have predefined interfaces that must be agreed upon ahead of time. Hive [18] is a system, similar to Jini, which implements applications as an ecology of distributed agents. Hive agents use local resources and communicate with each other to realize applications. The difference between Hive and Jini is that Hive provides a separation between code that is local and trusted, and code that is networked and untrusted. Hive also uses a location based model, where Jini focuses on services, without regard for location.

The part of our system most related to these frameworks is the messaging subsystem. Our messaging system is much simpler than these frameworks. On the one hand, it does not provide support for object-based communication, advertising resources, transaction processing, etc. On the other hand, it does not impose a particular object model on the programmer. Our Messaging subsystem is really just a step above TCP – it is a tool to send and receive messages, which an application could use to implement an object-based protocol, such as CORBA or DCOM. Our system could not be used to implement Jini, since our system is not based on Java.

The Data Aggregation portion of our framework is most closely related to the Context Toolkit, described in [5]. The Context Toolkit is based, conceptually, on the toolkits used for developing GUIs. In the Context Toolkit, there are a number of abstract objects, which serve as a conceptual framework for supporting context-aware applications. Context widgets interface with sensors. Interpreters abstract the information acquired from widgets (translating coordinates into symbolic description, for example) and combine multiple pieces of context to produce higher-level information. Aggregators gather logically related information and make the data available within a single software component. Discoverers are responsible for maintaining a registry of resources available in the environment. In the paper, they lay out these various objects as a pattern for context-aware applications to

follow, but they do not describe how these objects could or should be implemented. Our Data Aggregation system implements functionality related to the Context Toolkit's context widgets, interpreters, and aggregators.

The framework we have created is, by no means, limited to the intended platform or the application that we ultimately developed. There are many areas of computing where our framework could apply, including ubiquitous and pervasive computing, context-aware computing, ad hoc networks, sensor networks, and computer-supported collaborative work.

Ubiquitous computing describes a vision for the future of computing, where computation has moved out of the conventional desktop and into the environment [1]. There is a body of research in ubiquitous computing systems which overlaps with Computer-Supported Collaborative Work, where researchers have sought to create technologies that allow users to work together. Users interact with such systems (which often take the form of electronic white boards, with various accoutrements) in the environment, and not with a mouse and keyboard on a desktop computer. One such system is the Tivoli system [19], which is designed to support small working meetings, where participants work closely together on a problem or idea. Another such system is the ClearBoard system [14], which aims to allow users to seamlessly move between a shared workspace and interpersonal space. A more recent system is the Stanford iRoom [15] which is an interactive meeting space where displays are embedded as part of the physical environment to support meetings.

There is another body of ubiquitous computing literature that has moved towards context-aware pervasive computing. The premise of such systems is that "enabling devices and applications to automatically adapt to changes in the surrounding physical and electronic environments will lead to an enhancement of the user experience," [5]. Context is defined broadly as "any information that characterizes a situation related to the interaction between users, applications and the surrounding environment," [5]. Many such systems use location as the primary context, especially older systems like the PARC Tab system [24]. Other location-based systems are surveyed in [13]. More ambitious systems, such as the CMU Aura project [10] seek to provide a sort of personal assistant that can help a user by discovering resources and presenting information, or by anticipating a user's needs and

taking action. All of this takes place in a very heterogeneous computing environment. Our work is related to context aware ubiquitous computing because our system of smart wall tiles exists on the wall of a room, and users interact with the tiles in the environment, rather than by using a mouse and keyboard.

Ubiquitous computing systems are intimately related to ad hoc networking. An ad hoc network is "a collection of communication devices that wish to communicate, but have no fixed infrastructure, and have no pre-determined organization or available links," [20]. In an ad hoc network, nodes must relay packets for each other, since nodes may not have direct connectivity with each other. Routing algorithms for ad hoc networks need to be able to adapt to changes in link connectivity and quality, which may be due to node mobility, or power consumption constraints. An open problem in ad hoc networking is handling scalability. As the number of nodes in the network increases, communication protocol overhead increases. To address the increasing overhead of ad hoc routing protocols, a hierarchical, multi-level approach may be adopted [4].

Sensor Networks employ a large number of densely deployed sensors, equipped with computational power, in order to monitor or collect information about some phenomenon [2]. Our work is related to Sensor Networks, because we are using many cameras, working together, to track objects in a room. Networks with computationally heavier sensors, such as camera networks have less stringent networking requirements than other types of sensor networks with extensive routing overhead, but still must handle a lot of data. To balance the quality of sensors with their cost and bandwidth needs, a multi-tiered network may be adopted [16]. A multi-tiered network is a sensor network with different layers, each layer with a different modality. Information from the different layers is combined together to use resources efficiently, while producing high quality data. An example is a camera network which has different types of cameras. The lower layer, with more nodes, uses inexpensive, low resolution cameras, which then activate more expensive, higher quality cameras when something interesting happens. To conserve resources (bandwidth, power) in sensor networks, nodes may negotiate with each other before sending data, to ensure that only useful data is sent when disseminating information [12]. Many sensor networks use compression schemes to

save bandwidth when representing and transmitting data; however, the irregular spatial distribution of nodes, and irregular sampling times may adversely affect these compression schemes [9].

The main challenge of Computer Supported Collaborative Work is "enabling people to work effectively together in teams mediated through computers," [17]. There are many familiar collaborative systems, such as video-conferencing and electronic meeting rooms [11], and document authoring and management systems [7]. A framework for building collaborative visualization applications that runs on collections of PCs is described by Wood et al in [25]. This framework is limited because it assumes that, at each location, there is a single user/computer, and it also assumes a centralized server. Our work is related to Computer Supported Collaborative Work because we have created a system where users in one or more separate physical locations can create conditions to collaboratively work on a simulation.

In all application areas where computation is distributed, applications must confront the problem of attaining consensus among all of the participating nodes. The archetypal consensus problem is simply getting a group processes in a distributed system to agree on a value. Consensus protocols must provide agreement, termination, and non-triviality. The Fischer consensus impossibility result says that it is impossible to guarantee agreement among a collection of asynchronous processes, if any of them can fail undetectably [8]. There are three types of asynchrony to consider: *processor asynchrony*, where a process can be delayed for an arbitrarily long time, while other processes continue to run, *communication asynchrony*, where message delivery latency is unbounded, and *message order asynchrony*, which means that messages can arrive out of order. Dolev et al [6] extend this work by identifying the minimum conditions under which consensus can be guaranteed. In addition to the three types of asynchrony described in [8], they identify two properties that are favorable for creating consensus: *atomic broadcast*, where a node can send a message to every other node, in one time step, and an *atomic receive/send* where nodes can both receive messages and send them out again in the same time step. The authors identify exactly four cases where consensus can be guaranteed, each of which requires at least one type of synchrony, identified above. Since the Fischer consensus impossibility result was discovered,

many authors have circumvented it, using three main strategies: randomization, stronger timing assumptions, and failure detectors. In [3], Aspnes describes a number of randomized protocols for gaining consensus.

There are three main contributions of our work. The first contribution is the implementation of a method for disseminating and gathering data from a large collection of sensors. The second is the implementation of a messaging module which serves to free the programmer from the network model imposed by TCP, while only minimally imposing its own model on the programmer. The third contribution of our work is the combination of systems for process management, and data backup/retrieval.

The remainder of this thesis is organized as follows. Chapter 2 describes general patterns for managing data and control. These are patterns we identified, recurring throughout the development of the framework and subsequent applications. Chapter 3 describes the three modules of the framework, including their overall purpose, the application interface, and their inner workings. Chapter 4 describes the applications built for the system, including the distributed simulator, distributed ball tracking, distributed input system, and finally, the interactive simulation application. Chapter 5 describes the user testing that we conducted to test the application which was developed to demonstrate the viability of the framework. In chapter 6, we discuss our conclusions and directions for future work.

Chapter 2

Patterns for Managing Data Flow and Control

This chapter describes a collection of techniques for managing data in distributed applications. Books and papers on distributed computing focus on formal algorithms and elaborate middleware. A discussion about simple techniques for handling data and control is lacking in the literature. Since it is impossible to know what sort of data an application might need to exchange, we did not roll these patterns into a software module, but we have included a discussion of them so that others may benefit from our experience. They will be highlighted in applications where they are used in chapter 4.

In our application, we used TCP enabled Linux computers that communicate with each other over 802.11 wireless. However, the techniques we discuss here assume only that all nodes can communicate with each other in some form, and all nodes can be identified uniquely. Even if communication happens indirectly (by routing the communication through other nodes, for example), the techniques we describe should still apply, although modifications might be needed to adopt them for special networking situations.

The applications we aim to support with these techniques are any type of application where processes running in different locations (generally, without shared memory), need to share

data. Data sharing may take many forms. In the simplest case, one process may simply need to periodically inform another process of some value or set of values. In a more complex case, two processes might have a vested interest in, or may even be sharing control of the state of some object. In this case, the processes will need to constantly inform each other of each other's actions, and negotiate about what action should be taken next.

2.1 Guaranteeing Agreement

It is impossible to guarantee that nodes will reach agreement in the presence of node failure. However, even in the absence of node failures, agreement can be very difficult to achieve when the behavior of the network is unpredictable e.g. dropped packets, delayed packets, out of order packets. Given this, it is possible, to some extent, to choose the way in which hosts will disagree. If it is, to some degree, known how the hosts might disagree, it is possible to manage disagreements by having mechanisms in place for renegotiating when disagreements arise.

We can minimize disagreements by using as much local information as possible (e.g. local address, local time, local coordinate frame) and then using intelligence of some sort to go between the local spaces of different hosts. Using local information aids extensibility. Mechanisms for reaching global agreement are error-prone, and can take a long time to resolve themselves. Additionally, having a global system in place ahead of time (for example, a global coordinate frame) can make it difficult to add new hosts to the system at a later date.

2.2 Creating Unique Identifiers

It is very important, especially in a distributed application, for everyone to be talking about the same thing. Things (pieces of data) need to be tagged in some unique way, so that everyone knows what they are talking about. The alternative is to have only one thing that everyone is talking about, which simplifies things a great deal.

Throughout the framework and in subsequent applications described in this thesis, the chosen method for generating globally unique identifiers is to use combinations of locally available identifiers. For example, a piece of data could be tagged uniquely by a combination of the address of the originating host, and some id number generated on the originating host. Our method for generating identifiers assumes that hosts can be uniquely identified in some way, for example, IP address, MAC address, serial number, GPS coordinates, RFID tags, or with some other arbitrary scheme. This scheme means that more bits must be dedicated to identification than may strictly be necessary, but offers the benefit that every host can generate as many identifiers as it likes without needing to ask anyone else for help or approval. Unique identifiers can be generated quickly, at the price of a few bytes per piece of data.

If space is at a premium and time is not, an alternative method for generating identifiers is to have some pool of numbers shared among all the hosts. Whenever a host needs a unique identifier, it pulls one out of the pool. This scheme allows identifiers to be very small, but it provides a myriad of potential headaches. First, it is extraordinarily easy for two or more hosts to grab the same number at the same time. Assuming that the hosts all detect this, while sorting out who should get what identifier, someone else might grab one of the identifiers that they are negotiating about. Using some sort of semaphore-like scheme (described more below), it can take a long time for the hosts to certify each other's id numbers. The best bet might be to use some sort of algorithm modelled on the ethernet access protocol; if two hosts grab the same number at the same time, they both let it go and try again. It will be more time-consuming, and more bandwidth intensive, to create the unique identifiers, but will allow them to be small. So, if the identifiers are generated once, traded often, and bandwidth is expensive, this scheme might make sense.

2.3 Broadcast

One of the simplest methods for transmitting data is to simply tell everyone we know about everything (broadcast). The main advantages are that it is very simple to program, and

Summary for Broadcast	
Programming Complexity: .	minimal
Book-keeping requirements:	none
Bandwidth usage:	significant
Handles faults:	no
Network assumptions:	weak
Advantages:	
Easy to throw data away	
Disadvantages:	
Lack of scalability	
Potentially wasted bandwidth	
Potentially wasted attention	

Figure 2-1: Summary of properties, advantages and disadvantages of broadcast.

there is no book-keeping that needs to be done. The main disadvantages are a lack of scalability, wasted bandwidth and wasted attention. This technique is most appropriate in a producer/consumer model, or when every host really has a vested interest in all of the data being exchanged.

In the absence of true network broadcast, if there is a very large number of hosts, sending everything to everyone may overload the network. Even if we do not overload the network, if not everyone on our receiving list cares about our data, we are wasting the time it took to send it to them, and we are making them spend time filtering the data they are receiving.

In the case where there may be network failures, this scheme provides no way to detect and respond to these failures – we cannot necessarily be sure that everyone got the data. On the other hand, we also make no assumptions about the expected latency of the network : it is okay for the network to be slow. Since we do not expect to hear back, if the host on the receiving end must throw data away, it is very easy – they do not need to respond to every piece of data that they see.

2.4 Spewing

Another simple technique for exchanging data is to tell other hosts about all of our data in a targeted way. The main advantages of this technique are that it is still very simple

Summary for Spewing	
Programming Complexity: .	minimal
Book-keeping requirements:	minimal
Bandwidth usage:	significant
Handles faults:	no
Network assumptions:	weak
Advantages:	
Easy to throw data away	
Disadvantages:	
Potentially wasted bandwidth	
Potentially wasted attention	
No information from receiver end	

Figure 2-2: Summary of properties, advantages and disadvantages of spewing.

to program, and the only book-keeping needed is to keep track of where we want to send the data. The main disadvantages of this technique are potentially wasted bandwidth and attention (as with broadcast), and no way to detect and react to network failures. As with broadcast, this is very appropriate in a producer/consumer model. It is also very appropriate when a potential query for data from a receiver is approximately the same size as the data to be sent. (Contrast this with using queries, as in section 2.7.) This technique differs from later techniques in that the sender *does not wait for a response* from the receiver.

As with broadcast, in the case of network failures, this scheme does not provide a way to detect and respond to those failures. On the other hand, it makes no assumptions about expected latency. Since we do not expect to hear back from our receivers, it is very easy for the receivers to throw data away without wasting time responding to it.

The catch is that nodes can get into nasty feedback loops with each other, since they are not communicating about the status on the receiving end. For example, if node A is trying to give something to node B, but node B doesn't want it, then node B will just give it back to node A, who will give it back to node B. Of course, you can write safeguards for this (if you see the same piece of data so many times, ignore it or do something different), but that increases programming complexity quickly (what if they really do mean to give it back to each other, and it is not just feedback?), and there are more direct ways that node B could

say "No", with handshaking, as described in 2.5.

2.5 Handshaking

Summary for Handshaking	
Programming Complexity: .	moderate
Book-keeping requirements:	minimal
Bandwidth usage:	moderate
Handles faults:	no
Network assumptions:	weak
Advantages:	
Information from the receiving end	
Disadvantages:	
Potentially wasted bandwidth	
Potentially wasted attention	

Figure 2-3: Summary of properties, advantages and disadvantages of handshaking.

A slightly more complicated, but more flexible technique is to send data to a receiver, who then sends back a response. The main advantage of this is that it allows the receiver to say what happened, while still requiring only minimal book-keeping on both ends. The main disadvantage is that we have now increased the bandwidth usage (by assuming/requiring acknowledgements and thus doubling the number of messages going back and forth), and we might still waste bandwidth and attention with unwanted data. The programming complexity of this technique is moderate. For each piece of data, the sender must send it, the receiver must receive it and formulate a response, and the sender must receive the response and react to it.

As with broadcast, in the case of network failures, this scheme does not provide a way to detect and respond to those failures. On the other hand, it makes no assumptions about expected latency. The demands on the receiver's attention have become more egregious, however, than with spewing, as in section 2.4, since receivers are assumed to send some sort of acknowledgement (although, at this point, there is nothing forcing them to do so.)

Let us consider the scenario from section 2.4, where node A is trying to give some unwanted thing to node B. Upon receipt, node B can simply send a response to node A saying that it

does not want the data, at which point node A can find something else to do with the data. To make the book-keeping very simple, node B would send a copy of the data back to node A, so that node A would not need to maintain a record of the data it originally sent.

2.6 Handshaking with Timeouts

Summary for Handshaking with Timeouts	
Programming Complexity: .	significant
Book-keeping requirements:	moderate
Bandwidth usage:	moderate
Handles faults:	yes
Network assumptions:	strong
Advantages:	
Information from the receiving end	
Handles network and node failures	
Disadvantages:	
Requires unique identifiers	
Potentially wasted bandwidth	
Potentially wasted attention	

Figure 2-4: Summary of properties, advantages and disadvantages of handshaking with timeouts.

A more complicated variant of handshaking, as in section 2.5, is to use time-outs to handle node failure. Although it seems like a simple modification from 2.5, the addition of time-outs significantly compounds the programming complexity of this technique. In this technique, the sender sends its data, making a record of the time when it sent it. When the receiver receives the data, it sends back a response. If the sender does not receive a response, it takes some other action, such as re-sending the data, or sending the data to someone else. Although more complicated than plain handshaking, the main advantanges of this technique are that we have now added vocabulary for nodes to talk about and respond to node and network failures, and information (most likely) will not get lost, and if it does get lost, it will at least be accounted for.

This is at the cost of significant programming complexity and increased book-keeping demands. Not only does the sender need to keep records of when it sent what to who, it also

needs to periodically check to see if any of those has timed out. Unique identifiers are no longer optional – sender and receiver must be talking about the same piece of data. Most important, handling timeouts can be very tricky. For example, the host on the other end might not have failed outright – the link might simply have gotten congested so that the message/response did not get through in time. The receiver must be able to handle the case when a response comes in for a piece of data that it had previously timed out. We also make a significant assumption about network latency – we must define how long we will wait for a response. There are techniques for determining this dynamically, but these techniques further increase programming complexity.

Let us consider the case where node A is trying to give some piece of data to node B, but assume, this time, that node B wants it. So, node A sends the data to node B. After a suitable waiting period, node A does not hear back, and so it sends the data to node C. However, after clearing a whole pack of messages spewed to it from node D, node B finally gets and responds to the data from node A. Now, both node B and node C think they have the one true copy of the data. Now what? We address problems of this type in section 4.1, when we discuss a simple resolution technique that we employ when duplicate agents arise in the distributed simulation.

2.7 Queries for specific data

In this moderately simple technique, receivers know what they want, and they know who to ask for it. The receiver sends a query to the relevant node, who then sends them the desired data. In this technique, the burden of intelligence is squarely on the receiver, who must know what to ask for (although, the receiver no longer needs to filter data being sent to it). The advantage of this technique is that it potentially makes very good use of network bandwidth, since the receiver will only get exactly what it wants, and so there will be little wasted bandwidth. The disadvantage is that it potentially makes very poor use of network bandwidth, since the nodes could end up exchanging double the number of messages. Book-keeping requirements are minimal; the receiver must be able to identify the thing it wants

Summary for Queries for specific data	
Programming Complexity: .	minimal
Book-keeping requirements:	minimal
Bandwidth usage:	minimal
Handles faults:	no
Network assumptions:	weak
Advantages:	
No wasted attention	
Disadvantages:	
Places burden on the receiver	
Bandwidth usage will depend on the situation	

Figure 2-5: Summary of properties, advantages and disadvantages of using queries for specific data.

using its unique identifier, and the sender must know what that means, but no one needs to keep records about what messages have been sent previously.

This technique is most appropriate for cases when there is a small number of pieces of data to be requested, and those pieces of data are large when compared with the size of the query needed to ask for them. It might seem strange that a receiver would know what it wanted, without actually having what it wanted. We will address a problem of this type in section 4.1, when we discuss what happens when a tile running the simulation encounters an agent with an unknown type.

2.8 Queries for streams of data

This technique is a bandwidth and attention saving variation of spewing, as described in section 2.4. In this technique, the receiver gives the sender a set of criteria for the information it wants to receive. When the sender has new data, it compares the data against the list of requests it has. If the data matches a request, it sends the data to the appropriate receiver. This technique makes very good use of network bandwidth since receivers only get the data they want, and they only have to send one request to get it. The book-keeping requirements are moderate, since the sender must maintain a list of all requests. This technique is used extensively in the applications developed in this thesis.

Summary for Queries for streams of data	
Programming Complexity: .	moderate
Book-keeping requirements:	moderate
Bandwidth usage:	minimal
Handles faults:	no
Network assumptions:	weak
Advantages:	
No wasted attention	
Minimal wasted bandwidth	
Disadvantages:	
Receiver must know what to ask for	
Sender must filter data to send	

Figure 2-6: Summary of properties, advantages and disadvantages of using queries for streams of data.

In this technique, the computational burden of filtering is shifted onto the sender. It is most appropriate when there is a small number of receivers. If there is a large number of would-be receivers, handling the queries for all of them could cripple a sender, and simple broadcast, as in section 2.3, might make more sense, especially if true network broadcast is available, and it is expensive to compute the given criteria. On the other hand, since sending data over the network is just about the most expensive thing to do in a distributed system, handling the queries for a large number of hosts might make sense, in the absence of true network broadcast, if the criteria is easy to compute.

2.9 Keep-alives

In the previous techniques, we were trying to solve a problem of how to get data across the network, while keeping everyone informed and not compromising the integrity of the data. Now, we will try to solve a slightly different problem. Here, two hosts are essentially sharing some piece of information; one host is in possession of the piece of data, the other host has a vested interest in the piece of data, and the two are trying to coordinate about it. There are at least two essential cases that might apply. In one case, the host wants to use some piece of data that another host has possession of, and so it sends periodic messages saying that it still wants the relevant piece of data. In the other case, one host has a vested

Summary for Keep-alives	
Programming Complexity: .	significant
Book-keeping requirements:	significant
Bandwidth usage:	moderate
Handles faults:	yes
Network assumptions:	strong
Advantages:	
Interested parties know data is valid	
Disadvantages:	
Bandwidth overhead for keep-alive messages	

Figure 2-7: Summary of properties, advantages and disadvantages of keep-alives.

interest in the data of another host, and the host in possession of the piece of data sends periodic messages to the vested hosts to let them know that the data is still good.

The primary advantage of using keep-alives is, believe it or not, its simplicity, when compared to the cost of keeping a large number of hosts informed about the whereabouts and state of some piece of data. Book-keeping for transferring data can get very complex, when handling unpredictable network behavior. Keep-alives provide a conceptually simple way for hosts to know if their data is still valid, while tolerating node and network failure.

Although simpler than the logic required to maintain state consistency as data moves around among many hosts, the programming complexity and book-keeping requirements of this technique are significant. The sender needs to keep track of who has a vested interest in what, and when was the last time it told them about it. The receiver needs to keep track, minimally, of when it last heard about each piece of data. Both the sender and receiver need to periodically check for time-outs: the sender, so that it can send the appropriate keep-alive messages, the receiver, so that it can react to any time-outs. Reacting to the time-outs is the most subtle and complex aspect of this technique. As with handshaking with time-outs, as described in section 2.6, nodes might not have failed outright and messages might simply have been delayed. It is necessary to be able to handle the receipt of a keep-alive message for a piece of data that has already timed out. In addition to the programming and book-keeping requirements, this technique requires significant bandwidth overhead (although the degree of the overhead depends on the frequency of the keep-alive messages.)

2.10 Semaphores

Summary for Semaphores	
Programming Complexity: .	significant
Book-keeping requirements:	significant
Bandwidth usage:	significant
Handles faults:	yes
Network assumptions:	weak
Advantages:	
Can get agreement, eventually	
Disadvantages:	
Agreement will take a long time	
Susceptible to host/network failure	

Figure 2-8: Summary of properties, advantages and disadvantages of semaphores.

Now we are trying to solve a problem of how to get everyone to agree. Since we have already established that we cannot guarantee that everyone will agree, we will simply do our best. In this technique, a host has something that it wants some number of other hosts to agree on. The host sends each of them a query, asking if the thing it wants is okay. Then, it waits for the other hosts to answer. The originating host must not do anything until everyone else has answered.

The main advantage of this technique is that you can get nodes to agree on a value eventually, if only by waiting long enough. The main disadvantage of this technique is that it can take a long time, especially if there are a lot of hosts. If N hosts try to run the same algorithm at the same time, there will be N^2 messages flying across the network. Depending on N, it may or may not choke the network. Furthermore, each host must respond to N queries, which, depending on N, may or may not take a lot of time.

The programming complexity and book-keeping requirements are moderate. For each piece of data the host is synchronizing on, it must keep a list of the hosts asked for approval (or, minimally, the number of hosts asked for approval). The host must also have handlers in place for any possible response. If the originating host asks if something is okay, and someone answers "No," there needs to be a reasonable response for that answer. This technique tolerates unbounded network latency, but if a host goes down, or if a message

gets lost, the whole process will fail to terminate. Using a list, instead of an integer, to do the book-keeping is favorable, since if a node failure is detected, we can then remove the affected host from the semaphore, while safeguarding against removing the same host multiple times from the semaphore. This might happen if we simply decremented an integer when detecting a failure, since a single failure might be detected multiple times.

This technique is applied in section 4.3, for the initial negotiation for control of a cursor in the distributed input system.

2.11 Semaphores with Timeouts

Summary for Semaphores with Timeouts	
Programming Complexity: .	significant
Book-keeping requirements:	significant
Bandwidth usage:	significant
Handles faults:	yes
Network assumptions:	moderate
Advantages:	
Can get agreement, eventually	
Less susceptible to network failures	
Disadvantages:	
Agreement will take a long time	
Susceptible to host failure	

Figure 2-9: Summary of properties, advantages and disadvantages of semaphores with time-outs.

Adding time-outs to the semaphore-style synchronization scheme does not dramatically increase the programming complexity, since the application already needs to poll the semaphore to find out if it is able to go ahead. Timeouts offer the advantage that they renders the semaphore-style scheme significantly less susceptible to network failure.

In this technique, the originating host sends a request out to all the other hosts on the network, asking if something is okay, making a list of the hosts where it sent the requests. Then, it waits for responses. If, after a certain amount of time, some hosts have not responded, the originating host asks them again. An important point is that the originating

host cannot timeout the other hosts' approval – it has to wait for them. If the originating host detects]] a failure of another host, then the other host can be removed from the semaphore without their approval (presumably). For this technique to work, the originating host needs a list of the hosts where it sent the requests; an integer is not sufficient.

The reason time-outs do not significantly increase the programming complexity, in this case, is because it is acceptable for an originating host to receive an answer from the same remote host twice. So, if the request times out, and is sent again, both requests might go through, and the host might answer twice. The first time the originating host receives an answer, the remote host will be removed from the semaphore (assuming they say "yes"). The second time the originating host receives an answer, the remote host will not be in the semaphore, and so the originating host will not need to take any action. In this particular case, we can eliminate a whole class of "what-ifs," and get the benefits of time-outs without the hairy programming headaches.

Chapter 3

Framework Description

The software framework presented in this thesis consists of four sets of services, in three subsystems. The Messaging subsystem is the basis for all other systems in the framework. It provides a suite of services to asynchronously exchange messages with other hosts on the network. The Process Management and Data Backup services, which are contained in the same subsystem, provide support for fault tolerance in applications. The Data Aggregation subsystem provides support for distributed sensing and sensor network applications.

