Programming Idioms and Runtime Mechanisms for Distributed Pervasive Computing

A Thesis
Presented to
The Academic Faculty

by

Sameer Adhikari

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
October 2004

Copyright © 2004 by Sameer Adhikari
Programming Idioms and Runtime Mechanisms for Distributed Pervasive Computing

Approved by:

Umakishore Ramachandran, Advisor

Calton Pu

Karsten Schwan

Brian Cooper

Raj Kumar (HP Labs)

Date Approved: 6 October 2004
DEDICATION

To my parents
ACKNOWLEDGEMENTS

This dissertation would have been impossible without the support of a lot of people. I would like to take this opportunity to acknowledge all of them.

The foremost person, is my adviser Umakishore Ramachandran. His advice and assistance have played an immeasurable role in this dissertation. I owe him an irredeemable debt for the opportunities he has given to me over the years. I would also like to acknowledge the other members of the committee. Calton Pu, Karsten Schwan, Brain Cooper, and Raj Kumar, gracefully accepted being a part of the committee. They provided invaluable feedback that helped me in this dissertation.

Next I would like to convey my appreciation for the members of the Ubiquitous Presence group. It was a great experience to work with a group of talented researchers. I would like to acknowledge Arnab Paul, Rajnish Kumar, Bikash Agarwalla, and Sandip Agarwala. They gave freely of their time to discussions that have helped me in my work. Thanks are also due to Hasnain Mandviwala, Junsk Shin, Matthew Wolenetz, and Phillip Hutto.

Finally, I would like to thank all the friends who made my stay here memorable. Poulomi and Manuj for the discussions about life, universe, and everything. Andrew and Vinod for being friends, even though they had to put up with me as their roommate. Gaurav, Arnab, Rajnish, Bikash, and Sandip for all the time they spent hanging out with me. Minaxi for the lively and engaging get-togethers over the course of the years. Last, but not the least, thanks to the Pizza and Beer crowd too for the great company.
# TABLE OF CONTENTS

DEDICATION ................................................................. iii

ACKNOWLEDGEMENTS ......................................................... iv

LIST OF TABLES ............................................................. viii

LIST OF FIGURES ............................................................ ix

SUMMARY ................................................................. xi

CHAPTER 1  INTRODUCTION .................................................. 1
  1.1 Problem Statement ................................................. 2
  1.2 Description of Approach ............................................. 3
    1.2.1 D-Stampede: Handling Temporal Streams .................. 3
    1.2.2 Crest: Reasoning about Events ............................ 5
  1.3 Dissertation Outline .............................................. 6

CHAPTER 2  APPLICATION DOMAINS ....................................... 8
  2.1 Surveillance ..................................................... 9
  2.2 Traffic Management ................................................ 13
  2.3 Mobile Commerce .................................................. 17
  2.4 Summary ............................................................ 19

CHAPTER 3  REQUIREMENTS .................................................. 20
  3.1 Stream Handling .................................................. 21
  3.2 Event Reasoning ................................................... 22
  3.3 Distributed Heterogeneous Components .......................... 23
  3.4 Dynamic Join/Leave ............................................... 23
  3.5 Plumbing .......................................................... 24
  3.6 Dynamic Resource Management ................................... 24
  3.7 Failure Handling ................................................ 25

CHAPTER 4  RELATED WORK .................................................. 26
  4.1 Pervasive Computing Middleware .................................. 26
    4.1.1 Context Toolkit ............................................. 26
CHAPTER 1: INTRODUCTION

1.1 Overview ................................................................................. 7
1.2 Literature Review ................................................................. 9
1.3 Goals .................................................................................. 12
1.4 Architecture.......................................................................... 14

CHAPTER 2: D-STAMPEDE IMPLEMENTATION

2.1 System Structure .................................................................. 16
2.2 Design.................................................................................. 18
2.3 Implementation..................................................................... 21
2.4 Summary............................................................................. 27

CHAPTER 3: CREST IMPLEMENTATION

3.1 System Structure .................................................................. 28
3.2 Design.................................................................................. 30
3.3 Implementation..................................................................... 33
3.4 Summary............................................................................. 39

CHAPTER 4: EVALUATION

4.1 D-Stampe: Handling Temporal Streams............................... 40
4.2 Crest: Reasoning about Time, Location, and Identity.......... 42
4.3 Summary............................................................................. 47

CHAPTER 5: CONCLUSION

5.1 Research Contributions......................................................... 48
5.2 Future Work ........................................................................ 50
5.3 Summary............................................................................. 52

REFERENCES ........................................................................... 54
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Table comparing object-oriented and relational databases</td>
<td>64</td>
</tr>
<tr>
<td>Table 2</td>
<td>Table to store event definitions</td>
<td>66</td>
</tr>
<tr>
<td>Table 3</td>
<td>Table to represent class EVENT</td>
<td>67</td>
</tr>
<tr>
<td>Table 4</td>
<td>Table to represent class TIME</td>
<td>68</td>
</tr>
<tr>
<td>Table 5</td>
<td>Table to represent class LOCATION</td>
<td>68</td>
</tr>
<tr>
<td>Table 6</td>
<td>Table to represent class IDENTITY</td>
<td>68</td>
</tr>
<tr>
<td>Table 7</td>
<td>Alternate table to represent class EVENT</td>
<td>69</td>
</tr>
<tr>
<td>Table 8</td>
<td>Delivered Bandwidth (MBps) as a function of data size and number of clients.</td>
<td>85</td>
</tr>
<tr>
<td>Table 9</td>
<td>Delivered Frame Rate (fps) as a function of data size and number of wireless clients.</td>
<td>87</td>
</tr>
<tr>
<td>Table 10</td>
<td>Delivered Bandwidth (KBps) as a function of data size and number of wireless clients.</td>
<td>87</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1  Octopus Hardware Model ............................................. 4
Figure 2  Computational Model: A dynamic Thread Channel Graph .... 4
Figure 3  Universe Model in Crest .............................................. 5
Figure 4  Reasoning Representation ............................................. 6
Figure 5  Generic Structure of a Distributed Pervasive Application .... 8
Figure 6  Generic Structure of a Distributed Pervasive Application .... 20
Figure 7  Reasoning Example ....................................................... 22
Figure 8  Octopus Hardware Model ............................................. 38
Figure 9  Computational Model: A dynamic Thread Channel Graph .... 39
Figure 10 D-Stampede Implementation ......................................... 46
Figure 11 Registration process for faults ..................................... 49
Figure 12 Notification process for faults ...................................... 50
Figure 13 Universe Model in Crest .............................................. 52
Figure 14 Reasoning Representation ............................................. 54
Figure 15 Participation Protocol: Join .......................................... 56
Figure 16 Participation Protocol: Leave ........................................ 57
Figure 17 Integration of D-Stampede and Crest in an application ....... 71
Figure 18 Generic Structure of a Distributed Pervasive Application .... 72
Figure 19 Architecture of Video Conferencing Application ............... 74
Figure 20 Experiment 1 ............................................................. 77
Figure 21 Experiment 2, Configuration 1 ....................................... 77
Figure 22 Experiment 2, Configuration 2 ....................................... 78
Figure 23 Experiment 2, Configuration 3 ....................................... 78
Figure 24 Intra-Cluster Results (Experiment 1) .............................. 79
Figure 25 C based End Device and Cluster Results (Experiment 2) .... 80
Figure 26 Java based End Device and Cluster Results (Experiment 3) .. 81
Figure 27 Failure Notification Results (Experiment 4) ...................... 82
Figure 28 Single Threaded Version performance ............................ 84
Figure 29 Multi-threaded Version performance .............................. 85
Figure 30  Architecture of Survey Application ............................... 91
Figure 31  Latency of join and leave operations .......................... 94
Figure 32  Bandwidth delivered to the consumer .......................... 95
Figure 33  Frequency distribution of correlate operations .............. 96
Figure 34  Latency of correlate operations ................................. 97
Figure 35  Application bandwidth for 1 and 2 mediators ............... 99
Figure 36  Application bandwidth for 3 and 4 mediators ............... 100
Figure 37  Application bandwidth for 5 and 6 producers ............... 101
Figure 38  Application bandwidth for 7 and 8 producers ............... 102
Figure 39  Surveillance Application ........................................ 107
SUMMARY