3.1 Messaging Subsystem

The first problem faced by any distributed application is getting data back and forth among all of the processes participating in the application. The simplest approach is to use UDP, where the application simply drops packets into the network and hopes that they get where they are going. UDP offers the benefits of simplicity, but has no guarantees about reliability. Another approach is to use TCP, which has sophisticated mechanisms for ensuring that delivery is complete and in order. TCP offers reliability, at the cost of some overhead, in terms of programming complexity, and bandwidth use. The greatest drawback of TCP, in the author's opinion, is that TCP draws a very strong, and somewhat arbitrary line in the sand, between what process is a "sender" and what process is a "receiver." Such

delineations are necessary for TCP to work, but they are, nevertheless, annoying if you are developing a program that must both be able to send data to lots of other hosts, and receive data from lots of other hosts. What we want is some way to have the ease of use of UDP, with the reliability, flow control, and other positive aspects of TCP. We have created such a system, which we present in this section.

The messaging subsystem is the basis for all other systems in the framework. It facilitates the exchange of data by providing an object-oriented set of services to send and receive messages in an event-driven and asynchronous manner. Our system uses TCP with blocking I/O, but prevents the application from being delayed by the I/O by using threads. The messaging subsystem is a collection of six types of objects : Message, Messenger, MessageQueue, MachineNode, MachineNodeMgr, and Connection.

The main unit of currency in the messaging subsystem is the Message. The Message class is a container that allows applications to easily tag pieces of data, using six 32 bit integers. Two of the fields of the Message are the 'type' field and the 'extra' field. The type field is used by the Messenger for dispatching Messages to the application (to be discussed below). The extra field is for use by the application to store whatever information it needs. In the applications developed later in this thesis, the 'extra' field is used extensively for sub-typing. The payload of a message is a byte array (as opposed to a null-terminated string). The MessageQueue class is a thread-safe container for Messages, which serves as the interface between the main Messenger class and the threads that do the sending and receiving.

The Messenger is the main point of contact for the application. It enqueues outgoing messages for sending, dispatches incoming messages to the application's callbacks, manages the threads that do the sending and receiving, stores information about hosts on the network, and manages persistent connection setup / tear-down. The Messenger has, as data members, two MessageQueues (one for outgoing messages, one for incoming messages), and a MachineNodeMgr (to be discussed below).

3.1.1 Primary application interface

When the application has data to send, it packages its data as a Message, and then passes this Message to the Messenger. The Messenger provides wrapper functions to create and send messages in various forms. One of the methods allows the application to send data without ever interacting with the Message class; the application simply passes in the message payload, the length of the payload, the type, and the desired value of the extra field. The Messenger also has a method to allow the application to send a message to all known hosts with a single command. Once the application has passed a Message to the Messenger for sending, the application does not need to take any further action.

When packaging data as a Message, the application must provide a byte array for the payload of the message, the length of the payload, the type of message, and the receiver. The application also has the option of setting other tags in the Message. Each tag provides an opportunity for multiplexing the messages on the receiving end. The 'extra' field of the Message is used extensively in the framework and applications for sub-typing.

To receive messages, applications must periodically call Messenger::ProcessMessages(). When the application calls Messenger::ProcessMessages(), the Messenger iterates through the messages that have been placed on the incoming queue. The Messenger looks at the type of each message, and dispatches the message to the appropriate application callback function.

The Messenger provides a callback registry for different types of Messages. Applications can register callbacks for 20 different types. Predefined message types include : commands, notifications, queries, and errors. Callback functions take, as arguments, the Message itself, and any one piece of data which may be provided by the application.

To use the Messenger and its associated services, the application registers callbacks for its desired types, and then makes repeated calls to Messenger::ProcessMessages(). In the callbacks, or in between calls to ProcessMessages(), the application passes messages to the Messenger to send. The structure of a C++ program using the Messenger is shown in figure 3.1.1.

```
Messenger *messenger;
void callback(Message* message, void* data){
  //do work
}
int main(){
  messenger = new Messenger;
  messenger->RegisterCallback(MESSAGETYPE_NOTIFY, & callback, NULL);

  while(true){
    messenger->ProcessMessages();
    //do work
  }
}
```

Figure 3-1: The basic structure of a program using the Messenger and its associated services.

3.1.2 Networking details and persistent connections

There are two main threads created by the Messenger : a main client thread, and a main server thread. The main client thread polls the outgoing MessageQueue for messages to send. (It sleeps for a brief period when there are no more messages to send.) When it has a message to send, it sets up a TCP connection, sends the message, and then tears down the connection. If the client thread is unable to establish a connection to a remote host and send the message, it sends an error message to the local server thread. The server thread waits for incoming connections. When it receives an incoming connection, it accepts one message, places it on the incoming queue, and then tears down the connection.

This scheme may seem somewhat wasteful, since there is overhead associated with setting up and tearing down TCP connections. Also, if we have a lot of data for one host, setting up and tearing down TCP connections breaks TCP's guarantee that data will arrive in order. If we have a lot of data for one particular host, it might make sense to keep the connection open to the given host, and send all the available messages before tearing down the connection. However, this might result in starvation of the other hosts, if messages are being continuously generated by the application, which is running in another thread. A solution might be to keep an open connection to all other known hosts on the network for which we might have data, but consider that, if there are 1000 hosts on the network, it is

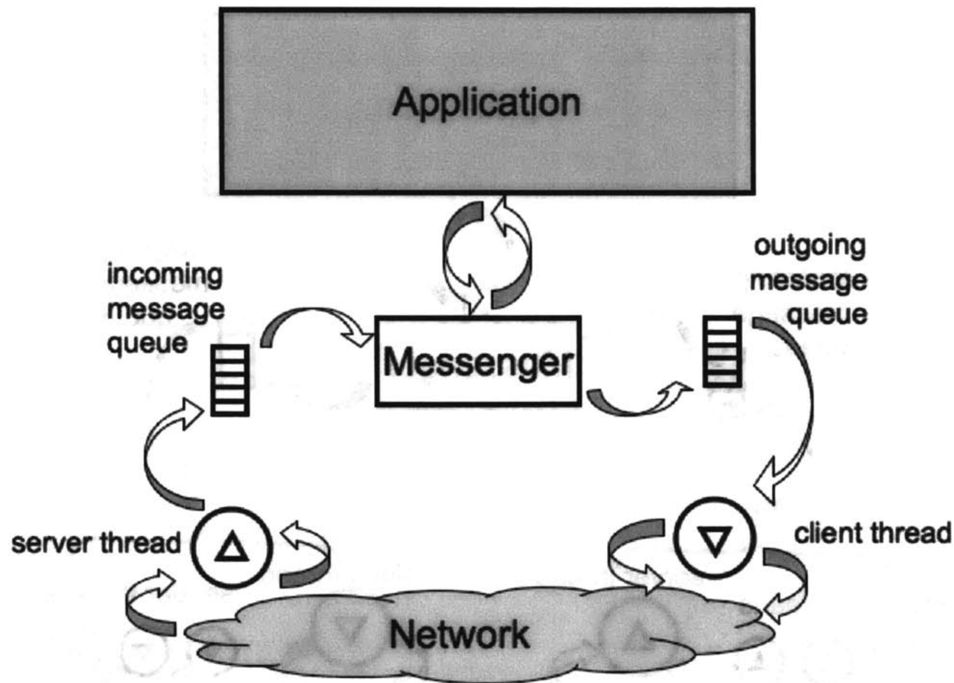


Figure 3-2: The Messenger has two MessageQueues, which serve as the interface between it and the networking threads that do the sending and receiving (the client and server threads). Arrows show the flow of data through the Messenger.

not practical to maintain a connection to each, nor does it make very much sense if we only have occasional data for most of those hosts. It might also make sense to have a separate thread sending each outgoing message, but this does not solve the problem of having the overhead of setting up and tearing down TCP connections.

The answer to these problems, adopted in this system, is that the Messenger provides a service where applications can request a dedicated, persistent connection to other hosts on the network. To start a connection to another host, the application makes a request to the Messenger, which then sets up the outgoing connection, sends a special connection message, and starts a sender thread dedicated to the particular host. The sender thread polls the outgoing MessageQueue for messages addressed to the host to which it is assigned.

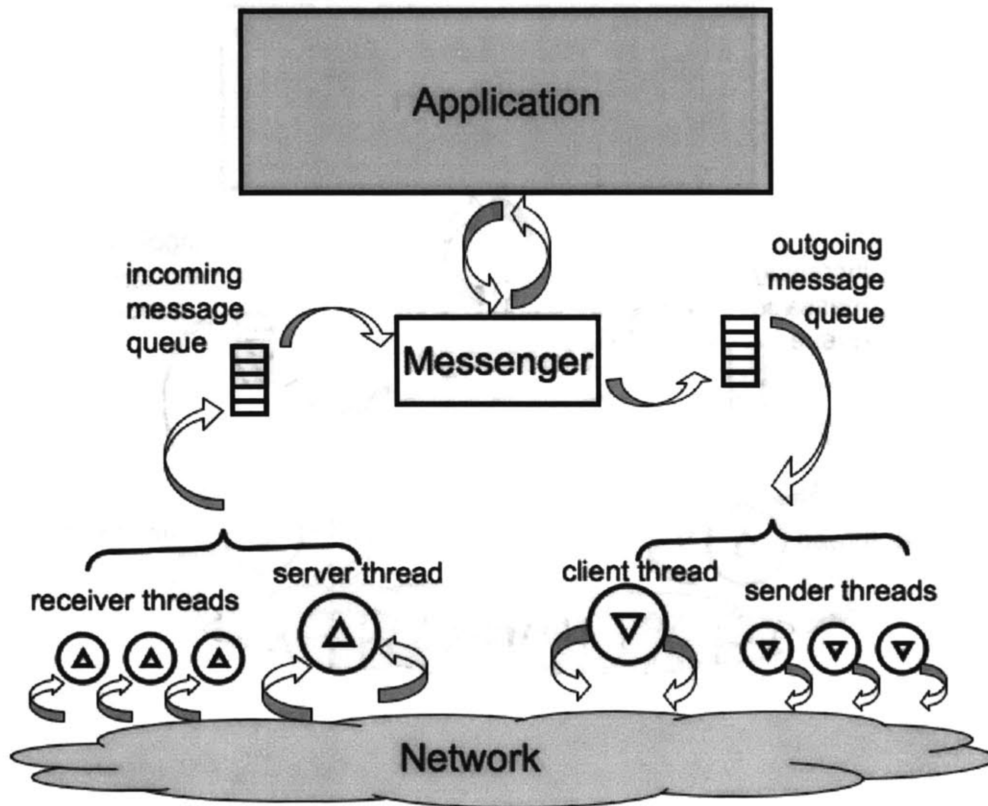


Figure 3-3: The Messenger has two MessageQueues, which serve as the interface between it and the networking threads that do the sending and receiving. Here, the sending and receiving are done both by the main client and server threads, and by auxiliary sender and receiver threads, which are dedicated to connections with a particular host. Arrows show the flow of data through the Messenger.

When it finds a message, it sends the message over the connection, and then looks for more messages. On the receiving end, when the main server thread encounters a request to set up a persistent connection, it passes this message on to the Messenger. The Messenger then intercepts this message and sets up a receiver thread that continuously receives messages from the given connection. The Messenger handles error cases that may arise when the remote end of a connection (incoming or outgoing) fails. These persistent connections allow the main client and server thread to stay open for intermittent traffic between hosts, while providing reliable service for pairs of hosts that exchange a lot of data.

The Connection object encapsulates the details of a TCP connection. It allows the Mes-

senger (or the application, if the application programmer chooses to work at that level) to send and receive messages symbolically by working in terms of Message objects, rather than in terms of byte arrays. If the use of threads is inappropriate for a given application, they may still use the Connection abstraction (in conjunction with the Message abstraction).

3.1.3 Keeping track of other hosts

The MachineNode class is a container for information about hosts on the network. In the applications we aimed to build, it was necessary for applications to know geometric information about other hosts. In addition to the address of remote hosts, the MachineNode contains a description of the remote host's coordinate frame, with respect to the local coordinate frame. The MachineNode has facilities to: determine if points are inside the extents of the host, calculate the distance from a point to the nearest extent of the host, and transform points into and out of the host's coordinate frame.

The MachineNodeMgr is a table of MachineNodes. In addition to fairly mundane tasks, the MachineNodeMgr provides some higher-level geometric intelligence. It is possible to search the MachineNodeMgr for nodes by address, or by their geometric relationship to the local host (e.g. by finding the host closest to a given point). Finally, the MachineNodeMgr provides thread-safe facilities for adding and removing nodes from the table.

The MachineNodeMgr is used extensively throughout our applications in an unheralded way, for simply storing information about remote hosts. Geometric information and intelligence is critically important in the distributed simulation and distributed input systems.

3.2 Data Backup/Retrieval and Process Management Services

In a distributed application, the failure of a process may not be merely annoying – it may adversely affect the entire system. It would be nice if programs could restart themselves

without human intervention. Sometimes, when programs fail, it is not necessarily the fault of the program itself. Programs that run correctly and without faults on one computer may not run properly on another. Programs that run properly on a freshly rebooted system may not run on the same system, once it has been running for a few hours or a few days. Hardware may be unreliable. Programs may be able to detect the failures, or they may just crash repeatedly. In the case of hardware or operating system corruption or failures, the only solution may be to reboot the computer. It would be nice if computers rebooted themselves when something went wrong. And it would be nice if everything that was running when the system needed to be rebooted, restarted automatically. To address these problems, we have designed a service that automatically restarts programs that have failed, and automatically reboots computers when critical failures are detected. We describe this system in detail in this section.

3.2.1 Process Management

The Process Manager is a piece of software that runs on each host. The Process Manager receives requests to start and stop programs from a client or host, over the network. If necessary, the Process Manager will make a request to the operating system to reboot the computer. There are three cases where the Process Manager will attempt to reboot the computer: if it receives an explicit request over the network, if a program signals that it has detected a critical error, and if a program has been restarted too many times. The rationale for this last case is that the only symptom of some styles of corruption is frequent restarts.

3.2.2 Primary application interface for the Process Manager

To start a program via the Process Manager, the client or application simply sends a message to the Process Manager, with the appropriate tags, using the name of the program to start as the payload of the message. The client or application does not need to take any further action.

Each program that is started via the Process Manager must set the maximum number of restarts (or, no maximum), and critical error code. The critical error code relates to the code or status that a program exits with. (In a C or C++ program, calling "exit(1234)" will cause the program to exit with code 1234. Alternatively, doing "return 1234" inside main() will also cause the program to exit with code 1234.) These parameters are set by sending messages to the Process Manager. (The exact nature of these messages is detailed in appendix F.)

Once started via the Process Manager, the program will be restarted the requested number of times (or indefinitely) if it terminates abnormally. If the process terminates normally, it will not be restarted. If the program terminates with the pre-defined error code, or if it needs to be restarted too many times, the Process Manager will automatically reboot the computer. When the computer reboots, and the Process Manager restarts, it will automatically restart the program, and any other programs that were being run when the Process Manager rebooted the computer. (Only programs being run through the Process Manager will be restarted – not every process that was running on the computer.)

3.2.3 How it works

When the Process Manager receives a command to start a program, it makes a record of the program being started, including its name, process id, the number of restarts allowed, and the pre-determined failure code. It then forks, and runs a wrapper process, which is responsible for handling the program when it terminates. The wrapper process forks again; one process becomes the specified program; the other waits for the program to terminate. When the program terminates, the operating system delivers the exit code to the waiting wrapper program. If the program terminates cleanly, the wrapper sends a message to the Process Manager, notifying it that the program has terminated. If the program does not terminate cleanly, the wrapper restarts the program. If the program has restarted too many times, or if it terminates with its critical error code, the wrapper sends a critical error message to the Process Manager. This process is summarized in figure 3-4.

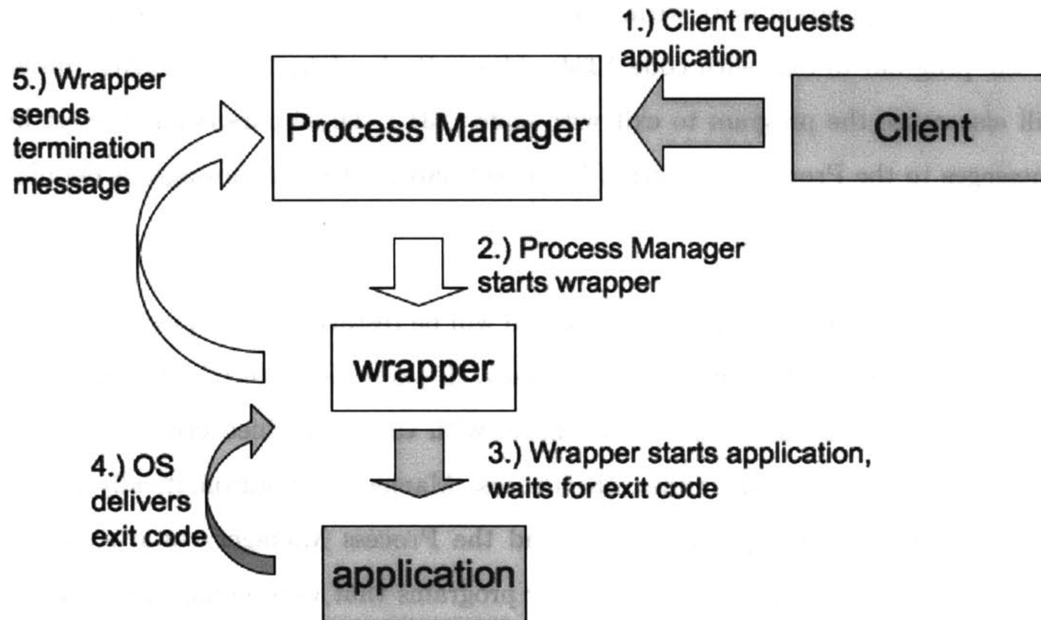


Figure 3-4: This figure shows the flow of control when the Process Manager receives a request to start an application. The Process Manager starts a wrapper program, which in turn starts the application. The wrapper program waits for the application's exit code, and sends a message to the Process Manager.

When the Process Manager receives a critical error message (or, if it receives a command from the network client) it makes a request to the operating system to reboot the computer. Before it does, it writes all of the unresolved program records to a file (as stated above, these records include the program name, the number of restarts allowed, and the critical error code). When the system comes back up, and the Process Manager restarts, it reads this file, and runs all of the programs with their original parameters.

3.2.4 Data Backup Services

We have described a service to automatically restart programs and reboot computers. But, what about the state of the programs that were running? Programs that crash might want to be able to recover their state when they restart. It seems silly to be able to restart programs after a computer has rebooted, if those programs cannot recover their state somehow. To facilitate this restoration of state, we have designed a data backup service.

The data backup services are rolled into the Process Manager. Applications send "Packages" of data to the Process Manager, to be backed up. Upon receiving this data, the Process Manager then propagates this data out to other hosts on the network. When the host reboots, and the Process Manager restarts, it goes out to other hosts on the network, gathers up left behind data, and makes this data available to the applications.

3.2.5 Primary application interface for the Data Backup services

The Package object encapsulates the information that the Process Manager needs in order to store data. Packages are uniquely identified by a combination of address/port/id, where address is the integer IP address of the host, port is the port where the application will be waiting to receive data, and id is a unique identifier that must be generated/stored by the application.

When the application wishes to store data with the Process Manager, it creates a Package, using the data to be stored as the payload. It tags the Package with its address, the port where it will receive data that is sent back to it, and a unique id number. It then sends this package to the Process Manager. When the application wishes to update information that it had previously sent to the Process Manager, it creates a Package, using the same tags it had previously used. It then sends this updated package to the Process Manager. The application does not need to take any further action to ensure that its data is safe.

When the application wishes to retrieve data that it left behind, the application creates a query with the tags of the Package it wishes to retrieve and sends this query to the Process

Manager. (It can also create a query to retrieve all Packages, instead of a single Package.) Then, the application needs to wait for the Process Manager to send all of the Packages. The exact nature of the messages that must be sent to work with the Data Backup services is detailed in appendix F.2.

3.2.6 How it works

When the Process Manager receives a new Package, it randomly selects two other hosts on the network, and sends them a copy of the Package. This process is shown in figure 3-5. When the application wishes to update information it had previously sent to the Process Manager, it sends a new Package, using the same address/port/id combination it had used previously. The Process Manager then sends this updated Package out to the other hosts where it had previously sent the Package. When the application wishes to remove the information it had previously sent, it sends a Package to the Process Manager with the same address/port/id combination (but no payload), asking the Process Manager to remove the Package. Then, the Process Manager sends a notification out to the other hosts where it had previously sent the Package asking them to remove the Package.

To be able to propagate data to other hosts on the network, the Process Manager must maintain a list of available hosts. The Process Manager begins this list by sweeping a range of IP addresses, and keeps it updated by monitoring its network traffic. When the Process Manager starts, the first thing it does is sends probes to other hosts on the network. If a probe is successful, the relevant host is added to the list of available backup hosts. When the Process Manager receives a probe, it knows that the host that sent the probe is now available, and it adds the sender of the probe to the list of available backup hosts.

Every so often, the Process Manager re-probes all of the hosts in its list, to make sure that each of the hosts is still available. If the Process Manager discovers that one of the hosts in its list is no longer available, it must find an alternate backup site for any data that it had sent that host, and, as a courtesy, it will propagate any data that the host had sent to it.

When a Process Manager receives a probe, it looks in its records to see if it had previously

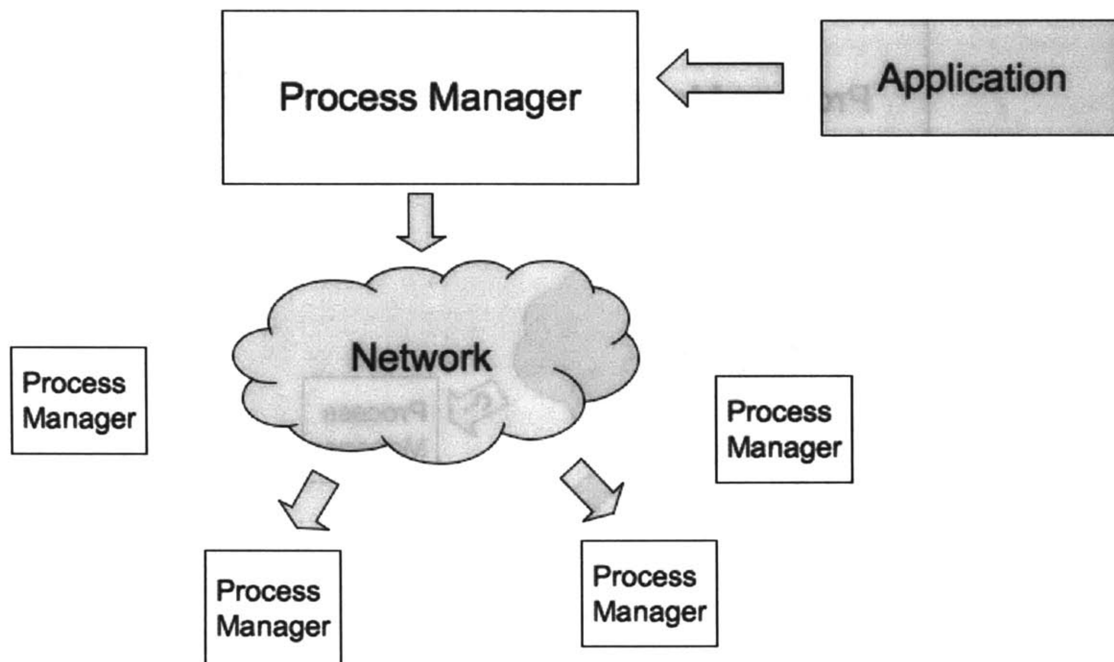


Figure 3-5: This figure shows the flow of data through the Process Manager when an application wishes to back up data. The Process Manager randomly selects two other hosts to receive copies of the data.

received any data from the remote host. If it had, it sends a copy back to them. Doing this allows a Process Manager that has restarted to recover the data it left behind, and make the data available to the application. This process is summarized in figure 3-6.

3.3 Data Aggregation Subsystem

Applications in ubiquitous computing and sensor networks are able to benefit by using a number of sensors to gather data. Using the data from multiple sensors, it is possible to derive high quality information that might not be available otherwise. A significant problem encountered in these types of applications is that the volume of data generated by

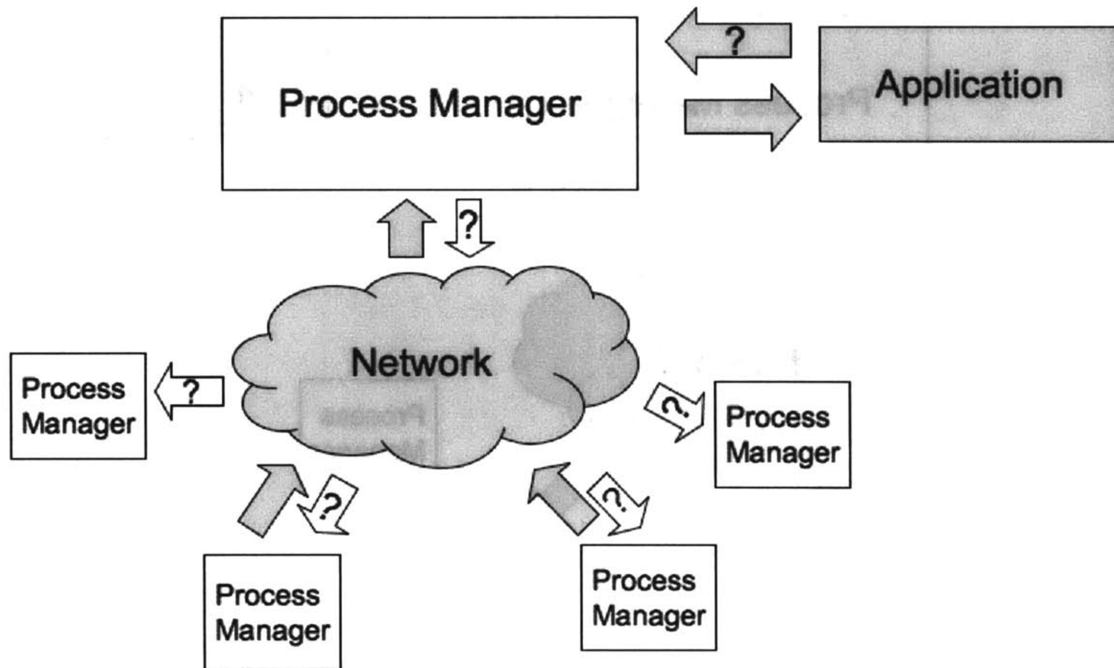


Figure 3-6: When the Process Manager (re)starts, it sends out a query to other hosts on the network. Nodes that receive the query, which have data sent by the host, send a copy to the Process Manager. The information is then available for the application to retrieve.

a collection of sensors can be quite large. For example, if some sensor is producing data at the modest rate of 10 pieces of data per second, and there are 6 such sensors in a system, in 10 seconds, the system generates 600 pieces of data. What we need is a way to organize this data, so that the application is not left with a tangle to unravel. We have created a system for organizing the data produced by multiple sensors, which we present in this section.

The Data Aggregation subsystem provides an object-oriented module for managing data from many sources. Data is organized first by the source from which it came. Then, data from different sources is grouped together to represent a phenomenon. The Aggregator manages the production, getting, sending, and association of data. We provide base classes for the primitives used by the Aggregator, which are extended to fit the application.