The emergence of pervasive computing power and networking infrastructure is enabling new applications. Still, many milestones need to be reached before pervasive computing becomes an integral part of our lives. An important missing piece is the middleware that allows developers to easily create interesting pervasive computing applications. This dissertation explores the middleware needs of distributed pervasive applications.

The main contributions of this thesis are the design, implementation, and evaluation of two systems: D-Stampede and Crest. D-Stampede allows pervasive applications to access live stream data from multiple sources using time as an index. Crest allows applications to organize historical events, and to reason about them using time, location, and identity. Together they meet the important needs of pervasive computing applications.

D-Stampede supports a computational model called the thread-channel graph. The threads map to computing devices ranging from small to high-end processing elements. Channels serve as the conduits among the threads, specifically tuned to handle time-sequenced streaming data. D-Stampede allows the dynamic creation of threads and channels, and for the dynamic establishment (and removal) of the plumbing among them.

The Crest system assumes a universe that consists of participation servers and event stores, supporting a set of applications. Each application consists of distributed software entities working together. The participation server helps the application entities to discover each other for interaction purposes. Application entities can generate events, store them at an event store, and correlate events. The entities can communicate with one another directly, or indirectly through the event store.

We have qualitatively and quantitatively evaluated D-Stampede and Crest. The qualitative aspect refers to the ease of programming afforded by our programming abstractions for pervasive applications. The quantitative aspect measures the cost of the API calls, and the performance of an application pipeline that uses the systems.
CHAPTER 1

INTRODUCTION

A key enabler of pervasive computing is the falling prices of computing devices and networking infrastructure. Computing power is available in diverse forms, from consumer devices (mobile phones, game consoles, etc.) to computing devices (clusters, desktops, laptops, etc.). Wired and wireless networks to connect these computing elements are becoming widespread too. The pervasiveness of computing power and networking infrastructure is in turn enabling new applications. Still, many milestones need to be reached before pervasive computing becomes an integral part of our lives. An important piece is the middleware that allows developers to easily create interesting pervasive computing applications. Consider the following scenario.

A video camera used for boundary surveillance in a high-security compound captures a car parked nearby on successive days. A program determines the duration of the car’s presence at the location to be a week long. The application then finds out that the car has been at various high-security locations over the past month. There are many things the application does next in parallel. One, it notifies guards, who have wireless handheld devices, to check the car. Two, it informs the applications in the other high-security compounds in the region to look out for the car. Three, it checks if this car was observed during some recent incident in the city. If it was, a police application then checks if there is anything suspicious about the car (reported stolen), or its owner (criminal record).

Today this scenario is imaginary, as are other possible applications in the domains of surveillance, traffic management and mobile commerce. These applications, though seemingly different, have some common requirements from the software infrastructure. By definition a pervasive application constantly senses the environment to piece together higher level event information. Thus the application works with a large number of data streams; access and process items from a stream, combine items from different streams, create streams.
Components of the application are physically distributed, with all the attendant needs of distributed programming. The components are also distributed over time; thus application level decisions are influenced by live data as well as historical data. The components are temporally dynamic in that the participating entities change constantly over time or an entity may participate in discrete intervals rather than a continuous interval. The components are spatially dynamic in that the participating entities may be mobile or they may change their behavior based on location. The application components that are spread over time and location may have widely heterogeneous computation and communication capabilities. Faults might arise in individual components. The ability of the application to tolerate and react to such faults will depend on the time and location of the fault. Fundamentally, these applications need to continually sense the environment for event information, and then reason about the events with respect to time, location, and identity in an integrated manner to control application behavior. Time, location, and identity refer, respectively, to the when, where, and who of events that drive the application behavior.