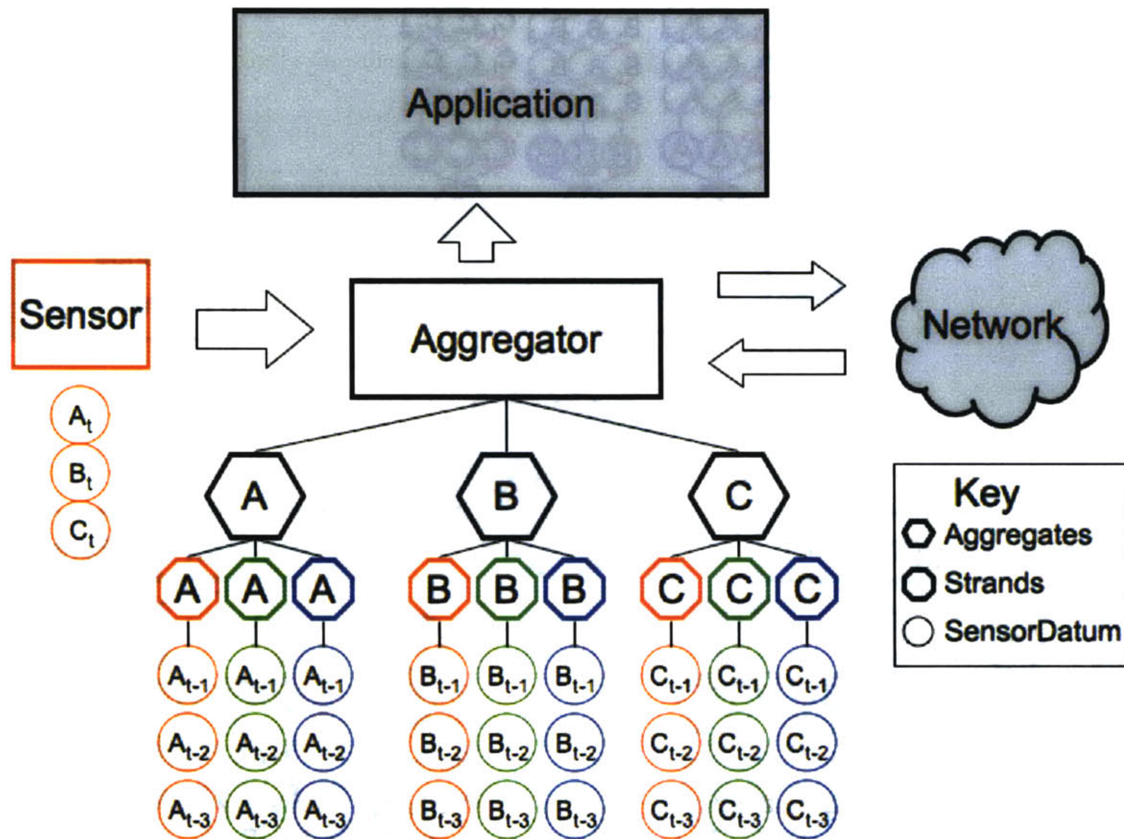


Figure 3-7: This figure shows the structure of the various objects inside the Aggregator. The Aggregator contains SensorDataAggregates (hexagons), which are comprised of SensorDataStrands (octagons). SensorDataStrands, in turn, are comprised of SensorDatum (circles). The different colors represent data from different hosts/sensors.

In the model used by the Data Aggregation subsystem, there is some Sensor, which is producing pieces of data (SensorDatum). These pieces of data are such that they can be strung together into streams (SensorDataStrand), reflecting some physical phenomenon over time. A stream of data represents a single physical phenomenon, as sensed by a single sensor. (If there is more than one phenomenon detected at each sensor, that sensor would generate multiple streams.) Streams of data from different sources, representing the same phenomenon, can be associated together, creating aggregate streams (SensorDataAggregate). SensorDatum are the only information exchanged by Aggregators running on different hosts. SensorDataStrands and, later, SensorDataAggregates, arise from collections of SensorDatum. Applications extend the Sensor, SensorDatum, SensorDataStrand,

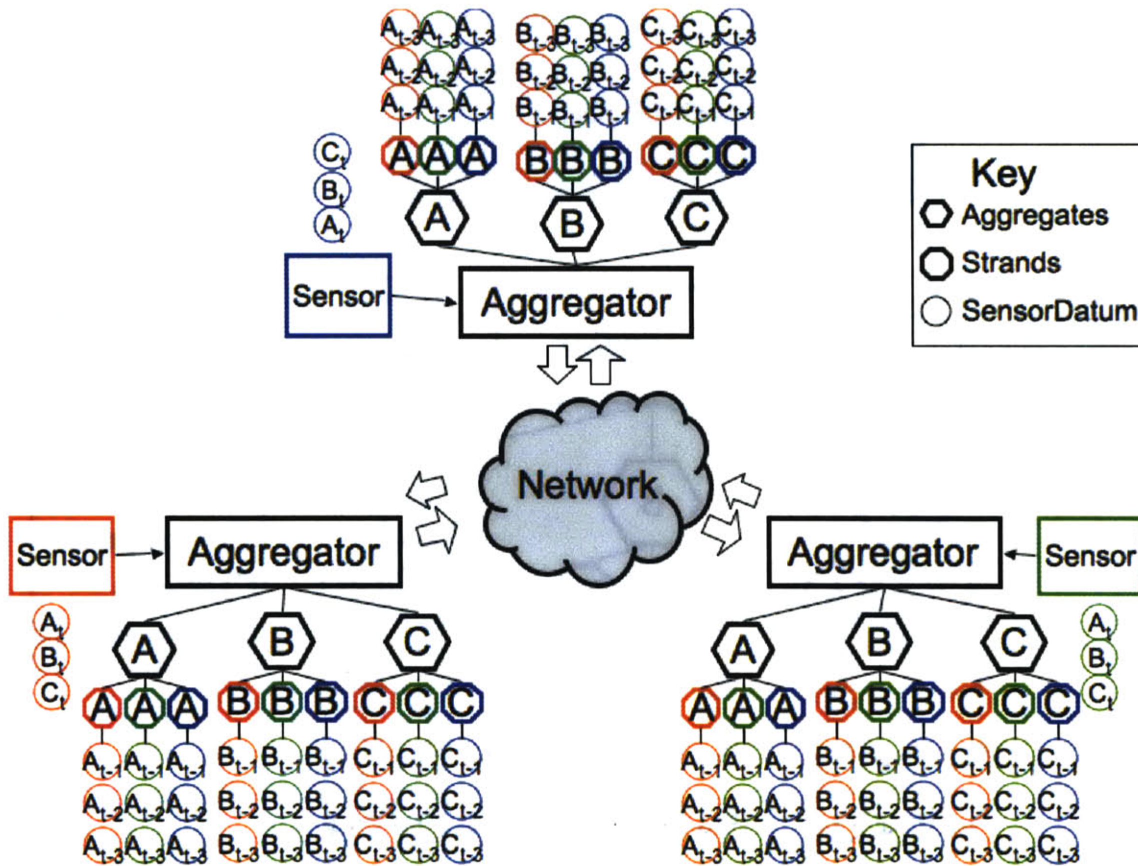


Figure 3-8: This figure shows how the Aggregator functions over the network. The colors represent data from different sources. Each of the three Aggregators has a sensor that is a different color, but by exchanging data, they are all able to aggregate data from all of the possible sources.

and SensorDataAggregate classes, to make the Aggregator work for their data types.

For example, a Sensor might be a face tracker. This face tracker produces locations and thumbnails of faces in a video stream. Furthermore, it knows that the face from one frame is the same face as the face in a subsequent frame, and so it tags the two consecutive pieces of information with the same identifier. (If it is tracking two faces at once, it would tag the data for each face with different identifiers.) Inside the Aggregator, these faces are exchanged over the network, and then, using their unique identifiers, they are strung together into streams of face thumbnails that represent one face, over time, as seen by one particular host. Using an application-defined similarity function, each host is able to

combine streams of faces, to collect different views of each face.

We have assumed a very general model, when developing the Data Aggregation system. The model used by the Data Aggregation system does not assume that data is being produced synchronously on different hosts (i.e. the different sensors are not necessarily waiting for each other). It does not assume that data is produced at the same rate on different hosts, and it doesn't assume that data is being produced at regular intervals on any particular host. It does not assume that the clocks of different hosts are synchronized, but it does assume that time moves forward. There may or may not be a delay in getting data from one host to another, and data may arrive out of order.

This general model makes the Data Aggregation system suitable for use in a wide range of applications, but limits its power to make sense of the data. The Data Aggregation system has been designed, primarily, to provide containers for pieces of data. It leaves the task of interpreting the data to the application and the application-provided derived classes. The system assumes that the Sensor has solved the data association problem for the data it produces. For purposes of determining if streams are similar and should belong to the same aggregate, the application must provide a similarity function through a derived class. Given that there is no assumption about any timing constraints, the system does not try to align the data in time (determine that two measurements happened at the same time).

3.3.1 Primary application interface

To use the Data Aggregation system, the application programmer must provide derived classes for the Sensor, SensorDatum, SensorDataStrand and SensorDataAggregate types. We describe an instance of the Data Aggregation system with derived classes in section 4.2.

To write a program that uses the data aggregation system with the derived types, the application creates an instance of a Sensor and passes the Sensor to the Aggregator. The derived Sensor class is responsible for creating objects of the correct derived types. The application registers callbacks for events that it wishes to respond to, and then makes

```
Aggregator * aggregator;
Sensor* sensor;
const in WORKING_PORT = 1234;
void callback(PointerList<SensorDataAggregate*>*aggregates, void* data){
  //Do work
}
int main(){
  sensor = new DerivedSensor;
  aggregator = new Aggregator(WORKING_PORT, sensor)
  //Aggregator setup (detailed in appendix G.)
  aggregator->RegisterCallback(AGGREGATOR_EVENTTYPE_TIMER, callback);

  while(true){
    aggregator->ProcessData();
    //do work
  }
}
```

Figure 3-9: The basic structure of a program using the Data Aggregation system. This program uses the callback registry to receive regular updates.

repeated calls to Aggregator::ProcessData(). The basic structure of a program that uses the Data Aggregation system is shown in figure 3.3.1.

The application can register callbacks to react to a number of different types of events. The application can register to receive updates at regular intervals (the intervals will not be shorter than the requested time, although they may be longer, depending on how long it takes the sensor to produce data). The application can also register to receive updates after at least some number of samples have been added (more than the requested number might have been added.) In both cases, the Aggregator will finish one iteration before calling the callbacks. The application can register to receive events when pieces of data arrive, when new strands or aggregates are created, or when strands or aggregates are removed. The application can also register to receive notifications when other hosts request local data.

The application must direct the Aggregator to request data from its peers. When the application directs the Aggregator to ask for data, the Aggregator sends a request to the relevant host. The receiver of this request must make a decision about whether it has resources available, and then must initiate a downstream persistent connection through the

Messenger.

To aid the application, or a derived class, with the task of aligning the data in time, the system does provide facilities to resample data, given an application-defined interpolation function. The default interpolation function simply uses nearest neighbors; it is not appropriate for data types that use higher order moments. The interpolation function is a member of the SensorDataStrand so that it may have access to all of the data available when doing the interpolation (for example, if the interpolation function involves high-order splines.)

3.3.2 How it works

In one iteration of the Aggregator, it goes through a cycle where it gets data from the sensor (synchronously), sends the data to other Aggregators running on other hosts (asynchronously), processes the data it produced, and then processes the data it received from other Aggregators. At the end of the iteration, it looks at all of the SensorDataAggregates to determine if any of them are stale. Stale SensorDataAggregates are removed from the system. Various events are generated throughout the cycle. These events are detailed in appendix G.

When the Aggregator receives/produces a SensorDatum, it first finds the SensorDataStrand where the SensorDatum belongs. The Aggregator then updates the SensorDataStrand and its parent SensorDataAggregate. If the SensorDatum does not match any existing SensorDataStrands, a new SensorDataStrand is created. Then, the Aggregator tries to find a SensorDataAggregate where the new SensorDataStrand belongs. If the SensorDataStrand does not belong to any existing SensorDataAggregate, a new SensorDataAggregate is created. Each SensorDataStrand will belong to exactly one SensorDataAggregate.

In order to find the SensorDataStrand where a given SensorDatum belongs, the Aggregator iterates through all the existing SensorDataStrands and asks them if the SensorDatum belongs to them. This allows derived classes to easily redefine the membership criteria. The base-class membership function simply matches the tags provided by the Sensor. When a

SensorDatum is added to a SensorDataStrand, the SensorDataStrand adds the SensorDatum in order (within a SensorDataStrand, SensorDatum are ordered by time), and then updates its modification time.

In order to find the SensorDataAggregate where a given SensorDataStrand belongs, the Aggregator iterates through all the existing SensorDataAggregate and asks them to generate a membership score. The Aggregator chooses the SensorDataAggregate with the best score. (This score is thought of as distance, and so the Aggregator chooses the SensorDataAggregate with the lowest membership score). The default membership function returns a constant number. When a SensorDataAggregate is updated, it updates its modification time, and then it iterates through all of its member strands, to determine if any of them are stale. Stale SensorDataStrands are removed.

The application can set the membership criteria, so that if none of the SensorDataAggregates generate good enough scores, the Aggregator will decide that the SensorDataStrand does not belong to any of them, and a new SensorDataAggregate will be created. Using the default membership function, depending on the application-define criteria, all SensorDataStrands will belong to different SensorDataAggregates, or they will all belong to the same SensorDataAggregate.

SensorDataAggregates must decide fairly early if a SensorDataStrand belongs to them (typically, after only one piece of data has arrived). If, subsequently, the SensorDataAggregate determines that the SensorDataStrand does not belong, it can remove it from its membership list, so that the SensorDataStrand will, temporarily, not belong to any SensorDataAggregate. The Aggregator will then try to find another SensorDataAggregate where the orphan SensorDataStrand belongs (or create a new one). On the other hand, if, initially, the SensorDataStrand is erroneously shunted to a separate SensorDataAggregate, the Aggregator will not interrogate these singleton aggregates/strands to see if they really belong as part of some other aggregate.

To determine whether or not a SensorDataAggregate (or SensorDataStrand) is stale, the SensorDataAggregate keeps a record of the time the most recent SensorDatum was added

to one of its strands. (When a SensorDatum is added, the SensorDataAggregate is updated with the local time, not with the timestamp on the SensorDatum.) SensorDataAggregates can determine that their member SensorDataStrands are stale, and remove them from their membership list. While the Aggregator is looking for stale SensorDataAggregates, it will also collect and remove stale, orphan SensorDataStrands. The application sets the definition of "stale" by specifying the maximum amount of time that an aggregate or strand can persist without being updated.

Chapter 4

Applications

All of the applications described in this chapter were written to run on the smart wall tiles of the Smart Architectural Surfaces project. Using the framework described in chapter 3, and the techniques described in chapter 2, we built a distributed, interactive simulation application. In the simulation, there are agents of various types. They move around inside the confines of the simulation, and interact with each other. Users can control the application using a colored ball as an input device.

We chose to write a distributed simulation because it was something that was fun, understandable, and relatively simple. The code for the simulator, which runs on a single computer is a few hundred lines. The simulator would also allow us to show the tiles sharing information, and handing off control of resources (agents). We could also showcase the decentralized nature of the system, since tiles in the simulation have no global clock and no global coordinate frame.

We chose to create a user input system using balls as input devices because the ball was a simple, understandable object. We opted for the ball as an input device, as opposed to a hand gesture based system, because the ball is something that a user could put down. The choice of input system was also influenced by practical concerns – the smart wall tiles are fairly computationallylightweight devices, and so we needed a modality that would not

require sophisticated processing. The use of brightly colored balls greatly simplified the requisite image processing.

In this chapter, we will describe the the distributed simulation, distributed ball tracking, and distributed input system that lead to our final, interactive application.

4.1 Distributed Simulation

In the simulation, there are a number of agents of different types, that move around and interact with each other. The simulation is fairly simple, when confined to one computer, but increases in complexity when moved to a distributed platform. There are three fundamental building blocks of the Simulator: Types, Agents, and Attractions.

The Type class provides a means for specifying the behavior of a set of agents. Characteristics of the behavior of an agent includes its maximum speed, the amount of randomness in its movements, and its maximum sight distance. The Type class also provides a mechanism where the application can specify the appearance of a set of agents, using polygonal segments. The appearance of a Type can have states, so that (for example) agents of a particular type could alternate being red and blue. Types are identified uniquely using a combination of the IP address of the host where they originated, and an id number, provided by the originating host.

Each individual agent in the system is represented by a state vector, which has fields for information about an agent's position and velocity, among other things. Agents are uniquely identified by a combination of their type (two numbers for the type), the IP address of the host where they originated, and an id number, which is provided by the Type residing on their host of origin (as opposed to a global pool of numbers being shared among all copies of the Type). Agents rely on Types and the Simulator to advance their state and decide where they should go next.

The Attraction class provides a simple way to describe relationships between Types. The Attraction class has fields for two Types : the active Type, and the stimulus Type. It also

has a parameter to describe how strong the attraction (or repulsion) is. All agents of a particular type will be attracted to or repulsed by all agents of another type.

At each step of the simulation, the Simulator goes through a cycle where it computes new velocities for all of the agents in the system and then integrates. To compute the new velocities, the simulator computes an aggregate attraction vector between each agent and all the other agents in the system. The attraction an agent feels for another agent depends the distance between the two agents and the strength of the relationship between the types of the two agents. Agents cannot be attracted to other agents that are beyond their maximum sight distance. The Type for the agent then computes the agent's next velocity by combining the attraction vector, the agent's previous velocity, and a random term.

Since we make no assumptions about synchronicity, handling errors and disagreements among tiles is a source of confounding complexity when moving this simple simulation from one computer to an array of computers. For the version of the simulator written for the our application, we created a system with no global clock and no global coordinate frame. Tiles were free to run at whatever pace suited them, exchanging messages when needed. All geometric information was kept in terms of the local coordinate system. Tiles know (or are told) about the relative transformation between them and all of their neighbors.

The first step towards a distributed simulation is to be able to transfer Types between tiles. Since the appearance portion of a Type can contain an arbitrary number of polygonal segments, divided into an arbitrary number of appearance states, the Type specification needs to be broken up into many messages. When sending a Type to another tile, the tile first sends the initial Type specification (the part that describes the behavior of the agents). When the other tile sends back an acknowledgment, the tile then sends the remaining geometric information. We decided to send the Types in this way for two reasons. The more practical reason is that, when tiles were simply spewing their Types to each other all at once, as in section 2.4, the network was getting congested and pieces were getting lost, despite TCP's best efforts. Breaking the Type into messages in this way provided a sort of flow control. The other reason is that, if a Type changes, there should be a way to

update the type. Sending the initial type specification first gives the receiving tile a chance to clear out the old, outdated geometry before new geometry arrives, and avoids the need to tag every piece of geometry in the type with a timestamp. It is possible that a computer running the simulation might encounter an agent with a type that the computer does not know about. Since agents are tagged with their type, the computer can simply request the relevant type from the host specified in the tags of the agent. This is a simple instance where a computer might know what it wants, and who to ask for it, without actually having the thing it wants, as discussed in section 2.7, where we discussed using queries to specific pieces of data.

The next step towards the distributed simulation is getting the Attractions between tiles. This mechanism is fairly simple, since Attractions are of a fixed length. The Simulator on one tile simply packages the Attraction and sends it to the other side. On the other side, the Simulator checks to see if it already has an Attraction for the given combination of active/stimulus types. If it does, it updates the parameter of that attraction with the newly arrived parameter. If it does not already have information about that relationship, it simply adds the newly arrived attraction to the system.

The final step towards the distributed simulation is getting information about the whereabouts of agents between tiles intelligently, and handing off control of the agents among tiles, correctly.

Tiles are able to ask each other for information they need by making spatial queries to each other, following the scheme using queries for streams of data, described in section 2.8. The query specifies that the asking tile should be notified of any activity that takes place within a requested distance of its borders. For such a query to work properly, the receiving tile must be aware of the transformation and extents of the asking tile. At the end of each iteration, the simulator compares each agent against its list of queries. If the agent is inside of the requested bounds, the tile sends a copy of the agent to the asking tile. If the agent was inside the bounds in the previous time step, but is no longer inside the bounds, the simulator sends a notification cancellation message, to let the other tile know that it can forget about that agent.

The process whereby tiles hand off the control of various agents follows the handshaking with timeouts scheme described in 2.6. While the simulator is running, agents move around inside the bounds of a space, where the bounds are defined by the application. If agents attempt to move outside one of these boundaries, and there is another tile next to that boundary, the Simulator sends a message to that tile, trying to pass the agent, and then waits for an acknowledgment. If the sending tile does receive the acknowledgement, it deletes the records for that agent, and continues. If it does not receive the acknowledgement after some amount of time, it will send a pass cancellation message. If an agent attempts to move past one of the simulation's boundaries, and there is no tile next to the boundary, the agent is "bounced" – placed back inside of the boundary, and the relevant portion of its velocity inverted. If, after a tile has attempted to pass an agent, it then needs to send a pass cancellation message, it will then bounce the agent, and continue as if it had never attempted to pass the agent.

There are a number of reasons that a tile might not receive a pass acknowledgement: the other tile might be busy and not have time to handle its messages, or the link between the sending tile and the receiving tile might be congested, or the link from the receiving tile to the sending tile (where the acknowledgement would need to travel) might be congested. In the first two cases, when the receiving tile finally gets its messages, the receiving tile will receive the agent, and then will immediately receive the cancellation, so that the agent will never have a chance to become established on the tile, and will not affect the behavior of other agents on the tile.

The last case, where the link from the receiving tile to the sending tile is congested, is somewhat more problematic. If the link from the receiver back to the sender is congested (but the reverse is not), the agent will have an opportunity to become established on the receiving tile. In the best case, the received agent will be the only agent on the tile, and it will remain on the tile until the cancellation message is received, at which time, the problem will be resolved. However, if the cancellation message follows some time after the tile has received the agent, the agent might escape to another tile before the cancellation message is received. This would mean that a duplicate agent could persist in the system for some

time. (Keeping records about every agent that has passed through each tile, for purposes for passing on cancellation messages, is both expensive and highly error prone.) What is worse, if this duplicate, imposter agent is in the system with other agents, the behavior of the other agents will be affected by its presence. In our case, this is not devastating. The resolution mechanism for handling duplicate agents is fairly simple. If a tile detects a collision (if it receives passes for two agents that have the same tags) it will simply remove one of them.

Initially, passing agents was done without timeouts. The tiles simply sent the pass messages to each other, and forgot about them. This worked sometimes, when the network was behaving well. At other times, if a link between two tiles were congested or otherwise faulty, many of the agents in the simulation would get "caught" in the space between the two tiles; the agents would seem to disappear when moving from one tile to another, and then some time later (possibly minutes), they would all emerge at once on the receiving tile. Sometimes, the agents would be lost outright, when moving from one tile to another along a congested link. The timeout scheme was adopted to be sure that agents were conserved in the system. We opted for a balance in the system where duplicate agents could arise, but where it was unlikely for agents to be lost.

4.2 Data Aggregation Instance : Distributed Ball Tracking

Now, we will discuss an instance of the Data Aggregation system, for distributed target tracking. Our task is for each tile to keep track of a number of brightly colored balls, even if they move out of the field of view of any particular camera. To use the Data Aggregation system for this purpose, we must provide derived classes for the Sensor, SensorDatum, SensorDataStrand, and SensorDataAggregate.

The TrackingMgr class (derived from the Sensor class) runs on a single tile, and has no awareness of other tiles in the system. It gets data from the camera and does the image processing to find brightly colored objects in the scene. It tracks the brightly colored objects from frame to frame using a nearest neighbors technique.

The TrackingMgr creates Ball objects (derived from SensorDatum). Ball objects include information about position, velocity, and color. The TrackingMgr tags each Ball with a unique identifier which corresponds to the object being tracked. At each frame, the TrackingMgr returns a list of Ball objects to the Aggregator.

Balls are strung together to create BallTracks (derived from SensorDataStrand). The BallTrack object uses the default SensorDataStrand association function (simply assuming that the TrackingMgr has solved the data association function). The BallTrack class keeps track of the color of the Ball added to it most recently. The update function updates the color of the BallTrack, using the color from the most recent Ball, and then otherwise uses the default function (updating the modification time of the BallTrack, and adding the Ball in order). The BallTrack also provides an interpolation function which interpolates the position of new Ball objects based on the old ones, and then recalculates the velocity of the new Ball objects accordingly.

BallTracks are combined to create Users (derived from SensorDataAggregate). The User class assumes that a single user in the world is represented by a collection of BallTracks. (Each BallTrack arises from a different camera.) The membership function of the User associates BallTracks together primarily by comparing their color. The membership function of the User also checks to make sure that BallTracks from the same host, which are being updated at approximately the same time, are not associated together. This allows the User class to give some consideration to the case where two different users happen to have balls with the same color, while allowing broken tracks from a given host (if the TrackingMgr loses, and then finds an object) to get stitched back together. (If BallTracks are being produced at the same time, on the same tile, they must correspond to different objects from the world. On the other hand, if BallTracks from the same host have matching colors, but one was produced before the other, it might be that the two BallTracks really represent the same object.) The User object also keeps a record of the color of the most recent Ball added to it, which is recorded by the update function.

To use the Aggregator to do distributed tracking, the application creates a new Aggregator, providing a Sensor (the TrackingMgr) which produces data (Ball objects). The application

needs to set two things: the maximum membership distance criteria, and the staleness criteria. In the case of distributed ball tracking, we want the application to receive regular updates, so the application must also set the timer, and then register a callback, which will be called at regular intervals. At this point, it is not necessary for the application to do anything special when strands or aggregates are created or destroyed.

We could write an application that showed how the views from different cameras are related (figure 4-1). In this application, the positions of the objects on the screen are set directly by looking at the position of the ball as seen in the different views. In this case, the application would tell the Aggregator to gather data from all the available nodes.

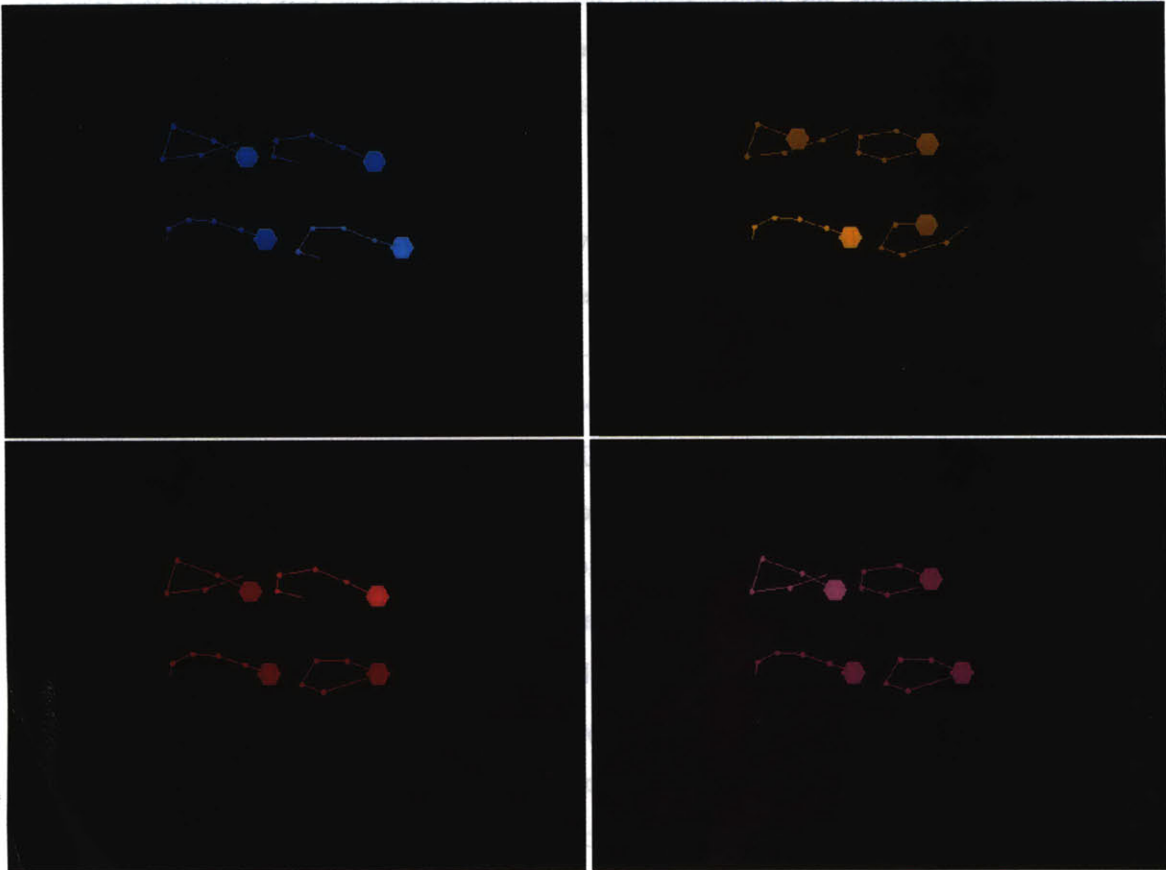


Figure 4-1: This figure shows the simultaneous state of four tiles participating in data aggregation, with the distributed ball tracking. The hexagon with the brightest color represents the local view of the ball. The darker colored hexagons represent remote views of the ball.