1.1 Problem Statement

At the boundary of real world and pervasive-application software are sensors that continually monitor the environment and generate live stream data. From the stream data an application determines the events that occur in the environment. The application reasons about the events to determine the actions that govern its behavior. To reason about the events, the application needs to relate the current events with historical ones based on some criteria. The most important criteria for an event are time (when), location (where), and identity (who). Thus, two fundamental issues that pervasive applications need to tackle are:

- Ability to access continuous live stream data from multiple sources using time as an index

- Ability to organize historical data to reason about events, using time, location, and identity
1.2 Description of Approach

We have explored the domains of surveillance, traffic management, and mobile commerce. The applications in these domains are candidates for distributed pervasive computing, but there is work to be done. Currently, the applications are not truly distributed, and they handle stream data from sensors in an ad-hoc manner. Individual applications reason about events in a limited and specific way; there is a lack of a generic capability that is available for all of them. The exploration has led us to the conclusion that applications in the domains of surveillance, traffic management, and mobile commerce, have common needs that can be met by a middleware. The most important requirements for the middleware are the ability to handle temporal streams, and the ability to organize historical data to reason about events. We have looked at work in other areas that can be used to build pervasive applications. The areas we have looked at are the following: 1) other software infrastructures for pervasive computing, 2) data mining, 3) general distributed programming systems, 4) location technologies, services and applications, and 5) some overarching pervasive computing projects. We have found that systems in the related areas do not support the requirements that we target: temporal streams and reasoning about events.

We have developed programming idioms and runtime mechanisms for pervasive applications. The D-Stampede system allows applications to handle live stream data. The Crest system allows applications to organize historical data and reason about events based on time, location, and identity.

1.2.1 D-Stampede: Handling Temporal Streams

The D-Stampede system helps pervasive applications to handle temporal streams. The hardware model assumed by D-Stampede is the Octopus shown in Figure 1. Different kinds of sensors (and data aggregators located near them) collect raw data and perhaps do limited processing such as filtering. However, extraction of higher order information content from such raw data requires significantly more processing power. For example, microphones may collect audio data, but higher order processing is needed for voice recognition. Thus there is a continuum of computation and communication resources that a pervasive computing
application spans.

Figure 2: Computational Model : A dynamic Thread Channel Graph

The computational model supported by D-Stampe is pictorially presented by the thread-channel graph in Figure 2. The model captures a huge class of inherently distributed applications that are emerging in the context of pervasive computing. The threads map to computing devices ranging from small to high-end processing elements scattered in the environment. Channels serve as the application level conduits among the threads, specifically tuned to handle time-sequenced streaming data. D-Stampe provides APIs for the dynamic creation of threads and channels, and for the dynamic establishment (and
removal) of the plumbing among them.

1.2.2 Crest: Reasoning about Events

![Diagram of Participation Server and Event Store]

**Figure 3:** Universe Model in Crest

The Crest systems allows applications to reason about events using time, space, and identity. The Crest system assumes a model of the universe shown in Figure 3. It consists of a set of participation servers and event stores, supporting a set of applications. Each application consists of a set of software entities working together. For example, in Figure 3 entities 1, 3, 4, 8, 9, and 10 participate in application 251. An entity can participate in more than one application (e.g., in Figure 3 entity 4 is a participant in three applications). The participation server helps a set of entities working together in an application to discover each other for the purposes of communication. Application entities can generate events, store them at an event store, and correlate events. The entities can communicate with each other directly, or indirectly through the event store.