Alternatively, we can create a single object, and then combine the information from the

different views in some way. In the application shown in figure 4-2, the position of the object is initially set by looking at the position of the ball in one of the views. Then, the position is changed by integrating the observed velocity of the ball. The views of the ball are combined when, at each time step, the application chooses one of the views to use to update the position of the object. A more sophisticated method might use a stereo vision algorithm.

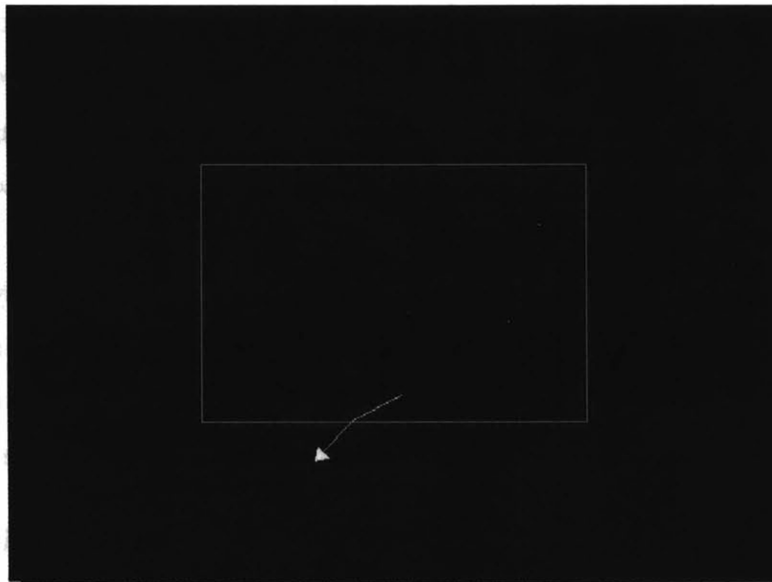


Figure 4-2: This figure shows the state of the cursor, where the position is determined by integrating the velocity of observed balls. The rectangle shows the field of view of the camera. The cursor is able to move outside of the field of view of the camera because the tiles are cooperating.

In the case where the application is updating a single object by selecting different views, the application does not need to receive continuous updates from all of the other nodes. Indeed, it is wasteful to do so, since we do not need data from all of the other hosts all of the time. The application shown in 4-2 keeps track of its immediate neighbors. When the ball moves near the edge of the field of view of the local camera, the application instructs the Aggregator to ask the node nearest the affected edge for help. When the ball moves away from the edge, the application instructs the Aggregator to tell the neighbor node to stop sending it data. In this way, the application is, both, able to conserve bandwidth, and able to track the ball, even when it moves out of the field of view of the local camera.

4.3 A Distributed User Input System, using Distributed Ball Tracking

The distributed tracking method proposed in the previous section would run on every tile – that is, each tile would maintain its own view of where the "cursor" would be, although each tile would be able to keep track of the cursor, even if it moved out of the field of view of its camera. Now, we will discuss the additional intelligence that must be added in order to have a single cursor among a set of tiles. To accomplish this, the CursorCoordinator receives, from the Aggregator, notifications when new aggregates are created, or when existing aggregates are destroyed, in addition to timed notifications (as with the distributed ball tracking). (The CursorCoordinator operates on a port separate from the Aggregator, so that they do not receive each other's messages.) We assume that the clocks of the tiles are not synchronized, and they are not producing data at the same rate. It would be possible, using a semaphore style synchronization scheme, as discussed in section 2.10 and 2.11, to force the tiles to move in lockstep, but that would preclude the usefulness of the system for real-time interaction.

Information about the derived position of the cursor is contained in CursorPosition objects. CursorPosition objects are used to record the history of a cursor's movement. This history, along with other information, is contained in Cursor objects. The most important of these other pieces of information is the control state. This field indicates whether, and to what extent, the CursorCoordinator has control of, or has a vested interest in the given Cursor. The possible values of this field include: true, false, pending, and a few other values that we will discuss in detail, later. The Cursor also has an separate state field, which is for use by the application.

The first hurdle to overcome is bringing all of the tiles to agreement about where the cursor should initially be, and who should be in charge of updating it. Given that we have no assumptions about any sort of synchronicity, we cannot actually guarantee that the tiles will agree, but we will do our best. Given that there is a user holding a ball, heuristically, the tile that has the best view of the ball should control the cursor. So, the task, then, is for the tiles to decide who has the best view, where the "best view" is defined as the smallest

distance between the apparent position of the ball, and the center of the field of view of the camera. The method to bring the tiles to agreement follows the semaphore-style scheme described in section 2.10.

When the tile is notified that a new aggregate has been created, it finds the local view of the ball, and calculates the distance between the local view of the ball and the center of the field of view of the camera. It also creates a new Cursor, corresponding to the new aggregate, and marks its control state as pending. Using the identifiers for the new cursor, the distance between the local view of the ball and the center of the field of view, and the appearance information from the local view of the ball, the tile creates a query, which is sent out to all of the other tiles in the network. As the tile sends out these queries, the tile makes a record of the distance it sent out, and each tile where it sent the query. This record serves as the semaphore. Two cursors under negotiation at the same time would get separate records.

The tile then waits to receive responses from the other tiles. The response contains the identifiers for the cursor (the same tags that were sent out), and the distance between the remote view of the cursor and the remote center of the field of view. If that distance is less than the distance the tile sent out, the control state of the cursor is marked false, and the semaphore is cleared. (The tile does not need to know that three other tiles have better views – one is enough.) If that distance is greater than the distance the tile sent out, the control state of the cursor is not modified, but the remote host is removed from the semaphore. At each iteration, the CursorCoordinator checks the semaphores of all pending cursors. If the semaphore is clear, the cursor is then activated, and its control state is set to true.

As the CursorCoordinator is running, if it receives a control negotiation query, it uses the appearance information in the query to identify what local cursor might correspond to the cursor in the query. If the control state of the matching cursor is pending, the tile will respond with the distance that it sent out, when it made its request (of which it made a record). If the control state of the matching cursor is true (the tile had previously negotiated for control of the cursor), it sends back a distance of zero. If the control state of the matching

cursor is anything else, or if the tile cannot find a matching cursor, it sends back a very large distance (essentially saying that it has no input). When formulating a response, it uses the tags given in the query, and the local distance information.

A real problem with this scheme is that there is absolutely no guarantee that the tiles all saw the ball at the same time. This is ameliorated somewhat by having tiles respond to each other with the distance that they sent out when attempting to negotiate for control (rather than the distance at the time they receive the query). Another problem is that this resolution process takes a long time (on the order of seconds), and the user can move the ball a lot in that time. So the tile that negotiates control of the ball might not have the best view by the time the negotiation is complete.

Initially, the tiles used a time-out scheme, as described in 2.6, for negotiating control. Tiles sent out the queries, containing their tags, distance, and appearance information, as above, and then waited for answers. If they did not receive an answer, saying that someone else had a smaller distance than their own, after a certain amount of time, they activated the cursor. When a tile received a control negotiation query, it only responded if it already had control of the cursor, or if its distance was smaller than the advertised distance. This scheme had the benefit that it used less bandwidth, and tolerated node failures, but it failed to account for varying latency. When the number of hosts negotiating for control increased, the timeout value also needed to be increased. This meant that, if there were less than the target number or hosts negotiating for control the user had to wait, even if all relevant data had been exchanged. More importantly, if all of the data did not get back and forth in time, multiple tiles would conclude that they had control of the cursor. It was due to this problem, that the semaphore style scheme was adopted.

Once initial control has been negotiated, the challenge is to keep the control of the cursor on a tile with a good view of the ball. (Requiring that it be the best view of the ball is unnecessary, and very difficult, if not impossible, without any assumptions about synchronicity.) Handling the control of the cursor, once initial control has been negotiated, follows the handshaking with timeouts scheme described in section 2.6.

To determine if a cursor needs to be passed, the tile checks to see if the local view of the

ball has moved too close to some edge, where "too close" is defined as within some tolerance of the edge. If the ball has moved too close to the edge, it sends a message to the neighbor closest to the affected edge, asking to pass the cursor, and then marks the control state of the cursor pending. Then, it waits to receive an answer. If the other tile accepts the pass, then it changes the control state of the cursor to false. If the other tile rejects the pass, the tile marks the control state of the cursor true, and then carries on. A pass will be rejected if the other tile is unable to find a cursor in its repertoire that matches the description of the cursor it has been passed.

If the tile does not receive an answer soon enough, it will take back control of the cursor by marking its control state true. If the other tile subsequently accepts, the first tile will send the other tile a message, saying that they should forget about the information that it had previously received. This is necessary, instead of simply letting them have control of the cursor, since there is no bound on latency; their response might be very stale. When the other tile receives the cancellation message, it gives up control of the cursor it had previously been sent. A cursor will only be sent, as a pass, to another cursor once. After that, it is then tagged, so that the tile will not try to send it again before the other tile answers or times out.

Since the fields of view may overlap significantly, there may need to be a mapping such that cursors in a significant portion of a given tile's field of view may need to be drawn on another tile. To accommodate this, the CursorCoordinator has facilities to send and receive notifications about activity related to cursors, so that cursors can be drawn on tiles that are not neccesarily controlling them. Information about different cursors is tagged with unique identifiers, so that a tile can receive information about multiple cursors at the same time.

Once a tile has received information about some cursor, an important question is how long the tile should keep this information. In this system, the answer we have adopted, is that a tile should keep such information until someone else tells them to remove it. This leads to a number of interesting problems. For example, if the mapping from the field of view to the screen is such that movement on one tile might end up drawing the cursor on *two* other tiles, what should happen? What if the application wants to draw one big, long tail,

showing everywhere that the cursor has been? Clearly, if the associated ball disappears, both of the tiles should be notified of the cursor's demise. Also, it is clear that we should not remove the cursor from the first tile when we start sending to the second tile, but it also does not make sense to continue to send the first tile continuous updates. Also, if control of the cursor changes hands while a tile is receiving updates, tiles receiving updates should continue to receive these updates, no matter who has control of the cursor.

To handle problems such as these, we have adopted the idea of a "receiver" and a "stakeholder". A receiver is a tile that is receiving continuous updates about the position and state of a cursor. A stakeholder is a tile that is receiving intermittent updates about a cursor, such as changes in the application-defined state, or the cursor's destruction. These notions are reflected as possible control states for each cursor. (A tile will know that it is a stakeholder or a receiver for a given cursor.)

When the CursorCoordinator determines that it should send data to another tile, it registers that tile as a receiver of information about the affected cursor. If the CursorCoordinator then determines that the tile should no longer receive continuous updates, it changes the status of that tile from a receiver to a stakeholder (it modifies its own records, and sends messages to the tile, to notify it of its change in status). When control of a cursor passes from one tile to another, the sending tile gives the receiving tile the list of receivers and stakeholders for the given cursor, including itself as a stakeholder.

After introducing the notion of receivers and stakeholders, we must make a modification to what happens when one tile tries to pass the cursor to another. When the passing tile receives a message that the remote tile has accepted the pass, it does not mark the control state of the cursor as false, as we said before. It marks that it is a stakeholder of the cursor.

A stakeholder for a cursor doesn't really care where the cursor information comes from, but it does care that the information has the right tags. So, when passing control of the cursor between tiles, maintaining the integrity of these tags is important. Tiles cannot simply adopt their own, local tags, and expect them to work. They must use the tags that everyone is expecting.

When a tile receives a pass from another tile, the pass includes information about the tags from the previous controller of the cursor, as well as a description of the ball associated that cursor. Using the description, the tile finds a cursor in its own repertoire that matches. It also checks to see if there is any cursor that matches the tags of the newly arrived cursor. If there is a single cursor that matches both, the description of the ball, and the tags, then everything is all set. If there is no cursor that matches the tags of the new cursor, but there is a cursor that matches its description, then the cursor matching the description is given the appropriate tags. If there is a cursor that matches the description, and another cursor that matches the tags, the cursor with the matching tags is set to use the ball associated with the cursor that matched the description (and the cursor initially matching the description is deleted). This situation arises if the tile had previously received information about a cursor to draw, which would have the same tags, although it would not be associated with any ball. When the tile is passed the cursor, with its description, it is finally able to make the association between the ball it sees and the information for the cursor it was receiving remotely.

Since there is a lot of information flying around about who should be receiving what information about what cursor, there is an awful lot of room for inconsistencies to creep into the system. When inconsistencies creep into the system, generally, cursors on tiles that were receiving remote updates are not removed properly. To trap all of these errors, we have adopted keep alive messages, as described in section 2.9. A tile that has control of a cursor must periodically send keep alive messages to all of the stakeholders for the cursor. If a tile is a stakeholder for any cursor, for which it has not received the requisite keep alive message, that cursor is deleted. When receivers get information about the position of the cursor, that is an implicit keep alive message.

Finally, when the CursorCoordinator receives a notification from the Aggregator that a ball has disappeared (a User object is being removed), if the tile has control of the cursor, it sends a termination notification to all of the cursor's receivers and stakeholders before removing the cursor. After a stakeholder or a receiver is instructed to remove a cursor, if it still sees the ball that was associated with that cursor, it will create a new cursor, and

begin the initial control negotiation process.

The last concern is that users must have some way to indicate intent (like a mouse click). In our system, we have designed a small vocabulary of gestures that users may utilize to communicate with the system. The gestures are : a horizontal shake, a vertical shake, and a circle. The path of the ball is tokenized into strokes by looking for stops (when the ball moves a very small amount for some number of frames). Each token (stroke) may be classified as one of the gestures, or as a non-gesture. Strokes are trivially classified as non-gestures if they are either too long or too short. Otherwise, a stroke is classified by making a histogram of the angles of the velocities inside the stroke. This histogram is then compared to a set of templates using mean-squared error (MSE). If the MSE is below a certain threshold, the stroke is categorized according to the template that best matched its histogram.

Recognition of gestures is done by the tile controlling the cursor. The tile delivers notifications of the gesture events to all receivers and stakeholders. The application can register to receive notifications when gestures happen. To allow system-wide events to be propagated at the application layer, rather than at the cursor layer, events that happen locally and remotely are registered separately. If this were not the case, consider the case where some gesture toggles the state of some variable in the application that is shared among the tiles. The application would register to receive notifications about the relevant gesture. Say now, that the gesture happened. If the tile were a receiver of cursor data, it would receive a gesture event, and the application variable would toggle. But, the tile controlling the cursor would also generate a gesture event, toggling the variable, and so the variable would be changed twice.

4.4 An Interactive Application

In the interactive application we have designed, users are able to control aspects of the distributed simulation, using a ball, via the distributed input system described in 4.3. Users are able to add agents of various types to the system by using gestures. Using a GUI style

collection of buttons and sliders, users are able to set up interactions, where agents of one type will be attracted to or repulsed by agents of another type. Users that are editing the same relationship are able to see each other's cursors, so that they will have some sense of the presence of the other side.

To set up an interaction, users must have a way to modify all of the critical parameters of the Attraction object: the active type, the stimulus type, and the strength of the attraction/repulsion. To do this, users access a special purpose screen, which contains two buttons and a slider. The slider both, reflects the current state of any interaction between the two selected types, and serves as a way to modify the state of the interaction between the two types. When an interaction is edited on any one tile, it becomes effective throughout the whole system.

A button is essentially a special, active area of the screen. To activate a button, a user moves the cursor into the active area, and then makes a gesture. The gesture does not need to end in the active area, but it does need to start there. When a gesture occurs, the application does intersection tests with all of the relevant active areas (e.g. buttons). If the gesture happened inside one of the buttons, that button is activated. This is analogous to "clicking" on a button in a traditional GUI. Doing different gestures inside the button is analogous to "right-clicking" or "left-clicking".

A slider is similar to a button, in that it is an special area of the screen. However, it is different, since the user must be able to make some sort of continuous adjustment to the slider. In traditional GUIs, this is usually done with the "click and hold" paradigm. Since no such paradigm is available when using the ball as an input device, we have devised another way. When a user activates the slider, the cursor used to activate the slider becomes associated with the slider. Subsequent movements of the cursor adjust the value of the slider. When the user releases the slider (with another gesture), the cursor is disassociated, and further modifications are not made.

In our user input system, there can be multiple cursors, and cursors can disappear (complications not faced in traditional GUIs). So, when a slider has been activated, and is

associated with some cursor, it must prevent other cursors from activating it. Also, if a cursor that is associated with a slider disappears, the association between the cursor and the slider needs to be reset so that some other cursor can use it. There are also some interface design issues. For example, if there is a vertically oriented slider, it is suboptimal to use a vertical shake as the gesture to release the slider.

As users manipulate the slider, the relationship between the types is updated locally, and then propagated out to all other nodes running the simulation through the Simulator. A relationship edited on one tile is reflected in the whole system. So, if two (or more) users are looking at the slider for the same pair of agent types, they are essentially looking at the "same" slider – they each have a slider in front of them that controls the same parameter in the system. This leads to two interesting problems: 1.) if one of the users manipulates the slider, the changes should be reflected in the sliders that all the other users see, 2.) only one user should be able to manipulate the slider at once.

To handle the first problem, the application is notified whenever an updated relationship has been received. If the tile is editing that relationship, the slider is updated to reflect the new parameters. In this way, all of the sliders are updated by proxy. A more direct way to do update the sliders might be to use "sessions", as described below.

To handle the second problem, we have adopted the notion of a "session". A session applies to some control, or set of controls, and allows the nodes running the application to exchange information about what is happening to those controls, so that it can create a cohesive response. If a tile is in a session, and its state changes, it sends out updated session information to all of the other tiles in the system. If a tile is in a session, and it encounters a state change in another tile's session (that matches its own), it takes some appropriate action.

For example, the session for the slider lists the relationship that the slider is editing, and whether or not the slider has been activated. When a user selects a pair of agent types, the application starts a session for editing that relationship. This session is sent out to all other tiles in the network. Subsequently, if a user at another tile wants to edit the same

relationship, that tile also sends out a session for that relationship. It is *not* important who starts the session for a given relationship, or if two tiles try to start a session at the same time. All that is important is that the tiles are exchanging information about the state of controls that are being manipulated by users at each tile. When one of the users activates the slider, that tile sends out updated session information, saying that its user has taken control of the slider. In response, the other tile will not allow its user to take control of the slider. If the tiles detect a collision, where both users have tried to take control of the slider at the same time, they will both relinquish control, and try again. If a user then chooses to edit a different relationship, the application ends the session for the first relationship (by sending out a notification that it is ending its session) before starting a session for the next relationship.

Chapter 5

Evaluation

To evaluate the framework's ability to support distributed applications, we examined the usability of the final, interactive application described in Chapter 4. Three rounds of user testing were conducted. Modifications were made to the system to respond to users' needs.

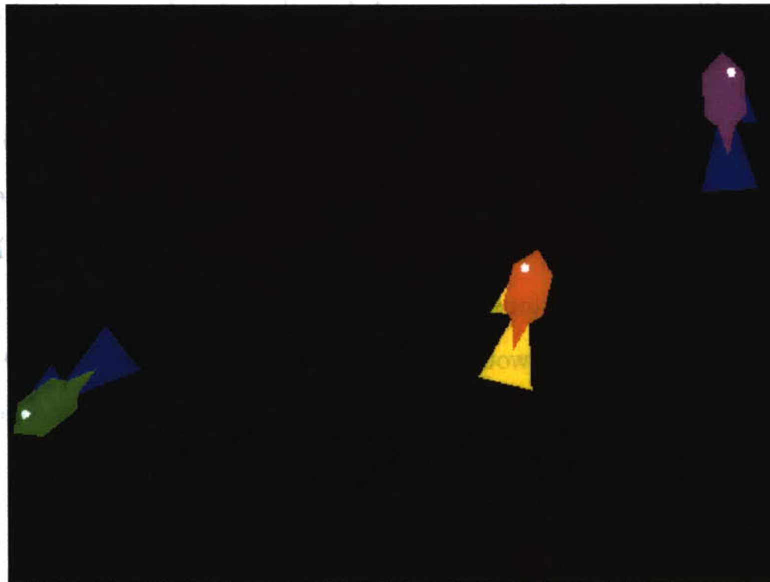


Figure 5-1: This figure shows the agents moving around in the simulation.

In the application developed for the user testing, agents could be one of three types of fish : orange, green, and purple (shown in figure 5-1). In all of the trials, there were two

installations of the wall tiles, containing four tiles each. The installations were in different parts of a conference room, facing away from each other. Users could not easily see each other while standing in front of their respective installation of tiles. All eight tiles were involved in running the unified simulation. Agents being simulated in one set of tiles were reflected in the other set of tiles. Agents being simulated in different sets of tiles could still be attracted to each other. In contrast, there were two separate ball tracking networks set up, one for each installation. We set the test up this way, so that we could see that the distributed simulation was working properly, that the distributed input system was working properly, and that users were able to control the system and collaborate.

In all of the tests, the goal was for the users to create multiple fish (at least one of each type), and set up two relationships. Before attempting to complete the task, users were initially shown how to create fish and how to set up relationships. Users were able to ask questions about the user interface while they were working with the system. After either completing the task, or a sufficient period of time had elapsed, users were asked to fill out short questionnaires. Users were asked to describe what they did, describe the system, describe what it was like to use the system, and describe what happened while they were using the system.

In the first study, there were three modes in the application : a simulation mode, a selection mode, and an interaction mode, shown in figure 5-2. In the simulation mode, the user viewed the fish as they swam around, and was able to add fish of various types. The other two modes were used to create attractions between different types of fish. In the selection mode, users chose what type of fish would like what other type of fish. The would make a gesture on a button to move into the "interaction" mode, where users controlled a slider to determine the strength of the attraction/repulsion.

Ten users, in five pairs, participated in the first study. Users were instructed to create ten fish, and set up two relationships. Only two of the five pairs were able to complete the task. All groups were able to create the requisite number of fish, but users struggled to create relationships.

Users had difficulty with the precision required to make gestures on all of the various

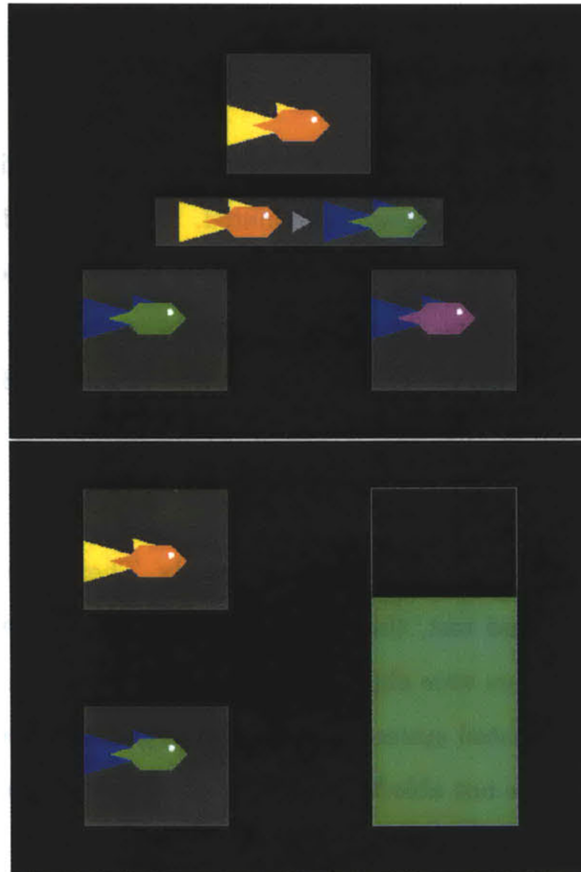


Figure 5-2: This figure shows the two modes for editing interactions, in the first version of the application. The selection mode is shown on top. The interaction mode is shown on the bottom.

buttons. Users also had trouble when gestures were recognized erroneously, taking them out of their desired mode, and requiring them to navigate through the modes, back to the screen where the slider could be manipulated.

A surprising result was that users did not collaborate very much. It was expected that users would try to divide the task amongst themselves, by having one person create one relationship, and the other person create the other relationship, or by having one person create the fish, and the other person create the relationships. This behavior was not observed. Users remained focused on their installation of tiles, and did not talk to each other very much. Part of this could be attributed to the description of the task, but the main cause of this is that users were so focused on using the input system, that they did not have attention

available to talk to each other about what relationships they should create, or how they might otherwise divide the task.

To address the problems users had activating the buttons, the application was modified. In the second study, the "selection" mode was eliminated. Users selected the agent types for the interaction in the "interaction" mode, on the same screen with the slider, by iterating through the different available types. To address the problems had with erroneous gestures being identified, the gestures used to activate buttons and change modes were adjusted, to make it less likely that gestures that would have a detrimental effect would be recognized erroneously.

Eight users, in four pairs, participated in the second study. They were instructed to create a "love triangle." In the second test, three of the four groups were able to complete the task. In the remaining test, users were able to create the number of fish, and set up multiple relationships, but, due to repeated system failures, the application needed to be restarted three times, and the users were not able to create the all of the relationships and all of the fish in a single session.

In the second user, study, there was more collaboration. All but one of the pairs talked to each other about what they should do next. They negotiated about what types of fish they should each make like or dislike each other. This change in the nature of the user's behavior can largely be attributed to the change in the description of the task.

Users continued to struggle with the necessary precision to activate the buttons. There were fewer buttons, and so this posed less of a problem. Erroneous gestures remained a problem, but having only two modes reduced the cost of having a gesture recognized improperly.

In both the first and second tests, many users characterized the system as "frustrating" or "difficult" (even users who were able to complete the task). One of the sources of frustration is that the cursor can take some time to appear. When a user first shows the ball to the system, it can take the tiles some time to resolve, amongst themselves, who should be in charge of the cursor. While waiting, people tend to wave the ball around, trying to get the tiles' attention. This exacerbates the situation, since the tiles then have trouble agreeing

because they are all seeing the ball in different places. Another source of complaints was that the cursor would sometimes disappear. This is actually by design – when the tiles detect a collision of control (where two tiles think they should have the cursor) both tiles let go, and then they all renegotiate control.

Most of the frustration was derived from the difficulty of controlling the cursor in a predictable way. At first, the processing power of the computers was blamed, for not being able to track the balls and produce data at a sufficient rate to keep the tracking smooth. However, this complaint could not be reconciled with the observation that the cursor worked fairly well when only one tile was involved. Next, network capacity and latency were blamed for not getting data back and forth fast enough. However, this could not be reconciled with the observation that the pure data aggregation application (discussed in Chapter 4, where the positions of the ball on all of the tiles were shown on the screen) seemed to run in real-time, without very much appreciable lag.

Finally, we concluded that it was the "jerkiness" of the cursor (that it would go slowly, and then suddenly speed up, and then suddenly slow down again) that gave users difficulty. This jerkiness was a result of the way the CursorCoordinator instructed the Aggregator to gather information. The CursorCoordinator would only ask for help from other tiles when a ball was within some tolerance on the edge of the field of view of the camera. When the Aggregator was getting information from another tile, this would slow the system down. When the ball moved away from the edge of the field of view (or passed to another tile), the system would speed up again. Due to the way the position of the ball was integrated, a ball being controlled on one tile might be drawn on a different tile; there was not a strong relationship between the position of the ball in the field of view of the camera, and the cursor on the screen. So, it made no visual sense when the cursor would slow down and speed up.

To address this problem, the system was modified again. This time, The CursorCoordinator received data from all of its neighbors, all of the time. This had the effect of slowing the system down, but it made the control of the cursor much smoother and more predictable.

After cursory tests with previous subjects, showing that this was indeed a significant im-

provement, we did a third round of user testing. In the last test, the goal was really to test the usability of the input system, and less the ability of the users to collaborate. So, users participated alone. Six users participated in the last study. Users were instructed to create a "love triangle." The goal was the same, as with the pairs of users; users were to create at least one fish of each type, and set up two relationships. Five of the six users were able to complete the task successfully.

As predicted, users struggled a lot less with control of the cursor. Only one of the six users (the user who had been unable to complete the task) characterized the system as "frustrating." However, problems remained.

The user who had been unable to complete the task had a darker complexion, and orange-ish tones in his skin. The image processing on the tiles intermittently picked up his face, as if it were a ball. This led to cursors jumping all over the place, which rendered the system mostly unusable. (A similar problem was observed in the second user test, where one of the users in the pair unable to complete the task had similar skin tones.) Even users with lighter complexions had their faces identified as balls occasionally. Early in the development of the system, a balance had to be struck between identifying clothes and skin as balls, and being able to identify the balls as balls. The solution to this problem, really, is more sophisticated image processing. In its present form, the image processing simply looks for areas of saturated color that are of at least a minimum size. A face detector would help to screen out people's faces, and stronger appearance priors (about the ways shadows are cast on the balls) would help to screen out clothing. To do better image processing, the system needs computers with significantly more processing power.

Another problem users had was with duplicate cursors arising. This problem comes from errors in the Aggregator, when strands are not associated together into aggregates properly. (This is shown in figure 5-3.) The cause of strands not being associated together properly is the weakness of the association function in the derived class of the SensorDataAggregate. Color is used as a discriminator to decide if strands belong together. However, all of the cameras are a little bit different, and motion blur can affect the average color of areas detected as balls. The solution to this problem is to use more and stronger appearance

Figure 5-3: Faulty data aggregation, due to inadequacies in the data association function, leads to multiple aggregates created for a single object, which leads to multiple cursors for the same ball. SensorDataStrands in the same SensorDataAggregate are represented by hexagons of the same hue.

information when deciding if strands belong together. This would require more sophisticated image processing, and, potentially, more network bandwidth (if the appearance information took up a sufficiently large number of bytes).

Aside from the difficulties with the user input system, the framework was fairly successful at supporting the distributed simulation. In the user surveys, there were no mentions of fish disappearing, for example. Users also understood very quickly that interactions that they edited on one tile were propagated to the other tiles (and to the other installation of tiles, where applicable).

The way users described the system speaks most to the success of the framework in supporting a distributed, interactive application. Users, as a group, did not get stuck on the fact that the application was being run on four (or eight) separate computers. Many of them (though, admittedly, not all) viewed the system as a cohesive unit. One user observed that "All objects on the total screen moved ... between screen partitions." This remark is particularly encouraging, since it shows that the user thought of the system as one large

thing, and viewed the different computers as mere partitions of the larger thing. One of the users from the second study described the system as "two sets of four tablet computers with cameras," but then went on to say "*It* is controlled." So, on the one hand, the user was aware of the hardware elements of the system, but also viewed it as a cohesive system.

Chapter 6

Conclusion and Future Work

In this thesis, we have presented a software framework to support programmers writing distributed applications. We have assumed a very general model of mesh network style distributed computing, where hosts have no shared clock, no shared memory, and latency is unbounded. We have not assumed the most general case of distributed computing, since we have assumed that hosts can be identified uniquely, but we do not feel that this limits the usefulness of our work.

When programming for a distributed environment, seemingly simple tasks become complex quickly, in the face of unbounded latency, network failures, and node failures. It is possible to manage this complexity by anticipating possible error conditions, and by uniquely identifying pieces of data, using combinations of locally available identifiers.

We have provided support for the exchange of data in distributed applications by helping applications to send and receive messages asynchronously, providing a mechanism that allows applications to handle the exchange of data in an event driven manner, and encapsulating the details of the networking protocol. We have provided support for applications susceptible to hardware and operating system failures by providing a piece of software that can respond to the status of terminated applications, and also serves to provide applications with a means to recover from node failure by backing up data and distributing it throughout

the network. We have provided support for applications that use data from many sources, by providing a set of containers for the data, along with a method for distributing and assembling the data.

Our framework, in its current form, is best suited to applications that need to be responsive and interactive. The data aggregation framework provides good support for applications to gather data from sensors about what users are doing, potentially leading to the use of a number of modalities. The consistent use of multithreading throughout the framework allows it to handle network traffic, while it continues working, so that the application remains responsive.

We make no pretension towards bandwidth efficiency. In our applications, we have sought to reduce unneeded flow of information, but the framework is not built to be lean or mean. The Message structure uses six 32 bit integers for tagging. This is most certainly excessive. Similarly, throughout the applications, we have used fairly loose packing for the data (using 32 bit integers throughout). On the other hand, if network bandwidth were at a premium, the fundamental structure of things in the framework would not need to change – all that would need to change is the method for packing the data.

We have used TCP for reliable data exchange. It is worth remembering, however, that TCP does not guarantee that data will get from one host to another, it only guarantees that if the data does get there, it will all get there together, and the data from a single connection will arrive in order. This means that applications that run where the network is faulty must still anticipate faults and handle them gracefully.

Our framework still operates at the level where the application programmer needs to write instructions for each individual node. We have not provided a system where a programmer could give instructions to the system as a whole, and have the system figure out what to do.

Applications where precision and accuracy are most important are supported by our framework, but require extra work on the part of the programmer. A significant short-coming of our framework is that we have not directly provided support for control and resource-sharing

logic. We have discussed patterns for such logic in broad terms (Chapter 2). Rolling these patterns into a software module would contribute significantly to the ease of developing distributed applications.

An obvious area for improvement to the framework, is to move away from the limitation of IP enabled computers. It would be useful to provide services for different types of networking protocols. This would allow the framework to expand its usefulness to bluetooth enabled and embedded devices. The inclusion of routing facilities would expand the usefulness of the system into ad hoc networking situations. Including more sophisticated geometrical intelligence in the MachineNode would aid the usefulness of the framework for applications where computational units are highly mobile, and where the orientation of the units matters. Adding facilities for describing location symbolically would serve to support applications where the relevant aspect of the location is not its coordinates, but its heuristic significance (i.e. in the bedroom).

Another area of improvement to the framework is the backup service. Currently, there is very little security built into the system (other than that the backup service will only give data back to the host/port combination that sent it). There is nothing to stop one node from impersonating another node, and stealing or modifying its data. Additionally, there are only minimal safeguards in place for ensuring that all of the copies of data in the network are consistent and up to date.

GUI developers on just about every platform use some sort of toolkit, which insulates them from the complexity and tedium of handling user interface elements. It is burdensome to require the application programmer to handle intersection tests with all of the buttons and other elements that might exist in a user interface. A natural extension of such toolkits would be a distributed user interface toolkit. In our application, we demonstrated the feasibility of the basic elements that would be needed for such a toolkit : buttons, sliders, and sessions. A user interface toolkit that handled distributed events related to interface elements, and managed sessions for those user interface elements would be very valuable. Programmers could create interfaces where multiple users could participate simultaneously from multiple locations, and elements of the user interfaces could be handled according to

the platform and modality of the receiver.

Improvements to the user input system described in 4.3 could be made by better taking advantage of the many available sources of data. An improved system might employ truer multi-camera tracking, with sleeker positioning of new cursors. Units could agree, not only on who should control a new cursor, but where it should appear, given other cursors in the system. Another improvement to the system might come in the form of distributed gesture recognition, where units in the system voted, or otherwise communicated with each other about what gestures they saw in their own streams of data.

Our framework provides a good foundation, by providing services aimed at the basic needs of distributed applications, and distributed applications that use sensor data. Future directions for service suites, based on our framework, include suites for advertising resources (this would support many ubiquitous computing applications), context awareness, sensor fusion, and data fusion.

Appendix A

System and Services Summary

A.1 Is this system for you?

Our framework was designed to work on computers that support TCP, and that have multi-threading capabilities. The framework has been used and tested under Linux and Mac OS X.

The messaging system is for people who want to do distributed computing, without worrying about networking details. It is for people who need reliable connections, but don't want the TCP connection model, and don't want to write their own networking protocol.

The messaging system does not allow developers to carefully control the use of network or operating system resources. The system uses a reasonable amount of system resources; each sending/receiving thread takes up time and resources, and each TCP connection takes time and resources. All data structures in the system are loosely packed, using 32 bit integers

The process management and data backup services are for people who want to write robust, distributed applications on somewhat unreliable hardware. The process management services are for use where programs may crash intermittently, due to problems with the underlying system. It is also for use where operating system or hardware failures can be

detected directly, or where frequent restarts are a symptom of failure. The data backup services are designed to help protect against intermittent, scattered outages. They are most useful where the state of programs can be summarized in a data structure or data structures.

The data backup service can't necessarily protect against data loss when large portions of the network go out simultaneously. Also, the data backup service does not provide strong mechanisms to make absolutely sure that all copies of data throughout the network are up to date and consistent.

The data aggregation system is for applications that want to share data gathered by lots of sensors, but don't want to worry about the details of getting the data back and forth. The data aggregation system is at its best when sensors collect data about multiple, identifiable phenomena at each site, and there is some function that can show how data from different sources are related.

The data aggregation system provides an interface where applications must request data from each other, so the application must know what other hosts it should ask for data. The data aggregation system does not solve the data association problem, or the alignment problem.

A.2 Summary of Services

The Messaging system provides services to send and receive messages asynchronously. The Messenger provides a callback registry, which allows the application to react to incoming messages in an event-driven manner. The Messenger allows applications to request persistent TCP connections between hosts, to reduce the overhead of setting up and tearing down connections, and to ensure that TCP's guarantees about complete, in-order delivery hold.

The Process Management system provides services to automatically restart programs that crash, and to automatically reboot computers where critical faults are detected. The Data Backup system allows applications to store data, for later retrieval. Data is propagated out to the network to provide robust storage, in the case of node failure. When a program

fails, and then restarts, it can retrieve data from the Process Manager when it restarts. If the computer needs to be rebooted, when the Process Manager restarts, it gathers up left behind data from other hosts on the network. Then, when the application restarts, it can retrieve its data from the Process Manager.

The data aggregation system provides a structure to organize data that is gathered from many sensors. The Aggregator manages the production, sending, receiving and organization of data. The application must know what its data is like and how to collect it, but it does not need to worry about sending or receiving it. Aggregators send their data only to hosts that have requested it. This allows the application to manage network utilization, by only requesting data that is relevant. The Aggregator provides a callback registry, which allows the application to react to data-driven events.

Appendix B

PointerList API

B.1 PointerList <YourTypeHere*>

The PointerList class implements a templated doubly linked list. It is designed to be used to store pointers to objects/memory that have been dynamically allocated.

The list has an internal "reference" which can be manipulated by the application. This reference is used to make iteration through the list, and certain other operations, fast.

When doing operations that involve searching the list (like the AddAfter() call), if there is more than one copy of a pointer in the list, the first found copy of the element is the one affected. The list is searched starting at the head (front) and ending at the back (end).

The PointerList is not thread safe. This decisions was made so that applications that use the PointerList do not necessarily have to link against libpthread. Applications must wrap the list with a mutex for thread safe adding, removing, and searching.

To instantiate the list, do:
PointerList <YourTypeHere *>mylist;
OR
PointerList <YourTypeHere *>* mylist = new PointerList <YourTypeHere * >;

To iterate through a list, do:

```
for(YourTypeHere* temp = mylist->Start(); temp!=NULL; temp = mylist->Next()){
...
}
```

To iterate through a list, where you may remove elements from the list, do:

```
YourTypeHere* temp = mylist->Start();
  while(temp != NULL){
    if(condition){
      mylist->Remove(temp);
      temp = mylist->Get();
    }
  else{
    temp = mylist->Next();
  }
}
```

bool Add(YourTypeHere *data)

Adds the given data to the front of the list. All Add* functions return true upon successful completion, and false in case of failure. All Add* functions will not add an element to the list whose value is NULL (or 0).

bool AddToFront(YourTypeHere* data)

Adds the given data to the front of the list. Returns true upon successful completion, and false in case of failure.

bool AddToBack(YourTypeHere* data)

Adds the given data to the back of the list. Returns true upon successful completion, and false in case of failure.

bool AddAfter(YourTypeHere* data, YourTypeHere* after)

Adds the given data to the list, placing it after the given value. If the "after" value is not in the list, the function returns false and the element is not added to the list. This

function searches the list for the "after" value, and so, if repeated "AddAfter" calls are going to be used, it will be more efficient to use the second version of "AddAfter".

bool AddAfter(YourTypeHere* data)
Adds the given data to the list, placing it after the value pointed to by the internal reference. If the reference is not pointing to anything, the function returns false and the element is not added to the list.

bool AddBefore(YourTypeHere* data, YourTypeHere* before)
Adds the given data to the list, placing it before the given value. If the "before" value is not in the list, the function returns false and the element is not added to the list. If the "before" value is NULL, the function returns false and the element is not added to the list. This function searches the list for the "before" value, and so, if repeated "AddBefore" calls are going to be used, it will be more efficient to use the second version of "AddBefore".

bool AddBefore(YourTypeHere* data)
Adds the given data to the list, placing it before the value pointed to by the internal reference. If the reference is not pointing to anything, the function returns false and the element is not added to the list.

bool AddInOrder(YourTypeHere* data,
bool(*compare)(YourTypeHere* thing1, YourTypeHere* thing2))
Finds the first element in the list where the comparison function is true, and adds the given data before that element. If there are no elements in the list when this function is called, it simply adds the element to the list.

YourTypeHere* RemoveFromFront()
Removes the first element of the list, and returns it.

YourTypeHere* RemoveFromBack()
Removes the last element of the list, and returns it.

YourTypeHere* Remove(YourTypeHere* item)

Finds and removes the first element of the list matching the given value, and returns it.

YourTypeHere* Remove()

Removes the element currently pointed to by the internal reference and returns it. The internal reference pointer is moved to the next element.

bool RemoveAll()

Removes all elements from the list, but does not call the C++ delete operator.

bool DeleteFromFront()

Removes the first element of the list, and deletes it with the C++ delete operator.

bool DeleteFromBack()

Removes the last element of the list, and deletes it with the C++ delete operator.

bool Delete(YourTypeHere* item)

Finds and removes the first element of the list matching the given value, and deletes it with the C++ delete operator.

bool Delete()

Removes the element currently pointed to by the internal reference and deletes it with the C++ delete operator. The reference pointer moves to the next element in the list.

bool DeleteAll()

Removes all elements in the list, and calls the C++ delete operator for each of them.

YourTypeHere* Start()

Sets the internal reference to the first element in the list, and returns that element.

YourTypeHere* End()

Sets the internal reference to the last element in the list, and returns that element.

YourTypeHere* Next()

Sets the internal reference to the next element in the list and returns that element. Returns NULL if the reference is not set prior to the call, or if the reference was at the end of the list prior to the call.

YourTypeHere* Next(YourTypeHere* item)

Sets the internal reference to the element after the first element in the list matching the given value. Equivalent to a call to Set(YourTypeHere*) followed by a call to Next(). Returns the value pointed to by the reference at the end of the call.

YourTypeHere* Prev()

Sets the internal reference to the previous element in the list and returns that element. Returns NULL if the reference is not set prior to the call, or if the reference was at the head of the list prior to the call.

YourTypeHere* Prev(YourTypeHere* item)

Sets the internal reference to the element after the first element in the list matching the given value. Equivalent to a call to Set(YourTypeHere*) followed by a call to Prev(). Returns the value pointed to by the reference at the end of the call.

bool Set(YourTypeHere* item)

Sets the internal reference to the first element in the list matching the given value.

bool IsIn(YourTypeHere* item)

Searches for the given element in the list. Returns true if the element is in the list, and false if it is not.

YourTypeHere* Get()

Returns the element that the internal reference is pointing to. Returns NULL if the internal reference is not set.

YourTypeHere* Get(int index)

Treats the list as an array, and returns the given element of the list. Get(0) returns

the first element of the list. Returns NULL if there are fewer than index elements in the list.

YourTypeHere* GetStart()

Returns the first element in the list. Does not alter the internal reference.

YourTypeHere* GetEnd()

Returns the last element in the list. Does not alter the internal reference.

YourTypeHere* GetNext()

Returns the element in the list after the element pointed to by the internal reference. Returns NULL if the reference is not set, or is at the end of the list. Does not alter the internal reference.

YourTypeHere* GetNext(YourTypeHere* item)

Returns the element after the first found instance of the given value. Does not alter the internal reference. Is not affected by the internal reference.

YourTypeHere* GetPrev()

Returns the element in the list before the element pointed to by the internal reference. Does NOT alter the internal reference. Returns NULL if the reference is not set, or is at the head of the list. Does not alter the internal reference.

YourTypeHere* GetPrev(YourTypeHere* item)

Returns the element before the first found instance of the given value. Does not alter the internal reference. Is not affected by the internal reference.

PointerList <YourTypeHere*>Detach(YourTypeHere* item)

Finds the (first instance of) the given value, and returns all subsequent elements in a new list. Elements in the new list are removed from the parent list.

bool Trim(int howmany)

Counts the given number of elements in the list, and deletes all elements after that

element. Returns true on successful completion, and false if there are less than the required number of elements in the list.

bool TrimAfter(YourTypeHere* after)

Finds the (first instance of the) given value in the list, and deletes all subsequent elements of the list. The given element remains in the list. Returns true on success, and false if the given value is NULL, or if it is not in the list.

bool TrimBefore(YourTypeHere* before)

Finds the (first instance of the) given value in the list, and deletes all previous elements of the list. The given element remains in the list. Returns true on success, and false if the given value is NULL, or if it is not in the list.

int Length()

Returns the number of elements in the list. It actually goes through and counts the elements in the list, rather than maintaining a count as elements are added and removed.

void ApplyToAll(void (*function)(YourTypeHere*)

Takes the given function and applies it to each element in the list. The function to be applied will take a single element as an argument. Do NOT add or remove nodes to the list inside this function.

Appendix C

Messaging API

The messaging subsystem is a collection of objects and functions to facilitate the exchange of data between processes running on different hosts. Applications package their data in Messages, which are then given to the Messenger. The Messenger then takes control of the messages and sends them. Messages are received asynchronously by the Messenger. Applications register callback functions for different types of messages and receive messages in an event-driven loop. The application must periodically call Messenger::ProcessMessages, which dispatches messages to the appropriate callbacks, in addition to handling internal messages. The architecture is asynchronous, multi-threaded, and uses blocking I/O.

C.1 Message

The Message is the main unit of currency in the system. It is mostly a container for tagging data, and holding onto buffers.

The member variables of the Message class are: int type;
int extra;
int timestamp;

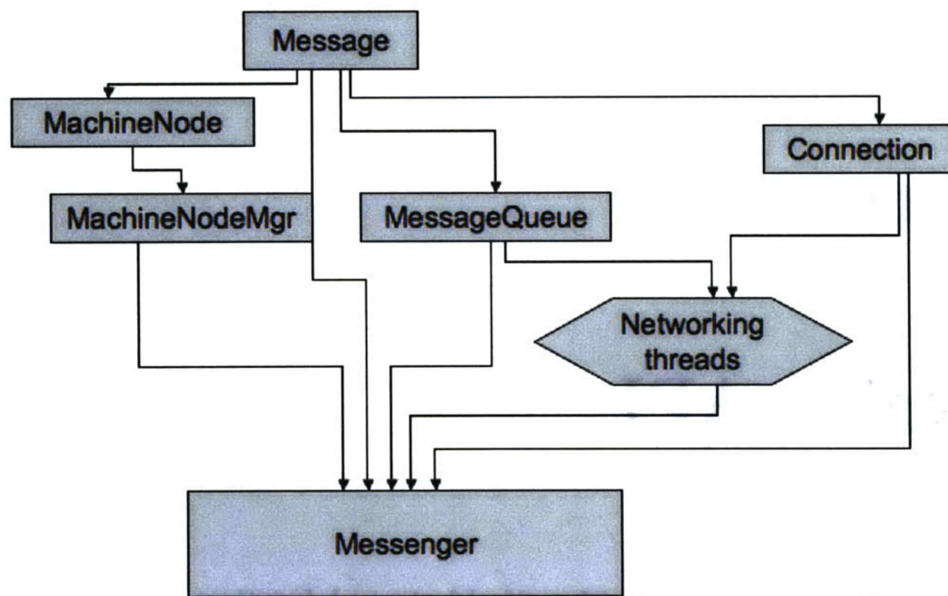


Figure C-1: The dependency diagram for objects in the Messaging subsystem.

int seq_no;

int sender;

int receiver;

int length;

char* message;

Message()

Creates a blank messages with all members initialized to 0 or -1.

Message(Message* msg)

Creates a new message that is a copy of the given message. Creates a copy of the message buffer.

Message(char* msg)

Creates a message that contains the (parsed) information contained in "msg".

Message(int type, int extra)

Creates a blank message, but initializes "type" and "extra" to the given values.

Message(int receiver, int sender, int type, int extra)

Creates a blank message, but initializes, receiver, sender, type and extra to given values.

void SetMessage(char* message, int length)

This function is VERY IMPORTANT. Data in an application should ultimately be packaged into byte arrays. This is, as opposed to null terminated strings. The good thing about using byte arrays is that there can be zeros in the middle, and you can transmit binary data easily. The bad part about using byte arrays is that you need to be told how long the buffer is. The byte array could contain a null-terminated string, but you still need to use strlen+1 to say how long the buffer is. To use this function, package your information as a byte array and figure out its length. When you pass in a buffer, the message "takes control" of the memory (and will try to delete it when its destructor is called).

void ReadHeader(char* msg)

This function takes a byte array and parses out the header, but does not create a new buffer or copy the data into its message.

void ParseString(char* msg)

This function takes a byte array and parses out the header, and uses the length parameter to allocate a buffer, and copy in the relevant information from the remainder of the buffer.

char* CreateString(int &len)

This function returns a byte array, which contains the information contained in the

Message. An integer should be passed in, which will be modified. This is to indicate the size of the buffer returned.

void SetExpiration(int future)
This function sets the expiration time of a message (in seconds). Internally, when this message is about to be sent, its expiration time is checked, and if it has passed, the message is quietly deleted. Do not set the expiration time, and there will effectively be no expiration time. Beware of edge effects when setting expiration times 1 second in the future.

Message* Reply()
This function creates a new message, where it has swapped its own sender and receiver fields. Otherwise, the returned message is blank.

void Print()
Prints the information of the message in this format: || sender, receiver || type, extra (timestamp, seq_no) length message If the message is a byte array, the end will look like garbage.

C.2 Message Types

MESSAGE_TYPES 20
MESSAGETYPE_ERROR 0
MESSAGETYPE_COMMAND 1
MESSAGETYPE_NOTIFY 2
MESSAGETYPE_QUERY 3
MESSAGETYPE_PASS 4
MESSAGETYPE_COORDINATE_REQUEST 5
MESSAGETYPE_COORDINATE_RESPONSE 6
MESSAGETYPE_NEIGHBOR 7

MESSAGETYPE_CONNECTION 8
MESSAGETYPE_CONTINUATION 9
MESSAGETYPE_ROUTING 10

ERROR_NOSOCKET 1
ERROR_BADHOST 2
ERROR_BINDSOCKET 3
ERROR_CONNECTIONTERMINATED 4
ERROR_NEEDREBOOT 5
ERROR_NOPROGRAM 6

COMMAND_QUIT 'Q'
COMMAND_STOP 'S'
COMMAND_CONTINUE 'C'

NEIGHBOR_ADD 'A'
NEIGHBOR_REMOVE 'R'
NEIGHBOR_MODIFY 'M'

CONNECTION_START 'S'
CONNECTION_END 'E'
CONNECTION_TERMINATED 'T'

MESSAGEFORMAT_P2P 1
MESSAGEFORMAT_PERSISTANT 2
MESSAGEFORMAT_GROUP 3

C.3 Messenger

The sending and receiving inside the Messenger happens on a specific port. You designate the port in the constructor. Messengers on different physical machines must be listening / talking on the same port to be able to communicate. If an error occurs during setup (like, if the server cannot bind to the socket), an error message is placed on the queue and will be dispatched in the first ProcessMessages() call.

Once messages are placed on the queue, the application should not attempt to modify them in any way. DO NOT REUSE MESSAGES. They are deleted with the C++ delete operator when they are sent.

DEFAULT_LISTEN_PORT 4690
MAX_MESSAGE_LENGTH 512
MESSAGE_HEADER_SIZE 28

Messenger()
Sets up the Messenger to listen and send on the DEFAULT_LISTEN_PORT.

Messenger(int port)
Sets up the Messenger to listen and send on the given port.

Messenger(int port, bool using_sender, bool using_server)
Sets up the Messenger to send and/or receive on the given port. Allows the application to disable either the main client thread or the main server thread. The Messenger will still be able to set up persistant connection, even if the main client is not running. However, the Messenger will not be able to handle the setup of incoming persistant connections if the main server thread is not running.

int GetAddress()
This retrieves the integer value of the local IP address from the network table.

int ProcessMessages()

This function must be called periodically to dispatch messages to the registered callbacks. Messages for which there are registered callbacks are deleted. Messages without registered callbacks are put back on the queue.

The ProcessMessages function handles a maximum of 15 messages, before it returns. (This is to prevent starvation of the main application, if messages continue to arrive). The function returns the number of messages it has processed.

If there is no callback registered for the MESSAGETYPE_ERROR message type, these messages will also get placed back on the queue, and the application will not otherwise be notified of the problem.

void RegisterCallback(int type, void (*function)(Message*, void*)

Callback functions registered with the Messenger must have the signature: void myfunction(Message*, void*). The void* argument is to package whatever piece of information the application needs to its callback. This is very useful, for example, when the callback needs to be a method of a class. To do this, you declare a static method of the class, and then use the void* to package a pointer to the class. Your static member uses the pointer to call the proper method of the class, with access to all the data members, etc. This methodology is somewhat borrowed from GTK.

void RegisterCallback(int type, void (*function)(Message*, void*), void* data

You can use this version if you have no extra data to pass to your callbacks. (It just looks slightly cleaner).

Message* CreateMessage(int type, int extra, char* data, int len)

This function takes the provided arguments, and makes a new Message with the appropriate information. It sets the sender to be the local address. The application must set the receiver. If you know to whom you want to send the message (and you don't plan to make any copies), you can use the second SendMessage() function below.

void SendMessage(MachineNode* receiver, int type, int extra, char* data, int len)

This function takes the given information and sends it to the given host. It sets the sender and receiver of the message automatically.

void SendMessage(Message* message)

This takes a created Message and sends it. Do NOT try to modify a message once you have passed it to this function. If you do not set the receiver, then the message will come back to you, since receiver == 0 is localhost.

void SendMessageCopy(Message* message, MachineNode* receiver)

This function takes the given message, makes a copy, and sends the copy to the given node. It does not delete or alter the original message.

void SendMessageToHosts(MachineNodeMgr* hosts, Message* message)

This message takes a MachineNodeMgr and sends the given Message to every host represented in the MachineNodeMgr. It deletes the original message when it is done.

void SendMessageToAll(int type, int extra, char* data, int len

This function takes the given information, makes a message out of it, and sends a copy to every node in the internal "neighbors" MachineNodeMgr.

void SendMessageToAll(Message*)

This function takes the given message and sends it to every node in the internal "neighbors" MachineNodeMgr.

Message* SendMessageTo(Message* message, int address, int port)

Sends the given Message to the given address on the given port. This function sends the Message synchronously. The Messages returned by this function are possible error messages generated when trying to send the message. It should be noted that if this send fails due to a problem with the remote host, it may take a long time for the function to return.

Message* SendMessageTo(Message* message, int port)

Sends the given Message to the local address on the given port. This function sends the Message synchronously, but an error with a local address will return quickly if there is no server running.

int RequestConnection(int address)

These functions set up a persistant connection with the host with the given address. If this host does not exist in the internal "neighbors" MachineNodeMgr, it is created.

int RequestConnection(char* address)

These functions set up a persistant connection with the host with the given address. If this host does not exist in the internal "neighbors" MachineNodeMgr, it is created.

int RequestConnection(MachineNode* host)

This function sets up a persistant connection with the given host. You can only request outgoing connections. You can't request that a host that has been sending you a lot of data set up a persistant connection with you – they must initiate the connection. This sort of request must go through the application layer. There is no support for it in the Messenger. The function returns 0 if everything went well, and -1 on an error. If the connection to the remote host failed, an error message is generated and placed on the queue.

int CancelConnection(int address)

These functions close a previously requested persistant connection to the host with the given address. If the address is not represented in the internal "neighbors" MachineNodeMgr, the function returns -1;

int CancelConnection(char* address)

These functions close a previously requested persistant connection to the host with the given address. If the address is not represented in the internal "neighbors" MachineNodeMgr, the function returns -1;

int CancelConnection(MachineNode* host)

This function closes a previously requested persistant connection to another host. The function returns 0 if everything completed successfully, and -1 on an error.

MachineNode* AddNeighbor(char* address)

Adds a MachineNode with the given address to the internal neighbor MachineNodeMgr.

MachineNode* AddNeighbor(int address)

Adds a MachineNode with the given address to the internal neighbor MachineNodeMgr.

MachineNode* AddNeighbor(Message* message)

Takes the given message, parses it, and adds the parsed MachineNode to the internal MachineNodeMgr.

MachineNode* AddNeighbor(MachineNode* host)

Takes the given MachineNode and adds to the internal MachineNodeMgr. The application should not attempt to delete the MachineNode, once it has been passed to this function. (It does not make a copy.)

MachineNode* GetNeighbor(char* address)

Returns the MachineNode in the internal MachineNodeMgr that matches the given address.

MachineNode* GetNeighbor(int address)

Returns the MachineNode in the internal MachineNodeMgr that matches the given address.

int RemoveNeighbor(char* address)

Removes the MachineNode with the given address from the internal neighbor MachineNodeMgr.

int RemoveNeighbor(int address)

Removes the MachineNode with the given address from the internal neighbor MachineNodeMgr.

int RemoveNeighbor(Message* message)

Takes the given message, parses it, and removes MachineNode matching the address of the parsed MachineNode from the internal MachineNodeMgr.

int RemoveNeighbor(MachineNode* host)

Removes the given MachineNode from the internal MachineNodeMgr. It is assumed that the given MachineNode actually came from the internal MachingNodeMgr, and is not a copy or something. Unpredictable things will happen if the given MachineNode is not actually in the internal MachineNodeMgr. To delete from a copy, use the RemoveNeighbor(int) function.

MachineNode* Transform(MachineNode* host, MachineNode* neighbor)

Transforms the host into the neighbor's coordinate frame. Returns a node with the relevant transform.

void SetAsNeighbors(MachineNode* host, MachineNode* neighbor)

This function notifies the node about the neighbor. It takes the neighbor MachineNode, transforms it by the "node" MachineNode's transformation, and then sends that to the node. It forges the sender address so that it looks like the neighbor notification came from the neighbor MachineNode.

void SetupNeighborMess()

This function sets every node in the table as a neighbor of every other node in the table. It does not try to set each node as a neighbor of itself.

void SetupNeighborLattice(int degree)

This would more aptly be name "SetupNeighborChain()". This function goes through the neighbor table, and for each node, gives it the given number of neighbors from the table. The structure will be like a chain.

C.4 MachineNode

The MachineNode is a class for keeping track of nodes in the network. "Machine" Node comes from thinking of a State Machine. So, it refers to nodes running a state machine. This class also keeps track of physical topology / transformations. When dealing with physical transformations, the assumption is that the local node (the host maintaining the table) is at (0,0,0) and is axis-aligned. All other hosts are translated and rotated with respect to the local node. This class also has data members for keeping track of persistant connections. I do not recommend messing with these – leave that to the Messenger.

int address
int type (largely unused)
int t //Time offset
int x[3] //position
int r[3] //rotation (unsupported)
int s[3] //size/extent

MachineNode()
The default constructor for the MachineNode initializes all of the geometry to with appropriate zero values.

MachineNode(Message* message)
This constructor initializes the geometry to zero, but then parses out the address and physical transformations.

void Copy(MachineNode* host)
This function takes another MachineNode and copies all of its information (less connections and threads).

int Parse(Message* message)
This function takes a Message and parses out the address and physical transforma-

tions.

int ParseString(char* string)

This function takes a string, assumes that it is a packaged MachineNode, and parses out the address and physical transformations.

char* CreateString(int& length)

This function packages a MachineNode as a byte array. The len variable is passed in and modified to indicate the length of the returned byte array.

void Print()

This function prints the information of the MachineNode in the following format: || address || time-offset (X translation, Y translation, Z translation) ; (X size, Y size, Z size)

void Update(MachineNode* host)

This function takes another MachineNode and copies the physical transformation information, over-riding whatever information it previously had.

void SetAddress(int address)

This function stores the given integer as the address.

void SetAddress(char* address)

The function takes in a string, such as "18.85.18.131", or "ozy.media.mit.edu" and converts it to an integer, and stores that integer.

void SetType(int type)

This is to set the internal type variable. I had envisioned this for typing things as a tile or a robot or a PC, or whatever other device we might have.

void SetTime(int time_offset)

I envisioned this holding information about the offset between clocks on various hosts.

void SetTranslation(int X, int Y, int Z)

This is to set the translation vector (where the given host is with respect to the local host).

void SetTranslation(int* translation)

Here, the translation vector should take the form int v[3] = X,Y,Z.

int* GetTranslation(int* translation)

A vector (int[3]) should be passed in. This is then filled with the relevant information, and returned.

void SetSize(int X, int Y, int Z)

This assumes that nodes can be described by boxes. The boxes are X wide by Y tall, and Z deep.

void SetSize(int* size)

Vector version. int size[3] = width, height, depth;

int* GetSize(int* size)

A vector (int[3]) should be passed in. This is then filled with the relevant information, and returned.

int* Transform(int* point)

This function takes a point int X, Y, Z format. It transforms the point, from the outside, into the coordinate frame of the node, using the internal translation (and rotation, at some point in the future), and stores the result in the vector that is passed in. POINTS PASSED TO THE FUNCTION ARE MODIFIED, and then returned.

int* InverseTransform(int* point)

This function takes a point int X, Y, Z format. It transforms the point from the coordinate frame of the node, out to the local host's coordinate frame. It stores the result in the vector that is passed in. POINTS PASSED TO THE FUNCTION ARE MODIFIED, and then returned.

bool Inside(int* point)

This function takes a point, in the external coordinate frame, and determines if it is inside the extent of the node. It transforms the point internally, so don't transform it before passing it in. It does NOT modify points that are passed in.

bool InsideWithTolerance(int* point, int* epsilon)

This function takes a point, in the external coordinate frame, and determines if it is inside the extent of the node, with the given tolerance, epsilon. There can be different tolerances in each of the three directions. It does NOT alter points that are passed in.

int SquaredDistanceToCenter(int* point)

This takes a point, in the external coordinate frame, and determines the (squared) distance to the center of the node. It does the squared distance to avoid expensive sqrt calls, and most cases I could think of, the squared distance was just as good as the plain distance.

int SquaredDistanceToExtent(int* point)

This takesa point, in the external coodinate frame, and determines the (squared) distance to the nearest border. If the point is outside the volume, the return value is positive. If the point is on a border, the return value is zero. If the point is INSIDE the volume, the return value is NEGATIVE.

bool IsLeft()

Determine if the node is to the left of the local host.

bool IsRight()

Determine if the node is to the right of the local host.

bool IsUp()

Determine if the node is above the local host.

bool IsDown()

Determine if the node is below the local host.

C.5 MessageQueue

The MessageQueue object serves as the interface between the main program and the sender and server threads. Applications should not really need to interact with the message queues. The exception is when the application elects to bypass the internal Messenger::ProcessMessages function, and handle the incoming messages itself (which I don't recommend, because there are internal messages to handle.)

int Enqueue(char* message)

The function takes the character buffer, creates a message, and puts it on the queue.

int Enqueue(Message* message)

The function takes the given message and puts it on the queue. Return value of -1 means there was a problem enqueueing the message.

Message* Dequeue()

Returns the first message on the queue.

Message* DequeueByType(int type)

Returns first message on the queue with the given type.

Message* DequeueByFormat(int format)

Returns the first message on the queue with the given format. Don't worry too much about this. It doesn't mean the data is any different – It is mostly for ensuring that certain threads get or don't get messages they should or shouldn't get.

Message* DequeueBySender(int address)

Returns the first message on the queue with the given sender.

Message* DequeueByReceiver(int address)

Returns the first message on the queue with the given receiver.

C.6 MachineNodeMgr

The MachineNodeMgr is a table for MachineNodes. It holds the linked list representing all known hosts. It also provides some higher level geometric intelligence (like, finding the host that is nearest to a given point.)

int SetClient(MachineNode* host)

This function designates the given node as the client node. I had originally intended to use this for debugging (so every tile would dump back to some client), but I have never actually used it.

int SetSelf()

This function creates a node and sets its address using the gethostname() call.

int SetSelf(MachineNode* host)

This function uses the given node as the self node.

MachineNode* Add(char* address)

This function takes a string in the style of "18.85.18.131" or "ozy.media.mit.edu", creates a node with that address, and adds it to the list. It returns the created node. Geometrical information is left blank.

MachineNode* Add(int address)

This function takes an address as an integer, creates a node with that address, and adds it to the this. It returns the created node. Geometrical information is left blank.

MachineNode* Add(Message* message)

This function takes a Message, creates a node, parses the Message into the node, and adds the node. It returns the created node. Any geometrical information contained in the Message is reflected in the added node.

MachineNode* Add(MachineNode* host)

This function takes a pre-allocated MachineNode and adds it to the list. It assumes

responsibility for the memory, and will delete the node if it is not otherwise removed. There can only be one node with a given address in a MachineNodeMgr. If the Add function finds another MachineNode with the same address, it will replace the geometrical information with the given information using MachineNode::Update(), and then delete the passed in node. It will return either the node that is added (passed in), or the node that it finds with the same address. This function does not make a copy.

int Remove(int address)

This function takes an address as an integer and removes the node with the given address.

int Remove(char* address)

This function takes an address as a string and removes the node with the equivalent address.

int Remove(Message* message)

This function takes a Message and removes the host designated in the Message. If the Message is an error message, it looks at the receiver, and removes that node (these error messages are created in the Messenger). Otherwise, it parses the message as usual, and removes the node with a matching address.

int Remove(MachineNode* host)

This function takes the given MachineNode and removes it from the list. This should be a node that is IN the list (that you got with one of the GetNode(*) functions), and not some copy that the application has kept outside of the list.

MachineNode* GetSelf()

Returns the designated "Self" node.

MachineNode* GetClient()

Returns the designated Client node.

Pointerlist <MachineNode*>* GetNodes()

Returns the list of MachineNodes.

MachineNode* GetNode(MachineNode* after)

Returns the next node in the list, after the given node. Passing in NULL returns the head of the list.

MachineNode* GetNode(int address)

Returns the node in the table with the given integer address.

MachineNode* GetNode(char* address)

Returns the node in the table with an equivalent address.

C.7 Connection

This class is designed to minimize the pain of dealing with TCP network connections. It is designed to be used to send and receive information packaged as Messages. Unfortunately, it keeps with the TCP connection model: there is a server and a client. The server binds to the port and waits to receive connections. When it receives a connection, it creates a new Connection, which then receives the message. There are options for using non-blocking I/O. I tested the receiving functions in non-blocking mode, and they should work properly. I have not checked the send functions in non-blocking mode, and I think it is possible for the send to return before it has sent all of the data, so beware.

void Initialize()

Zeroes out all of the (private) data in the connection – resets the local address and port.

void SetPort(int)

Sets the port on which the connection will try to communicate. Does nothing if the connection is already open.

void SetHosts(int remote, int local)

Sets the remote and local addresses to the given integer addresses. The remote address is the address to which you will connect if you try to start a connection. Remote addresses do not make any sense for incoming connections, but they are convenient for book-keeping. If the connection is already open, it does nothing.

void SetHosts(MachineNode* remote, MachineNode* local)

Sets the remote and local hosts using the MachineNode abstraction.

void SetRemoteHost(int remote)

Sets just the remote address (to which you will connect) to the given address. Does nothing if the connection is already open.

void SetRemoteHost(MachineNode* remote)

Sets just the remote address, using the MachineNode abstraction.

int GetRemoteAddress()

Returns the integer value of the remote address.

void SetNonBlocking()

Puts the connection into non-blocking mode. (Can be done while connection is open). Use with caution!

void SetBlocking()

Puts the connection into blocking mode.

void DisableNagle()

The Nagle algorithm is a pretty simple little idea. For every packet sent over TCP, an ACK is sent in return. The guys designing TCP said "gee, it is kinda wasteful to use a whole packet to send a 1 bit acknowledgement." So, using the Nagle algorithm, they piggy-back the ACKs on real data, going the other way. If there is no data going the other way, however, (as is generally the case in this system, since it is easier to program), if the Nagle algorithm is enabled, the ACKs will never get sent,

and no more data will come. So you disable the Nagle algorithm to send the ACKs immediately.

void EnableNagle()

Enable the Nagle algorithm.

void DisableTimeWait()

When you open a TCP connection, and then you send data over the connection, it is possible that some of the data bound for the connection is delayed along the way. To this end, when you close a connection, you don't want to reuse the port immediately, since this stray data might come into the port later on and mess you up. So, you wait for some amount of time, until you can be reasonably sure there is no stray data coming to mess you up. This enables the port to be reused immediately.

void Close()

Closes the connection using the system close, and zeroes out the file descriptor. Does not modify the local or remote host addresses, or the port. Connection is ready to be reused.

bool IsOpen()

Returns false if the socket file descriptor is not a real file descriptor (fd ¡= 0). It does not do any more sophisticated tests to make sure the socket is good to read/write to.

Message* SetupToWaitForConnection()

This sets up the "server" end of a TCP connection. It does the bind() and listen() calls of the setup. If it is unsuccessful, creating or binding to the socket, it will return an error message. If all goes well, it returns NULL.

Connection* WaitForConnection()

This function waits for new connections to come in using the accept() call. This function will block if no connection is immediately available. Not sure what happens when the connection is in non-blocking mode. Returns a new Connection object for the accepted connection.

Connection* WaitForConnection(Connection*)

This function waits for new connections to come in using the accept() call. This function will block if no connection is immediately available. Not sure what happens when the connection is in non-blocking mode. It will use the given Connection, rather than creating a new Connection object. If the Connection is not open, it returns NULL.

Message* StartConnection()

This implements the client side of the connection. This initiates a connection to the stored remote address, on the stored port. If the connection is already open when this function is called, the old connection is closed. If the connection does not start successfully, an error message is returned. If all goes well, the function returns NULL.

int UseConnection(int file_descriptor)

This sets the internal file descriptor to point to the given file descriptor. This function is used, for example, when accepting new connections in a server.

Message* Receive()

This function receives messages from a connection. If the connection is not open, or if there is no data to read, it returns NULL. If the other side of the connection has closed the connection, it will detect this, close the connection, and return a connection message (which will be handled by the Messenger). Otherwise, this function allocates and returns the message that has just been read off the connection.

Message* Send(Message*)

This function sends Messages out over a connection. If the message is successfully sent, the function returns NULL. If there was a problem sending the message, the function returns an connection message, which will be handled by the Messenger. IMPORTANT: If there is a problem sending the message, this will happen because the receiving end of the connection has closed the connection. When a program tries to send something, and there is nobody listening, this raises the SIGPIPE signal. There is supposed to be a socket option for disabling this, but I couldn't get it to

compile. To work around this, in the Messenger, it sets up to catch SIGPIPE with the SIG_IGN macro. If you are using the Connection object out of the context of the Messenger, you will also need to catch SIGPIPE.

Appendix D

Network Client Interface

The Network Client is a piece of software that was written for communicating with various network pieces (like the tiles, and the robots). There are internal commands, telling the program to do things, and network commands, which are things to be sent over the network. Internal commands are capitalized. Network commands are lower-case. You can create files with sequences of commands, and then load them in. It will ignore blank lines and lines starting with '#'. It is otherwise quite finicky about whitespace.

– comment

D.1 Internal Commands

Q – Quit

H – Help

F filename – read script from file

T IP_address – set target receiver (T switches among existing targets. Use integer

versions to add targets.) (When using integer versions, new values for translation and extent override old values)

T1 IP_address

T3 IP_address x_translation y_translation z_translation

T6 IP_address x_translation y_translation z_translation x_extent y_extent z_extent

R IP_address – Removes the node with the given address from the internal network table

P IP_address – prints information about node with given address

M – Set up fully connected node mesh

L degree – set up lattice of connected nodes (specifically, a chain)

N IP_address IP_address – sets the two hosts as neighbors (shares geometric information)

D.2 Network Commands

Using <extra>with any network command sets the "extra" member of the message to be sent. The default value for "extra" is 0.

c <extra>string – send a command to current target c Q – send Quit command c C – send Continue command c S – send Stop command

n <extra>string – send a notification to current target

p <extra>string – send a pass to current target

q <extra>string – send a query to current target

Appendix E

Telephone

One of the simplest programs that we wrote was a telephone-like application for the tiles. In this application, one tile gets sound from its microphone, and sends it to another tile, who plays it through their speaker. To set up a telephone, one would have two sets of tiles: a sender and receiver, for each site.

The pseudo-code for this application is very simple (shown in figures E and E), and follows the pattern laid out in section 3.1.

```
Messenger *messenger;
void callback(Message* message, void* data){
  PlaySound(message->message);
}
int main(){
  messenger = new Messenger;
  messenger->RegisterCallback(MESSAGETYPE_NOTIFY, & callback, NULL);

  while(true){
    messenger->ProcessMessages();
  }
}
```

Figure E-1: This figure provides pseudo-code for a simple telephone receiver.

```
Messenger *messenger; MachineNode* receiver;
void neighbor_callback(Message* message, void* data){
  receiver = messenger->AddNeighbor(message);
  messenger->RequestConnection(receiver);
}
int main(){
  SoundSetup();

  char* buffer;
  receiver = NULL;
  messenger = new Messenger;
  messenger->RegisterCallback(MESSAGETYPE_NEIGHBOR, neighbor_callback);

  //wait to receive a notification about where we should send data
  while(receiver == NULL){
    messenger->ProcessMessages();
  }

  //send sound in a loop
  while(true){
    buffer = new char[SOUND_LENGTH];
    GetSound(buffer);
    messenger->SendMessage(receiver, MESSAGETYPE_NOTIFY, 0, buffer,
SOUND_LENGTH);
  }
}
```

Figure E-2: This figure provides pseudo-code for a simple telephone sender.

Appendix F

Process Manager / Backup Server Interface

The Process Manager serves two functions : it provides an interface for starting and stopping programs using the message passing interface (or the network client, described in Appendix D), and it offers a data backup service. It offers a service to automatically restart those programs, should they fail, and it offers a service to automatically reboot the machine, if a critical failure is detected. Data that is backed up using the process manager is propagated out to other machines, so that if the machine reboots, the data can be retreived, first by the Process Manager, and then by the application.

The port DEFAULT_PROCESS_PORT is 3680.

F.1 Program Management

Using automatic program restarts

The Process Manager will automatically restart programs that terminate abnormally (with some other return code than 0). By default, the Process Manager will restart

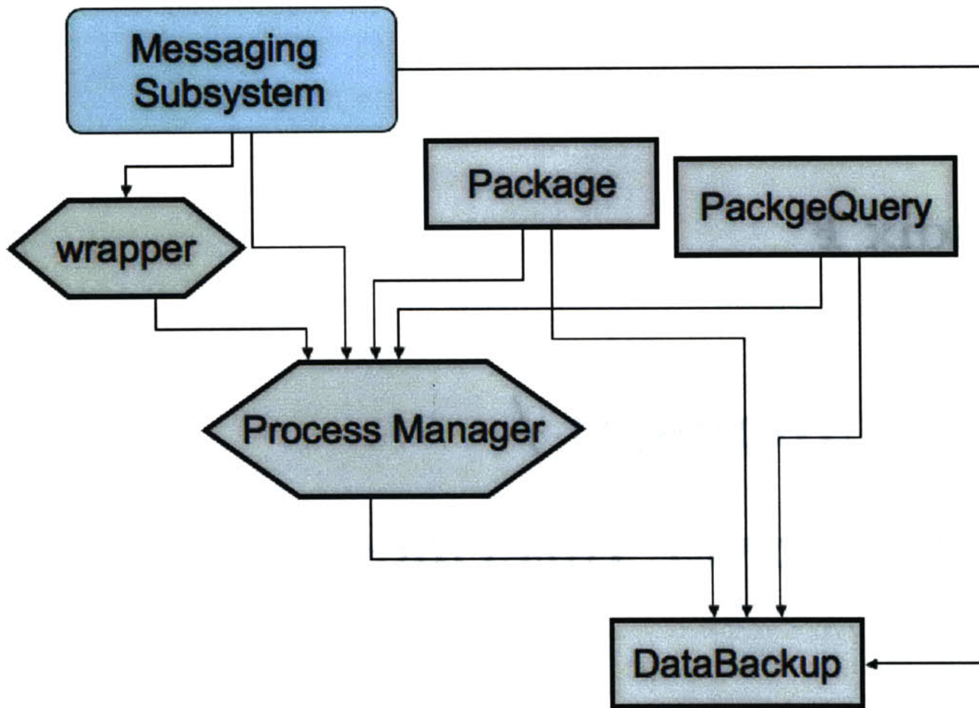


Figure F-1: The dependency diagram for objects in the Process Management and Data Backup subsystems.

programs indefinitely. (If they terminate with a return value of 0, they will not be restarted).

If set, the Process Manager will reboot the machine after a specified number of restarts. To set the maximum number of restarts, send a Message to the Process Manager on port DEFAULT_PROCESS_PORT, with type = MESSAGETYPE_NOTIFY, and extra = r. (case-sensitive. 'r' is for 'restarts'). The payload of the message is assumed to be a string, which is converted to an integer using atoi().

Using automatic reboot

The Process Manager will also automatically reboot the machine when a program returns with a specific error code. This allows programs that detect critical errors

while running to notify the Process Manager that the machine needs to be rebooted. This error code must be between 0 - 255, on a linux machine.

To set the code that a program will return to notify the Process Manager that it needs to reboot, send a Message to the Process Manager on port DEFAULT_PROCESS_PORT, with type = MESSAGETYPE_NOTIFY, and extra = 'c'. (case-sensitive. 'c' is for 'code'). The payload of the message is assumed to be a string, which is converted to an integer using atoi().

It is also possible to send a reboot code directly to the Process Manager. Send a Message to the Process Manager on port DEFAULT_PROCESS_PORT, with type = MESSAGETYPE_COMMAND, and extra = 'R'. (case-sensitive. 'R' is for 'Reboot').

Before the Process Manager reboots, it writes a file with a list of the processes that are running.

Starting a program

Send a Message to the Process Manager on port DEFAULT_PROCESS_PORT, with type = MESSAGETYPE_COMMAND, and extra = 'S' (case-sensitive. 'S' is for 'Start'), with the name of the process to start as the payload of the Message. The Process Manager will use the current values of the maximum number of restarts, and the reboot error code when starting the program.

When the Process Manager is told to start a program, it runs a separate program wrapper which actually restarts the designated program when it terminates abnormally. This wrapper program sends an error message to the Process Manager when the maximum number of restarts has happened, or when the program returns a special reboot code.

Terminating a program

Send a Message to the Process Manager on port DEFAULT_PROCESS_PORT, with type = MESSAGETYPE_COMMAND, and extra = 'K' (case-sensitive. 'K' is for 'Kill'), with the name of the process to kill as the payload of the Message. The name

to kill must be exactly the same as the name the program was started with (including the directory where the program is located.)

Terminating the Process Manager

Send a Message to the Process Manager on port DEFAULT_PROCESS_PORT, with type = MESSAGETYPE_COMMAND, and extra = 'Q'. (case-sensitive. 'Q' is for 'Quit'). Processes running under the Process Manager will continue to run, and be restarted when they terminate abnormally.

F.2 Backup Service Interface

The Package and PackageQuery classes are described below. The DataBackup object serves as an easy to use interface between the application and the services described below.

To backup data:

Create a Package with the address of the host requesting the backup, the port of the application requesting the backup, a unique identifier, and the desired data as the payload. Create a string from this package, and set it as the payload of a Message, with the type set to MESSAGETYPE_NOTIFY and extra set to PACKAGE_UPDATE. Send this message to the localhost address on the machine, using port DEFAULT_PROCESS_PORT. (This can be done using Messenger::SendMessageTo(int address, int port, Message* message).

To update backed up data:

Use the same procedure as when initially backing up data. Packages sent to the Process Manager that have the same address/port/id combination as an older package will overwrite the older package.

To remove backed up data:

Create a Package with the same address/port/id combination as the package you wish to delete (any payload of this package will be ignored). Set the id to -1 to remove all packages sent from this address/port combination. Create a string from this package, and set it as the payload of a Message with the type set to MESSAGETYPE_NOTIFY and extra set to PACKAGE_REMOVE. Send this message to the localhost address on the machine, using the port DEFAULT_PROCESS_PORT. (This can be done using Messenger::SendMessageTo(int address, int port, Message* message).

To retreive backed up data:

Create a PackageQuery with the same address/port/id combination as the data you wish to retreive. Setting the id to -1 will retreive all data for a given address/port combination. (The Process Manager will only send data using a query of this type to the address/port from whence it came). Setting the port to -1 will retreive all data for a given address. (The Process Manager will only send data using a query of this type to the DEFAULT_PROCESS_PORT, i.e. other Process Managers.)

The PackageQuery structure provides a reply_address and reply_port field. For a more flexible way of using the Process Manager, it would be possible to use these fields rather than ignore them. They are ignored in the present incarnation of the Process Manager to provide some modicum of security, but there is nothing to stop a program from impersonating another program by running on the same port, and thereby retreive another program's data.

Packages sent to the application will be contained in Messages with type = MESSAGETYPE_NOTIFY and extra = Package::PACKAGE_UPDATE.

F.3 Package

The package is used as a wrapper for the application's buffers, providing an easy way for the application to provide the tags needed by the Process Manager, to back up the data.

PACKAGE_UPDATE = 2011
PACKAGE_REMOVE = 2012
PACKAGE_QUERY = 2013

int address;
int port;
int id;
int length;
char* data;

void SetData(char* buffer, int len)
Sets the payload of the Package as the given buffer, and sets the length of the payload to be the given len value. This function takes the buffer, and does not make a copy.

void Clear()
Deletes the payload of the Package, using the C++ delete operator, and sets the pointer to NULL. It sets the length of the payload to 0.

char* CreateString(int &len)
Using the current parameters of the package (including the payload), make a new byte array. The byte array is allocated using the C++ new operator, and is suitable to pass to the Messenger for sending.

void ParseString(char*)
Takes a byte array, and parses it, assuming the byte array was produced by a Package. Sets all parameters, and allocates a buffer for the payload of the Package.

F.4 PackageQuery

The PackageQuery provides an abbreviated object, which is used when requesting data from the Process Manager.

int address;

int port;

int id;

int reply_address;

int reply_port;

char* CreateString(int &len)

Uses the current parameters and creates a new byte array. The byte array is allocated using the C++ new operator.

void ParseString(char* string)

Takes the given buffer, and parses out all parameters.

F.5 DataBackup

The DataBackup object provides an easy-to-use interface between the application and the backup services of the Process Manager. It communicates with the Process Manager, packages and unpackages data, and makes retrieved data available for pickup. This class encapsulates much the functionality described in F.2.

If using the DataBackup object, the application does not need to worry about the details of the Package and PackageQuery classes. If the application is not using the DataBackup object, then the application needs to use Packages and PackageQueries to store and retrieve data.

DataBackup(int port)

Starts the backup service, running on the given port. The DataBackup class contains a Messenger. It runs a separate thread, to service the Messenger, in between calls to PickUp from the application. The thread continuously gets messages from the Messenger, and puts them on an internal queue.

int Backup(char* buffer, int len)

Takes the given buffer, creates a Package, and sends the Package to the Process Manager. The buffer given to this function will be deleted using the C++ delete operator, so be sure to give it memory allocated with the C++ new operator.

void DeleteAll()

Sends a message to the Process Manager, asking it to remove all of the packages sent from this application.

void Delete(int ID)

Sends a message to the Process Manager, asking it to delete the package given by this ID number.

void RetreiveAll()

Sends a request to the Process Manager, for it to send all of the Packages the application had previously sent it. Packages requested with this function must be retrieved later by using the PickUp() function.

void Retreive(int ID)

Sends a request to the Process Manager, for it to send a copy of the Package given by this ID number. Packages requested with this function must be retrieved later by using the PickUp() function.

char* PickUp()

Returns the payload of a package retrieved using the Retreive() function. Returns only the buffer (which should be a copy of the buffer passed to the DataBackup when calling Backp()). Any information about length, or any tags must be included in the

buffer, since they are not returned by this function. To get all packages sent to the application by the Process Manager, the application should make repeated calls to PickUp(). The function will return NULL when there are no packages in the queue, but other packages might still be forthcoming from the Process Manager.

Appendix G

Data Aggregation API

The Data Aggregation system consists of an Aggregator and a collection of base classes, which provide some basic services. The application must provide derived classes in order for the system to function. For an example of a collection of derived classes, see Appendix H. Below, we describe the default function that are available, and the way these functions ought to function in derived classes.

G.1 Sensor

The Sensor is a purely abstract class. The application must provide a derived class. None of the members described below are implemented – this is only a description of what the function should do in a derived class.

The Sensor is responsible for creating and tagging pieces of data. It is assumed that pieces of data produced in a single iteration will all have different tags. The tags are the way the sensor can say that the same thing represented in two different iterations by different pieces of data. The sensor need not produce data for every phenomena at each time step, but the application programmer should be wary of the interaction between not producing data at each time stamp, and the timeout value

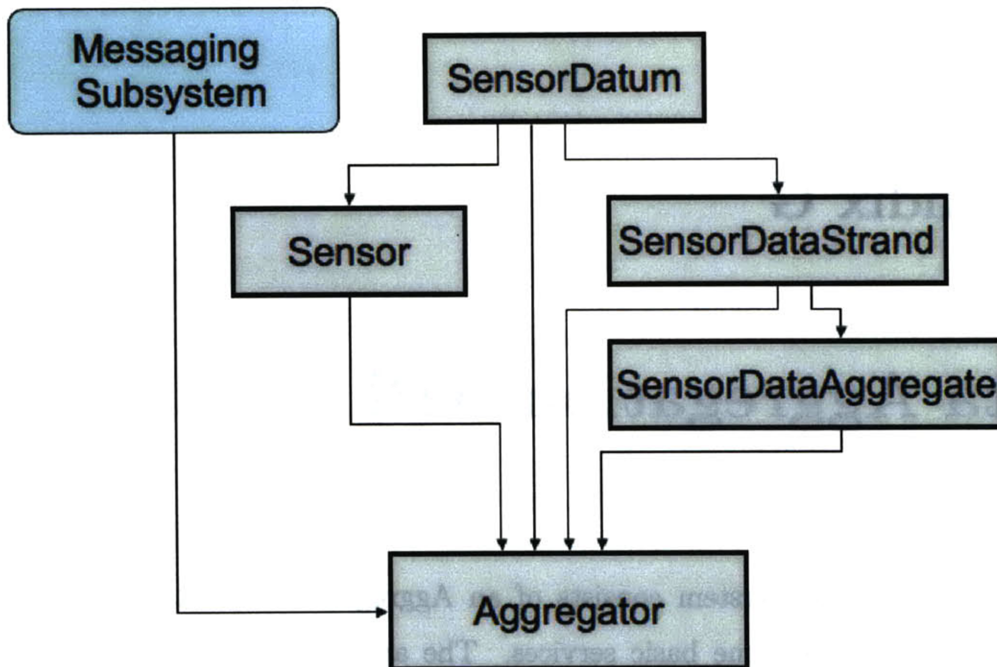


Figure G-1: The dependency diagram for objects in the Data Aggregation subsystem.

given to the Aggregator. The sensor could produce more than one piece of data for each phenomena. This makes the most sense when the different pieces of data have different time stamps.

PointerList <SensorDatum*>* ProduceData()

This function should return a list of the SensorDatum produced at each time step. Each SensorDatum in this list will probably have different tags.

SensorDatum* AllocateSensorDatum()

Allocates a SensorDatum of the derived type.

SensorDataStrand* AllocateSensorDataStrand()

Allocates a SensorDataStrand of the derived type.

SensorDataStrand* AllocateSensorDataStrand(SensorDatum*)
Allocates a SensorDataStrand of the derived type, and makes the given SensorDatum the first element in the strand.

SensorDataAggregate* AllocateSensorDataAggregate()
Allocates a SensorDataAggregate of the derived type.

SensorDataAggregate* AllocateSensorDataAggregate(SensorDataStrand*)
Allocates a SensorDataAggregate of the derived type, and uses the given SensorDataStrand as the first element in the aggregate.

G.2 SensorDatum

The SensorDatum is the main unit of currency in the Data Aggregation system. The SensorDatum is an abstract class, and the application must provide a derived class. Some of the members have default implementations, as described below.

The following variables are members of the SensorDatum. These variables are referred to collectively as the "header" (since they will be used to identify all types of derived SensorDatums).

int timestamp; //seconds from the system clock
int microstamp; //microseconds from the system clock
int duration; //the amount of time elapsed between this, and a subsequent SensorDatum
int source; //originating address
int type; //application-defined
int label; //defined by the sensor
int seq_no; //defined by the sensor

void Initialize()
Sets all variables in the SensorDatum to zero or negative one.

void SetTime()

Sets the timestamp and microstamp of the SensorDatum using the local clock.

void Copy(SensorDatum*)

Copies all variables of the header from the given SensorDatum. Derived classes should provide an alternative implementation that copies all relevant data, and also calls the base class version to copy the header.

char* CreateString(int &len)

This function is abstract and must be provided by the derived class. This function should Create a byte array from the SensorDatum, using all of the current data. Should sets the given len variable to the number of bytes in the returned buffer.

void WriteHeader(char*, int &len)

Writes the header information (the data from the base class), into the given buffer. The len variable is set to point to the integer after the end of the header (assuming that the buffer is an array of integers, instead of an array of bytes). To get to the byte after the header, multiply the value returned in len by sizeof(int).

void ParseString(char*)

This function is abstract and must be provided by the derived class. This function should parse a buffer into the the SensorDatum.

void ReadHeader(char*, int &len)

Parses the header information out of the front of the buffer. The len variable is set to point to the integer after the end of the header (as above). To get to the byte after the header, multiply the value returned in the len variable by sizeof(int).

void Print()

This function is abstract and must be provided by the derived class. Prints the data of the SensorDatum.

void PrintHeader()

Prints the header information of the SensorDatum in the following format:
(timestamp : microstamp : duration) || source || type : label ||seq_no

float Error(SensorDatum*)
This function is abstract and must be provided by the derived class. It is to be the difference between two SensorDatum objects (whatever that difference might mean).

G.3 SensorDataStrand

The SensorDataStrand is used to organize SensorDatum objects by their origin and tags. All SensorDatums in a given strand will have the same source, type, and label. The label variable of the SensorDataStrand is for use by the Aggregator. The labelequivalence variable is used to denote the label of the SensorDatum objects contained in the strand.

int source; //originating address
int type; //application-defined; will match all SensorDatums
int label; //for use by the Aggregator
int labelequivalence; //will match all SensorDatum::label variables
bool used; //denotes whether the strand has had any elements added to it
int totaltime; //amount of time covered by samples
int modification; //the seconds from the system clock, at the last modification time
int micromodification; //the microseconds from the system clock at the last modification time
PointerList<SensorDatum*>*measurements; //The list of SensorDatums contained in the strand
SensorDataAggregate* parent; //A pointer to the strand's parent SensorDataAggregate

The "label" variable is used internally by the Aggregator for identifying strands. The

"labelequivalence" variable is used to keep track of the tags that are assigned to the SensorDatum's produced by the Sensor.

SensorDataStrand()

Sets all variables to initial values.

SensorDataStrand(SensorDatum*)

Sets all variables to initial values, and then adds the given SensorDatum as the first element in the history of the strand, by calling Add(SensorDatum*).

void Initialize()

Sets all variables to initial values.

void Clear()

Removes all elements from the strand's history, maintaining the source, type, and labelequivalence information

void ClearAll()

Removes all elements from the strand's history, and sets "used" to false, priming the strand for a new source/type/labelequivalence labelling.

bool Belong(SensorDatum*)

Compares the tags of the given SensorDatum with its internal tags. (The tags that are compared are the source, the type, and the labelequivalence tag.)

bool Add(SensorDatum*)

If the SensorDatum is the first element to be entered in the history of the strand, the stand adopts the tags of the SensorDatum. (The source, type and labelequivalence variable are set to match the source, type and label variables of a SensorDatum.) Adds it in order by looking at the timestamp. Computes the duration of SensorDatums in the list by looking at the time elapsed between each element and the subsequent element.

bool AddDataPoints(PointerList<SensorDatum*>*)

Takes a list of data points, and adds them, one at a time, using the Add(SensorDatum*) function.

void Trim(int samples)

Trims the history to contain no more than the specified number of samples.

void TrimByDuration(int microseconds)

This function is designed to trim the list so that it only contains samples that happened in the last *microseconds* amount of time. Uses the GetByDuration function to find the SensorDatum that occured after the specified amount of time passed, and then removes all subsequent (less recent) items.

void TrimByTime(int seconds, int microseconds)

Looks for the SensorDatum that has occured after, but closest to the given time. Removes all subsequent SensorDatums.

void Regularize(int microseconds)

First, computes the number of samples that must be in the history, for the samples to be the specified number of microseconds apart. Then, resamples the data using the interpolation function.

void Resample(int samples)

Uses the interpolation function to resample the data, so that the history will have the specified number of evenly spaced samples.

SensorDatum* GetFirst()

Gets the first (least recent) element in the history.

SensorDatum* GetNth(int& n)

Gets the specified element of the history, where the 0th element is the most recent element.

SensorDatum* GetByDuration(int µseconds)

This function is designed to return an element that happend *microseconds* ago. Starting with the most recent element in the list, the function adds together the duration of each subsequent element. The SensorDatum directly after the specified amount of time has been accounted for is returned.

SensorDatum* GetByTime(int &seconds, int& microseconds)

Looks for the SensorDatum that happened directly before the given time, and returns it.

int Length()

Counts and returns the length of the history.

void Print()

Prints all of the SensorDatum in the history, using the SensorDatum::Print() method.

SensorDatum* Interpolate(float place)

The place variable should be between 0 and 1. Least recent element is 0. Most recent element is 1. Finds the given place in the list and creates a new SensorDatum, which is interpolated from the elements in the history. The default interpolation function uses nearest neighbors. This function is a wrapper that calls Interpolate(float place, SensorDatum**before, SensorDatum**after, float &percent) with dummy variables.

SensorDatum* Interpolate(float place, SensorDatum before, SensorDatum** after, float& percent)**

The place variable should be between 0 and 1. Least recent element is 0. Most recent element is 1. Finds the given place in the list, and creates a new SensorDatum, which is interpolated from the elements in the history. Taking the argument of the pointers to the before and after SensorDatum allows the application to get information about the two places in the history surrounding the point of interest. The default interpolation function uses nearest neighbors. This function should be overloaded by the application to do whatever makes sense with the data.

SensorDatum* FindPlace(float place, SensorDatum before, SensorDatum** after, float& percent)**

The place variable should be between 0 and 1. Least recent element is 0. Most recent element is 1. Finds the given place in the list, and returns the SensorDatum before and after that place, along with the percentage between them. If there were 10 evenly spaced elements in the list, and the place variable had a value of .72, the before variable would contain a pointer to the 7th element in the list, the after pointer would contain a pointer to the 8th element in the list, and the percent variable would be set to 0.2.

SensorDatum* AllocateSensorDatum()

This function is abstract and must be provided by the derived class. Allocates a SensorDatum of the derived type.

void Remove(SensorDatum*)

Removes the given SensorDatum from the history of the strand.

G.4 SensorDataAggregate

The SensorDataAggregate is a container to hold a group of related SensorDataStrands. The SensorDataStrands contained in a SensorDataAggregate should represent the same phenomena. All SensorDataStrands in an aggregate will (most likely) be from different sources. The aggregate is able to expel strands from its membership, if it determines that a strand no longer meets the necessary criteria. To aid with this functionality, the aggregate has pointers to two of the paramets in the Aggregator, which are set by the application. The two parameters at the ExpirationTime (the amount of time a strand is allowed to go without being updated, and the MatchScore, which is the maximum match distance, as given by the DataBelongs() function. Pointers are used so that subsequent changes to these parameters by the application will

be reflected in all of the aggregates. Classes derived from the SensorDataAggregate may use these pieces of information, if it wishes.

int* ExpirationTime; //pointer to Aggregator parameter for local processing
int* MatchScore; //pointer to Aggregator parameter for local processing

PointerList<SensorDataStrand*>*feeder; //the member strands
int totalreports; //the number of feeder strands

int modification; //the seconds from the system clock, at the last modification time
int micromodification; //the microseconds from the system clock at the last modification time

SensorDataAggregate()
Sets all variables to appropriate initial values.

SensorDataAggregate(int reports, SensorDataStrand* strand)
Sets all variables to appropriate initial values, and calls Add(strand).

void Initialize()
Sets all variables to appropriate initial values.

void SetParameters(int* expiration, int* matching)
Sets the internal pointers to the expiration time and matching threshold. These numbers are used subsequently for checks of staleness and to be sure that strands continue to meet membership criteria. These pointers are assumed to be pointers to the Aggregator's copy of these parameters.

void Clear()
Calls Clear() on each of the feeder strands.

void ClearAll()
Calls ClearAll() on each of the feeder strands.

void Print()

Prints all of the feeder strands.

float DataBelongs(SensorDataStrand*)

This function is thought of as a distance function, between the aggregate, and the given strand. The base class version of this function returns a constant. Derived classes should provide more meaningful membership functions, for the Aggregator to work well. The lower the score is, the more the strand seems to belong to the Aggregate.

int Add(SensorDataStrand*)

Adds the given strand to the list of feeder strands, and sets its parent pointer to point to the aggregate.

void Update(SensorDataStrand* strand)

Updates the modification time of the aggregate (with the local time), and then checks the feeder strands for staleness. Any stale strands are removed.

void Trim(int samples)

Calls SensorDataStrand::Trim(samples) on each feeder strand.

void TrimByDuration(int microseconds)

Calls SensorDataStrand::TrimByDuration(microseconds) on each feeder strand.

void TrimByTime(int seconds, int microseconds)

Calls SensorDataStrand::TrimByTime(seconds, microseconds) on each feeder strand.

void Regularize(int microseconds)

Calls SensorDataStrand::Regularize(microseconds) on each feeder strand.

void Resample(int samples)

Calls SensorDataStrand::Resample(samples) on each feeder strand.

void Remove(SensorDataStrand*)

Removes the given strand from the aggregates list of feeders. Sets the parent pointer of the strand to NULL.

float AlignmentScore(SensorDatum*, SensorDatum*)
Computes the mean squared error between two sets of SensorDatums (using the SendsoDatum::Error() function).

G.5 Aggregator

The Aggregator is the main point of contact for the application. It manages the production, sending, and receiving of data. The Aggregator takes a pointer to a Sensor, which it periodically calls, in order to produce data. The application must periodically call ProcessData() to allow the application to do its work.

The Aggregator has no public data members, but the application may request access to the list of all strands and aggregates in the system. The application should not modify elements in these lists.

A program that uses the Aggregator might look like the program sketched in figure G.5.

The Aggregator also provides a callback registry, so that the application can respond to events in an event driven way. Callback functions are called inside the ProcessData() call. Some callback functions have versions for SensorDataStrands, and SensorDataAggregates. The signature of the callback determines whether the callback is called with the relevant strand or aggregate.

The available events are:

AGGREGATOR_EVENTTYPE_DATA
A piece of data has been processed. Data may originate locally or remotely. The application may register separate callbacks for the strand or the aggregate, using void RegisterCallback(int event_type, void (*function)(SensorDataStrand*, void*), void*) or

```
Aggregator * aggregator; Sensor* sensor; const in WORKING_PORT = 1234;
void callback(PointerList<SensorDataAggregate*>*aggregates, void* data){
  //Do work
}
  int main(){
  sensor = new DerivedSensor;
  aggregator = new Aggregator(WORKING_PORT, sensor)
  aggregator->SetMatchScore(1234);
  aggregator->SetExpirationTime(1000000); //time is in microseconds
  aggregator->SetTimer(2500);
  aggregator->RegisterCallback(AGGREGATOR_EVENTTYPE_TIMER, & call-
back, NULL);

  while(true){
    aggregator->ProcessData();
    //do work
  }
}
```

Figure G-2: This figure shows the basic structure of a program using the Data Aggregation system. This program uses the callback registry to receive regular updates. Modified from figure 3.3.1.

void RegisterCallback(int event_type, void (*function)(SensorDataAggregate*, void*), void*). The strand callback will be called with the strand where the new piece of data is assigned. The aggregate callback will be called with the aggregate parent of the strand where the new piece of data is assigned. This function is called in the middle of ProcessData().

AGGREGATOR_EVENTTYPE_NEW_STRAND

This event is generated when a new strand has been created. (This would typically represent data arriving from a new source.) There is a strand version and an aggregate version of this callback. If using the strand version, the callback will be called with the new strand itself. If using the aggregate version, the callback will be called with the parent of the strand. This function is called in the middle of ProcessData().

AGGREGATOR_EVENTTYPE_NEW_AGGREGATE

This event is generated when a new aggregate has been created. (This would typi-

cally represent the recognition of a new phenomena, which had not previously been recognized.) There is a strand version and an aggregate version of this callback. If using the strand version, the callback will be called with the strand that triggered the creation of the new aggregate. If using the aggregate version, the callback will be called with the new aggregate itself (but the aggregate will only contain one strand. This function is called in the middle of ProcessData(). If a new aggregate is created due to a new strand, both a NEW_STRAND and a NEW_AGGREGATE event will be generated, and both sets of callbacks will be called.

Note: about the DATA, NEW_STRAND, and NEW_AGGREGATE events. These events are all generated in the same function, at the same time. It is possible that all three sets of callbacks will be called. If that happens, they will be called in this order:
NEW_AGGREGATE, NEW_STRAND, DATA. All of these functions are called after the data has been added to the system, and the strand has been added to the aggregate, etc.

AGGREGATOR_EVENTTYPE_REMOVE_STRAND

This event is generated when a strand needs to be removed. Typically, strands will be removed if they have not been updated sufficiently recently. There is a strand version and an aggregate version of this callback. The strand version will be called with the strand being removed. The aggregate version will be called with the parent of the strand being removed. The callbacks will be called before the strand is actually removed. The removal will happen directly after the callback executes.

AGGREGATOR_EVENTTYPE_REMOVE_AGGREGATE

This event is generated when an aggregate needs to be removed. This will happen if none of the strands in the aggregate have been updated sufficiently recently. It represents that the Aggregator has lost track of the phenomena represented by the aggregate. There is only an aggregate version of this callback, which is called with the aggregate to be removed. The callback will be called before the aggregate is actually

removed. The removal will happen directly after the callback executes.

AGGREGATOR_EVENTTYPE_TIMER

This event is generated at the end of ProcessData if the specified amount of time has elapsed since the function was last called. The callback is called with the list of all SensorDataStrands or SensorDataAggregates. Callbacks for this event are registered with
void RegisterCallback(int event_type, void (*function)(PointerList<SensorDataStrand*>*, void*), void*) and void RegisterCallback(int event_type,
void (*function)(PointerList<SensorDataAggregate*>*,void*), void*). The timing interval for this function is set with void SetTimer(int useconds). ProcessData() will complete one full iteration before calling this callback, so more time than specified may have elapsed.

AGGREGATOR_EVENTTYPE_SAMPLES

This event is generated at the end of ProcessData if the specified number of samples have been added since the function was last called. The callback is called with the list of all SensorDataStrands or SensorDataAggregates. Callbacks for this event are registered with
void RegisterCallback(int event_type, void(*function)(PointerList<SensorDataStrand*>*, void*), void*) and void RegisterCallback(int event_type,
void(*function)(PointerList<SensorDataAggregate*>*,void*), void*). The number of samples to be added is specified with void SetSamples(int samples). ProcessData() will complete one full iteration before calling this callback, so more samples than specified may have been added. This event does not distinguish between data generated locally or remotely.

AGGREGATOR_EVENTTYPE_REQUEST_RECEIVED

This event is generated when another host requests that we send them our data. The callback function is not able to affect whether or not the requesting host receives data. Callbacks for this function are registered by calling void RegisterCallback(int

event_type, void (*function)(Message*, void*), void*). The value given to the callback function will be the message used to request the data. The callback must not modify or delete this message.

AGGREGATOR_EVENTTYPE_LOSTHOST

This event is generated when a host to which the Messenger fails to send data to a host that we are supposed to send data to. This will typically happen if the other host fails. Callbacks for this function are registered by calling void RegisterCallback(int event_type, void (*function)(Message*, void*), void*). The value given to the callback function will be the message used to notify the Aggregator of the failure. The callback must not modify or delete this message.

Aggregator()

Sets all member variables to appropriate initial values.

Aggregator(int port)

Sets all member variables to appropriate initial values, and then calls Aggregator::SetPort(port).

Aggregator(int port, Sensor* sensor)

Sets all member variables to appropriate initial values, and then calls Aggregator::SetPort(port) and Aggregator::SetSensor(sensor).

void SetPort(int port)

Tells the Aggregator to operate on the given port. (The internal Messenger is set up in this call.)

void SetSensor(Sensor*)

Tells the Aggregator to use the given sensor to generate data.

void Request(char* address)

Instructs the Aggregator to make a request for data from the given host.

void Request(int address)

Instructs the Aggregator to make a request for data from the given host.

void Request(Message* message)

Instructs the Aggregator to make a request for data from the given host. The host information is parsed out of the Message

void Request(MachineNode* node)

Instructs the Aggregator to make a request for data from the given host. The given MachineNode is added to the internal table.

void CancelRequest(char* address)

Instructs the Aggregator to cancel any requests for data from the given host.

void CancelRequest(int address)

Instructs the Aggregator to cancel any requests for data from the given host.

void CancelRequest(Message* message)

Instructs the Aggregator to cancel any requests for data from the given host.

void ProcessData()

This is the main function to drive the Aggregator. The application must periodically call this function in order for the Aggregator to generate and exchange data. In this function, the Aggregator calls Sensor::ProduceData. Then, the Aggregator sends each piece of data to all other hosts that have asked to receive data. Then, the Aggregator processes the local data. Then, the Aggregator processes incoming data. The Aggregator then generates appropriate TIMER and SAMPLES events. Finally, the Aggregator checks for and removes stale aggregates and strands.

void RegisterCallback(int event_type, void (*function)(SensorDataStrand*, void*))

This function is a wrapper for

void RegisterCallback(int event_type, void (*function)(SensorDataStrand*, void*), NULL)

void RegisterCallback(int event_type, void (*function)(SensorDataAggregate*, void*))

This function is a wrapper for

void RegisterCallback(int event_type, void (*function)(SensorDataAggregate*, void*), NULL)

void RegisterCallback(int event_type, void (*function)(SensorDataStrand*, void*), void*)

This function registers strand callbacks for the DATA, NEW_STRAND, NEW_AGGREGATE, and REMOVE_STRAND events. You can register a strand callback for the REMOVE_AGGREGATE event, but it will not be called.

void RegisterCallback(int event_type, void (*function)(SensorDataAggregate*, void*), void*)

This function registers aggregate callbacks for the DATA, NEW_STRAND, NEW_AGGREGATE, REMOVE_STRAND, and REMOVE_AGGREGATE events.

void RegisterCallback(int event_type, void (*function)(PointerList<SensorDataStrand*>*, void*))

This function is a wrapper for

void RegisterCallback(int event_type, void (*function)(PointerList<SensorDataStrand*>*, void*), NULL)

void RegisterCallback(int event_type, void (*function)(PointerList<SensorDataAggregate*>*, void*))

This function is a wrapper for

void RegisterCallback(int event_type, void (*function)(PointerList<SensorDataAggregate*>*, void*), NULL)

void RegisterCallback(int event_type, void (*function)(PointerList<SensorDataStrand*>*, void*), void*)

This function is used to register callbacks for the TIMER and SAMPLES events. Call-

backs registered with this function receive the list of all SensorDataStrands. Beware using the void RegisterCallback(int event_type, void (*function)(SensorDataStrand*, void*), void*) function for these events – there will be no warning that you have done anything wrong, and you will get strange behavior.

void RegisterCallback(int event_type,
void (*function)(PointerList<SensorDataAggregate*>*, void*), void*)
This function is used to register callbacks for the TIMER and SAMPLES events. Callbacks registered with this function receive the list of all SensorDataAggregates. Beware using the void RegisterCallback(int event_type, void (*function)(SensorDataAggregate*, void*), void*) function for these events – there will be no warning that you have done anything wrong, and you will get strange behavior.

void RegisterCallback(int event_type, void (*function)(Message*, void*))
This function is a wrapper for void RegisterCallback(int event_type, void (*function)(Message*, void*), NULL). **void RegisterCallback(int event_type, void (*function)(Message*, void*), void*)**
This function is used to register callbacks for the REQUEST_RECEIVED and LOST_HOST events.

void SetExpirationTime(int)
Sets the staleness criteria. Aggregates and strands that are older than the given number of microseconds will be removed.

void SetMatchScore(int)
Sets the match criteria, for associating strands with aggregates. A strand must have a matching score (given by SensorDataAggregate::DataBelongs()) that is no greater than the given value. Scores must be lower than the given value, since they are thought of as distances, and smaller distances are good.

void SetTimer(int useconds)
Sets the notification interval, if the application registers a callback for the AGGRE-

GATOR_EVENTTYPE_TIMER event. Applications will receive notifications that are at least useconds microseconds apart. They may be further apart, but will not be less.

void SetSamples(int samples)

Sets the number of samples that must be received before the application receives a notification, if the application registers a callback for the AGGREGATOR_EVENTTYPE_SAMPLES event. Applications will receive notifications when at least the given number of samples has been received. More samples may have been received, but there will not be less.

SensorDataStrand* GetDataStrands()

Returns the list of SensorDataStrands.

SensorDataAggregate* GetDataAggregates()

Returns the list of SensorDataAggregates.

void Clear()

Calls Clear() on all SensorDataAggregates and SensorDataStrands.

void ClearAll()

Calls ClearAll() on all SensorDataAggregates and SensorDataStrands.

Appendix H

Ball Tracking API

The BallTracking system is an instance of the Data Aggregation system. The Ball Tracking system provides derived classes for the Sensor, SensorDatum, SensorDataStrand, and SensorDataAggregate.

H.1 TrackingMgr

The TrackingMgr inherits from the Sensor class. It finds and tracks brightly colored objects, using the camera on a single machine.

PointerList <SensorDatum*>* ProduceData()
Grabs an image from the camera, does the necessary image processing, and tracks objects in the scene. Returns a list of Ball objects.

SensorDatum* AllocateSensorDatum()
Allocates a Ball object.

SensorDataStrand* AllocateSensorDataStrand()
Allocates a BallTrack object.

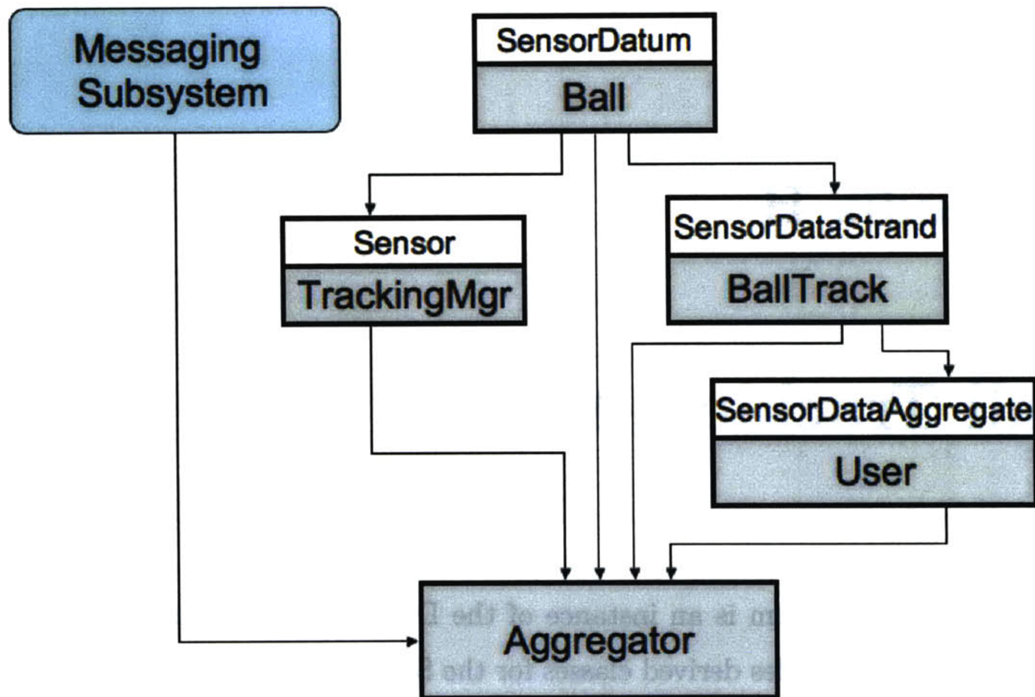


Figure H-1: The dependency (and inheritance) diagram for objects in the Ball Tracking subsystem.

SensorDataStrand* AllocateSensorDataStrand(SensorDatum*)

Allocates a BallTrack object, and sets the given Ball (SensorDatum) as the first item in the history of the BallTrack.

SensorDataAggregate* AllocateSensorDataAggregate()

Allocates a User object.

SensorDataAggregate* AllocateSensorDataAggregate(SensorDataStrand*)

Allocates a User object, and sets the given BallTrack (SensorDataStrand) as the first feeder.

int GetHeight()

Returns the height of the camera's field of view, in pixels.

int GetWidth()

Returns the width of the camera's field of view, in pixels.

H.2 Ball

The Ball class inherits from SensorDatum. It is the fundamental unit in the Ball Tracking system, representing the brightly colored objects found in the camera by the TrackingMgr.

Inherits from SensorDatum:
int timestamp;
int microstamp;
int duration;
int source;
int type;
int label;
int seq_no;

Has, on its own:
int position[3];
int velocity[3];
int YUV[3];

void Initialize()

Sets position, velocity and color information to 0.

void Copy(SensorDatum*)

Copies the header, position, velocity and color from the given Ball (SensorDatum) object. Passing in a SensorDatum that is not a Ball will yield strange results.

char* CreateString(int& len)

Creates a byte array, using the current values. The byte array is allocated using the

C++ new operator and is suitable for use with the Messenger. The variable passed in as len is modified to reflect the number of bytes in the buffer.

void ParseString(char*)

Parses values out of a byte array, assuming that the byte array is the correct format and size.

void Print()

First, prints the position and velocity in the following format: (x, y, z), (v_x, v_y, v_z), (Y, U, V). Then, prints the header on a separate line

float Error(SensorDatum*)

Computes the difference between itself the given Ball. The difference is the squared length of the vector difference between the velocities of the two Balls. The error is given by $(v_{x1} - v_{x2})^2 + (v_{y1} - v_{y2})^2 + (v_{z1} - v_{z2})^2$.

float ColorDifference(int YUV[3])

Computes the magnitude of the squared difference between the color of the Ball, and the given color. The return value is given by $(Y_1 - Y_2)^2 + (U_1 - U_2)^2 + (V_1 - V_2)^2$, where the subscript 1 denotes the internal value of the YUV variable, and subscript 2 denotes the color that was passed in.

H.3 BallTrack

The BallTrack is a container for Balls. It inherits from SensorDataStrand. It makes small modifications to the Add(), Resample(), Regularize() and Interpolate() functions, but otherwise basically uses the default implementations of most functions.

Inherits from SensorDataStrand:
int source;
int type;

int label;

int labelequivalence;

bool used;

int totaltime;

int modification;

int micromodification;

PointerList<SensorDatum*>*measurements;

SensorDataAggregate* parent;

Contains one additional variable:

int YUV[3];

BallTrack()

Calls the default constructor for the SensorDataStrand.

BallTrack(SensorDatum* ball)

Calls the default constructor for the SensorDataStrand, and then calls Add(ball).

void Initialize()

Calls SensorDataStrand::Initialize(), and then sets its YUV variable to 0,0,0.

void Clear()

Removes all elements in the history, except the first (most recent) one.

void ClearAll()

Removes all elements in the history.

bool Belong(SensorDatum*)

Uses the default, SensorDataStrand::Belong() function, which simply compares the tags of the given SensorDatum with the tags that the strand is expecting.

bool Add(SensorDatum* ball)

Sets the YUV variable of the strand to match the given ball, and then calls SensorDataStrand::Add(ball), which adds the item in order by time.

SensorDatum* Interpolate(float place, SensorDatum before, SensorDatum** after, float& prcnt)**
Calls SensorDataStrand::Interpolate(...) to get the before pointer, after pointer, and the percent between them. Does a linear interpolation of the position, between the before and after, according to the percent returned.

void Regularize(int microseconds)
Calls SensorDataStrand::Regularize, which uses the interpolation function to compute the positions of all of the samples. Then, it uses all of the derived positions to compute the velocities.

void Resample(int samples)
Calls SensorDataStrand::Resample, which uses the interpolation function to compute the positions of all of the samples. Then, it uses all of the derived positions to compute the velocities.

H.4 User

The User class represents a person, who has a ball in their hand, where the ball is seen by a number of different cameras. The User class inherits from SensorDataAggregate. It implements a different function for DataBelongs(), and contains two additional functions which are needed by the CursorCoordinator (see Appendix I).

Inherits from SensorDataAggregate:
int* ExpirationTime;
int* MatchScore;

PointerList<SensorDataStrand*>*feeder;
int totalreports;

int modification;

int micromodification;

Contains the additional variable:

int YUV[3];

User()

Creates a new User, using the default constructor.

User(SensorDataStrand* data)

Creates a new User, with the given strand as the first strand of the feeder strands.

void Initialize()

Calls SensorDataAggregate::Initialize(), and then sets the internal YUV variable to 0,0,0.

float DataBelongs(SensorDataStrand*)

First, computes the time difference between the most recent samples in the given sensor data strand, and any other strand from the same host. If the two strands contain samples that were generated at approximately the same time, the function returns a very large (constant) value. Otherwise, returns the difference between the color of the BallTrack, and its internal YUV variable, using the ColorMatch function.

int ColorMatch(int yuv[3])

Computes the squared length of the difference between the internal color, given by the YUV variable, and the color given as an arguemnt. The return value is given by $(Y_1 - Y_2)^2 + (U_1 - U_2)^2 + (V_1 - V_2)^2$, where subscript 1 denotes the internal value, and subscript 2 denotes the argument value.

void Update(SensorDataStrand*)

Updates the internal YUV variable with the color of the most recently added Ball in the SensorDataStrand, and then calls the default SensorDataAggregate::Update() function.

void Regularize(int microseconds)

Calls the Regularize function on each of the feeder strands.

void Resample(int samples)

Calls the Resample function on each of the feeder strands.

char* CreateString(int& len, int address)

Finds the local strand (given by the local IP address), in the User. Using the local strand, this function creates a Message, for sending to other Aggregators, containing the address, labelequivalence, and YUV variables of the local strand. This function is used by the CursorCoordinator.

bool MatchString(char*)

Assumes that the argument was created by the CreateString function. This function parses out the address, labelequivalence and color arguments. First, it determines if it has a feeder that matches the tags (address and labelequivalence) given in the buffer. If it contains a strand with matching tags, the function returns true. If it does not, it then computes the difference between its internal color, and the color given in the buffer. If this score is less than the Aggregator's maximum color distance (given by MatchScore), the function returns true. Otherwise, it returns false. This function is used by the CursorCoordinator.

Appendix I

Cursor Input API

The Cursor Input system uses the Data Aggregation and Ball Tracking systems, in order to provide a unified system for a mouse-like cursor interface.

The CursorCoordinator needs to get information about its neighbors – the ones immediately surrounding it, and the ones with whom it must negotiate for control of the cursor. The cursor system keeps a table indicating whether neighbors are to the left, right, etc. to determine to whom it should pass the cursor, or who it should ask for help. When attempting to negotiate control of the cursor, the CursorCoordinator sends information to all hosts in the Messenger's neighbor table.

The history of Cursors are tokenized into Strokes by looking for stops.

The CursorCoordinator runs on port 3460.

I.1 CursorCoordinator

The CursorCoordinator is the main point of contact for the system, handling the control, generation and destruction of cursors, getting and sending data to other

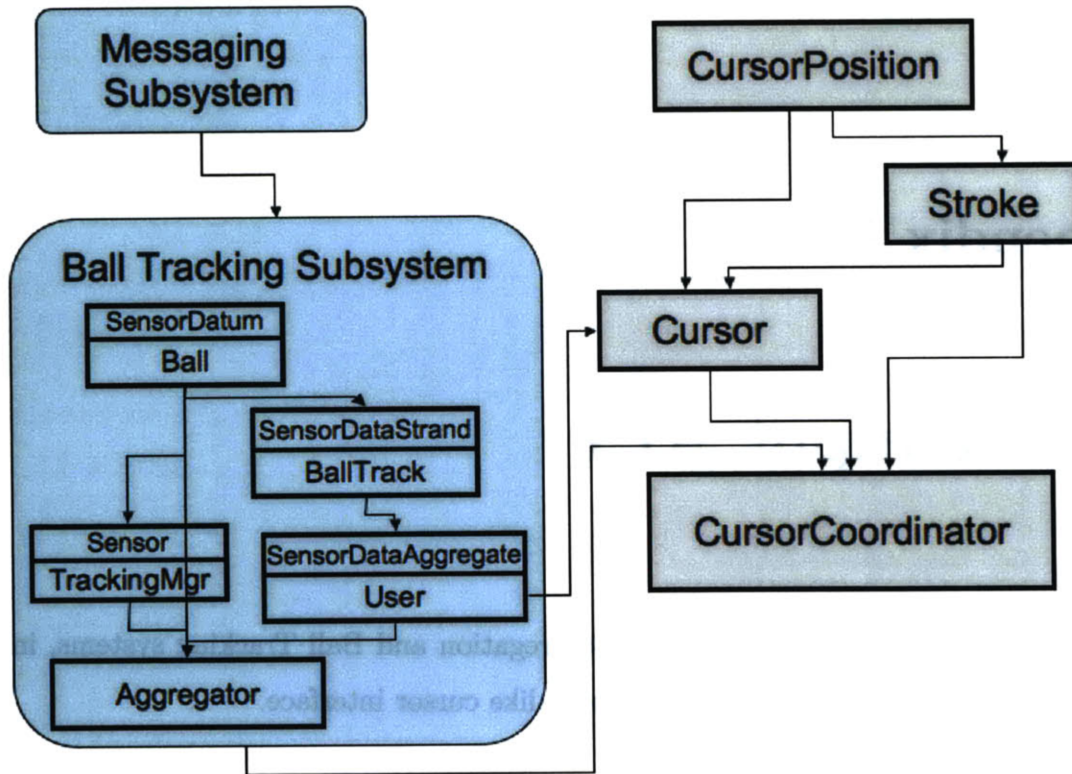


Figure I-1: The dependency diagram for objects in the distributed Cursor Input subsystem.

nodes, and generating events for the application. The application must periodically call HandleCursors() to let the CursorCoordinator do its work.

The CursorCoordinator is comprised of a very large amount of code, but most of its complexity lies in handling the control of the cursors. The interface for the application is not too complicated, but the application programmer must be careful of how they handle events.

The application registers to receive events by calling void RegisterCallback(int type, void (*function)(Cursor*, void*), void*). Events generated by cursors controlled locally and remotely are registered seperately, to reduce confusion when events need to be propagated at the application layer. The available events are:

CURSOR_EVENTTYPE_NEWCURSOR

A new cursor has been created and activated (the node has completed the control negotiation process, and has come up the winner).

CURSOR_EVENTTYPE_REMOVECURSOR

A cursor has been destroyed.

CURSOR_EVENTTYPE_MOVEMENT

The cursor moved

CURSOR_EVENTTYPE_STOPPED

The cursor is presently stopped.

CURSOR_EVENTTYPE_STROKE

The cursor completed a non-gesture stroke.

CURSOR_EVENTTYPE_CIRCLE

The cursor did a circle gesture. Before calling the callback, the stroke that was classified as the gesture is removed from the list of tokens on the cursor.

CURSOR_EVENTTYPE_HORIZONTAL

The cursor did a horizontal shake. Before calling the callback, the stroke that was classified as the gesture is removed from the list of tokens on the cursor.

CURSOR_EVENTTYPE_VERTICAL

The cursor did a vertical shake Before calling the callback, the stroke that was classified as the gesture is removed from the list of tokens on the cursor.

CURSOR_EVENTTYPE_ARRIVED

A cursor that another node is controlling has arrived for the first time.

CURSOR_EVENTTYPE_STATE_CHANGED

The state of a cursor being controlled by another node has changed. This callback is not called for cursors that are under local control. (If we changed the state of the cursor, we already know about it.)

CURSOR_EVENTTYPE_REMOTE_REMOVECURSOR

A cursor being controlled by another node has been removed.

CURSOR_EVENTTYPE_REMOTE_STOPPED

A cursor being controlled by another node is presently stopped.

CURSOR_EVENTTYPE_REMOTE_STROKE

A cursor being controlled by another node completed a non-gesture stroke.

CURSOR_EVENTTYPE_REMOTE_CIRCLE

A cursor being contolled by another node did a circle. The cursor can be used, as if it were local (any data associated with the gesture is removed).

CURSOR_EVENTTYPE_REMOTE_HORIZONTAL

A cursor being contolled by another node did a horizontal shake. The cursor can be used, as if it were local (any data associated with the gesture is removed).

CURSOR_EVENTTYPE_REMOTE_VERTICAL

A cursor being contolled by another node did a vertical shake. The cursor can be used, as if it were local (any data associated with the gesture is removed).

void RegisterCallback(int type, void (*function)(Cursor*, void*))

A wrapper for void RegisterCallback(int type, void (*function)(Cursor*, void*), NULL)

void RegisterCallback(int type, void (*function)(Cursor*, void*), void*)

Registers the given callback for the given event type.

void SendCursorTo(Cursor* cursor, int address)

Instructs the CursorCoordinator to send cursor information to its version of the host with the given address. (It uses the transform that it knows about in its own table.)

void SendCursorTo(Cursor* cursor, MachineNode*)

Instructs the CursorCoordinator to send cursor information to the given host, using the given transformation. This function does not add the given node to the internal

Messenger. So, it is possible to send cursor information to other hosts, without having them entered in the negotiation for control of the cursor.

void StopSendingCursorTo(Cursor* cursor, int address)

Instructs the CursorCoordinator to stop sending cursor information to the given host. The other host will be converted to a stakeholder, and will continue to receive keep-alive messages.

void StopSendingCursorTo(Cursor* cursor, MachineNode*)

Instructs the CursorCoordinator to stop sending cursor information to the given host. The other host will be converted to a stakeholder, and will continue to receive keep-alive messages.

void ChangeState(Cursor*, int state)

Changes the state of the given cursor to the given value. This function propagates the state change out to other hosts that have an interest in the cursor (so, while it is possible to change the state of a cursor directly, it is best to do it with this function.)

void TransferStatus(Cursor* cursor, bool keeplocal)

Gives the CursorCoordinator instructions to not transfer control of a given cursor. This is useful for when the application is busy using the cursor for something, and doesn't want it to disappear. The CursorCoordinator will ask for help from its neighbors, as necessary to keep information coming about the given cursor.

void TransferAuto(int radius)

Gives the CursorCoordinator instructions to automatically designate hosts as receivers or stakeholders, using its best judgement. It will designate a node as a receiver if it detects that a cursor is inside the bounds of the node (the node must be represented in the internal Messenger.)

bool Control(Cursor* cursor)

Returns true if the cursor is under local control, and false if it is under remote control.

PointerList<Cursor*>* GetCursors()return cursors

Returns the list of all active cursors, that should be drawn or reacted to.

void HandleCursors()

The main function to drive the CursorCoordinator. All callbacks are called in the context of this function.

void AddNeighbor(MachineNode*)

Adds the given host to the internal Messenger. Checks to see if the host is directly next to us. If it is, the host is listed in a table of hosts where we should pass cursors or ask for help.

void AddNeighbor(Message*)

Parses information out of the message, and calls AddNeighbor(MachineNode*).

void RemoveNeighbor(MachineNode* host)

Removes the given node from the internal Messenger, and from the list of hosts where data should be sent, etc.

void SetClampBoundaries(int left, int right, int top, int bottom)

Sets the clamping boundaries. If a cursor goes outside of the given bounds, it will be placed back inside. Using all zero values turns clamping off.

I.2 CursorPosition

The CursorPosition is a simple container for information about the position and speed of a cursor at a given time. Cursors and Strokes contain a list of CursorPositions, describing their history.

int source;

int id;

int seq_no;

int speed;

int position[3];

int velocity[3];

int modification;

int micromodification;

char* CreateString(int &len)

Creates a byte array, using the current internal information. The buffer is allocated using the C++ new operator and is suitable for sending with the Messenger.

void ParseString(char* string)

Parses information out of a byte array.

I.3 Cursor

Cursors are the main unit of currency in the Cursor Input system. They correspond (roughly) to the SensorDataAggregates contained in the Aggregator in the Cursor-Coordinator. They are used to summarize the information garnered from the many strands found in each SensorDataAggregate. They contain a list of CursorPosition objects, representing the history of the cursor's movement, and a list of tokens. The history of the cursor is tokenized by looking for stops, and the resulting segments are turned into Strokes. Strokes have removed all of the positions where the cursor did not move appreciably. The tokens are stored in a list. The Cursor has intelligence for updating and tokenizing itself, using the information from a User (SensorDataAggregate), but the application does not need to use that functionality, since it is invoked by the CursorCoordinator.

int source;

int id;

int length;

int position[3];
int velocity[3];

float color[3];

int state;

PointerList<CursorPosition*>history;
PointerList<Stroke*>tokens;

Stroke* GrabToken()

Takes the most recent token from the list, for the application's use. The stroke is removed from the token list.

Stroke* GrabToken(int index)

Grabs the given token in the list, for the application's use. The stroke is removed from the token list.

Stroke* GrabToken(Stroke*)

Grabs the given token from the list, for the application's use. The stroke is removed from the token list.

Stroke* BorrowToken(int index)

Grabds the given token from the list, but does NOT remove the stroke from the token list.

char* CreateString(int &len)

Uses the current internal parameters to create a byte array. The buffer is allocated with the C++ new operator and is suitable for use with the Messenger.

void ParseString(char*)

Parses the relevant parameters out of the given buffer.

I.4 Stroke

The stroke is a series of CursorPosition objects, wrapped up in a different package. The Stroke has intelligence to classify its history as one of 3 gestures, or as a non-gesture. The application does not need to use this intelligence, however, since the CursorCoordinator will identify gestures and notify the application via the callback system. The most interesting thing, for the application, is the stroke variable, which is the list of CursorPosition objects which comprise the stroke.

```
GESTURE_NONE=-1;
GESTURE_CIRCLE=0;
GESTURE_HORIZONTAL=1;
GESTURE_VERTICAL=5;

int source;
int id;
int length;
float color[3];
int position[3];

PointerList<CursorPosition*>* stroke;
```

Appendix J

Paint

The paint application was the primary vehicle for the development of the cursor system. The functionality of a paint program is very familiar : as a user moves the cursor, a trail is drawn behind them. Our paint application allowed users to choose the color of the paint they were using, to decide whether or not to keep the strokes they had created, and to clear the screen.

As users move the cursor, a tail would be drawn, showing where the stroke would be laid down. If user made a circle gesture, the stroke would be laid down in a thick line. If the user shook the cursor horizontally, the cursor would change color. If the user shook the cursor vertically, the screen would be cleared.

Once the cursor system worked properly, the paint program followed fairly simply. The paint program registered callbacks with the CursorCoordinator for the circle, horizontal, and vertical events. Then, the program went through a cycle where it would process incoming messages, handle the cursors, and then redraw the screen.

When a circle gesture occured, the application would take the most recent token from the cursor that made the gesture, and add it to a list of strokes to draw. When a horizontal gesture occured, the program would change the state of the cursor, using

the CursorCoordinator, and the state change would propagate throughout the system. (The state of the cursor corresponded to the color the cursor was drawn in.) When a vertical gesture occured, the program would remove all of the strokes from its list of strokes to draw.

When the paint program was moved from a single tile, to multiple tiles, the program then had to handle remote events. To do this, the program registered a callback for the remote circle event. When a remote circle event happened, the program would take the most recent token from the affected cursor, just as if the event had happened locally. The program did not register events for the remote horizontal and remote vertical events, because these things were propagated at the application level. (The change in color was propagated by using the CursorCoordinator to propagate the cursor state change. The clear screen was propagated by simply sending a message to all participating tiles.)

Bibliography

[1] Gregory D. Abowd. Software engineering issues for ubiquitous computing. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 75–84, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[2] I.F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102 – 114, Aug 2002.

[3] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, September 2003.

[4] Elizabeth M. Belding-Royer. Multi-level hierarchies for scalable ad hoc routing. *Wireless Networks*, 9(5):461–478, September 2003.

[5] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context aware applications. *Human-Computer Interaction*, 16(2, 3, 4):97 – 166, 2001.

[6] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[7] D. Eseryel, R. Ganesan, and G. S. Edmonds. Review of computer-supported collaborative work systems. *Educational Technology & Society*, 5(2), 2002.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[9] Deepak Ganesan, Sylvia Ratnasamy, Hanbiao Wang, and Deborah Estrin. Coping with irregular spatio-temporal sampling in sensor networks. *SIGCOMM Comput. Commun. Rev.*, 34(1):125–130, 2004.

[10] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Pete Steenkiste. Project aura: toward distraction-free pervasive computing. *Pervasive Computing*, 1(2):22–31, 2002.

[11] Jonathan Grudin. Computer-supported cooperative work: History and focus. *Computer*, 27(5):19–26, 1994.

[12] Wendi Rabiner Heinzelman and Joanna Kulik. Adaptive protocols for information dissemination in wireless sensor networks. In *Mobicom*, pages 174 – 185, Seattle, WA, 1999.

[13] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, Aug 2001.

[14] Hiroshi Ishii, Minoru Kobayashi, and Kazuho Arita. Iterative design of seamless collaboration media. *Commun. ACM*, 37(8):83–97, 1994.

[15] B. Johanson, A. Fox, and T Winograd. The interactive workspaces project: experiences with ubiquitous computing rooms. *Pervasive Computing, IEEE*, 1(2):67–74, Apr - Jun 2002.

[16] Purushottam Kulkarni, Deepak Ganesan, and Prashant Shenoy. The case for multi–tier camera sensor networks. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 141–146, New York, NY, USA, 2005. ACM Press.

[17] Kevin L. Mills. Introduction to the electronic symposium on computer-supported cooperative work. *ACM Comput. Surv.*, 31(2):105–115, 1999.

[18] Nelson Minar, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes. Hive: Distributed agents for networking things. In *ASAMA '99: Proceedings*

of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents, page 118, Washington, DC, USA, 1999. IEEE Computer Society.

[19] Elin Ronby Pedersen, Kim McCall, Thomas P. Moran, and Frank G. Halasz. Tivoli: an electronic whiteboard for informal workgroup meetings. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 391–398, New York, NY, USA, 1993. ACM Press.

[20] R. Ramanathan and J. Redi. A brief overview of ad hoc networks: challenges and directions. *Communications Magazine, IEEE*, 40(5):20–22, May 2002.

[21] D. Rogerson. In *Inside DCOM*. Microsoft Press, Redmond, WA, 1997.

[22] S. Vinoski. Corba: integrating diverse applications within distributedheterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, Feb 1997.

[23] Jim Waldo. The jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, 1999.

[24] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, 1993.

[25] Jason Wood, Helen Wright, and Ken Brodlie. Collaborative visualization. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 253–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.