Crest provides operations to correlate events across location, time and identity. Figure 4 pictorially presents the correlate operations. An event definition in an application can be thought of as a three-dimensional space. (Each event instance is a point in the space.) Each of the event attributes (time, location, and identity) is an axis in the space. Each axis is a line only conceptually. Two factors prevent it from being a real line. One, a point on the axis can have multiple attributes (location might consist of three coordinates). Two,
each axis is a collection of discrete values (unlike real numbers that are continuous). A correlate operation takes values in two event dimensions as arguments, and returns values in the third event dimension. The correlate operation works as follows:

- Determines the planar area defined by the values in the two input dimensions.

- Determines the cluster of points in three-dimensional space that projects on this planar area.

- If there is a projection, return values in the third dimension from the cluster as a result.

The correlate operations help an application to reason about events using time, location, and identity.

1.3 Dissertation Outline

The outline of this dissertation is as follows. In Chapter 2 we survey the domains of surveillance, mobile commerce and traffic management that motivate our work. Chapter 3 lists the requirements derived from the application domains in Chapter 2 for a middleware that supports pervasive applications. In Chapter 4 we present work that relates to the different aspects of our work. In Chapter 5 we summarize how the earlier chapters (Chapter
2, Chapter 3, and Chapter 4) help build the case for the intellectual contributions of this thesis. In Chapter 6 we present our first intellectual contribution, the architecture and implementation of D-Stampede system that allows applications to communicate stream data. In Chapter 7 we present our second intellectual contribution, the architecture and implementation of Crest system that allows applications to reason about events based on time, location, and identity. In Chapter 8 we show how distributed pervasive computing applications can use D-Stampede and Crest in an integrated manner. Chapter 9 presents an evaluation of D-Stampede and Crest, both qualitatively and quantitatively. The qualitative aspect refers to the ease of programming afforded by our programming abstractions for applications in our target domains. The quantitative aspect refers to measuring the cost of the API calls, and the performance of an application pipeline that uses the systems. We conclude the dissertation in Chapter 10 by listing our contributions and directions for future work.
CHAPTER 2

APPLICATION DOMAINS

In this section we discuss applications that belong to the following domains: surveillance, traffic management, and m-commerce. The applications present the need for programming systems that allow them to handle temporal streams, and to reason about events using time, location, and identity. For surveillance and traffic management we show projects that undertake the stream handling and reasoning tasks in a project-specific and an ad-hoc manner. This specific and ad-hoc usage across projects implies the need for a middleware that is usable across a broad spectrum of applications. For m-commerce domain we do not present examples of stream handling and reasoning. Rather, m-commerce shows the potential for reasoning about events as the devices used in m-commerce, mostly the cell phones, provide the perfect mechanism to obtain event information.

Figure 5: Generic Structure of a Distributed Pervasive Application

Figure 5 shows the generic structure of a distributed pervasive computing application. An application consists of input, processor, and output entities distributed in a an environment. The entities are connected to one another in application-specific topology and there is a continuous input-process-output control loop. The input entities control the sensors
that constantly monitor the environment, or take directions from operators about the tasks

to perform. The constant monitoring generates temporal data streams such as audio, video,
temperature, ambient light, and others. These streams are fed to processor entities that
operate on them to obtain event information. Some example events are a person entering
or exiting a room; the stream from a camera is analyzed to detect the entrance or exit. An-
other class of events is traffic incidents in a metro area; the streams from multiple cameras
are analyzed to detect congestions, stalls or accidents. The difference between analyzing
the stream from a room and multiple streams from a metro area, is the scale of process-
ing. The latter case requires analysis at computational perception speeds due to the large
volume of input data, and the fact that there is no human in the loop. These events are
communicated among the processor entities that reason about them to determine applica-
tion behavior. The output entities consist of output devices such as screens and speakers,
and effectors such as thermostats, and traffic lights. Based on the event reasoning done
by processor entities, the output entities take some action. For example, if the processors
determine that a person has entered a dark room, the output entities turn on the lights.
Similarly, if a traffic system detects a vehicle stall, it can page a response team that can
assist the motorist, and it can change some road signs to inform commuters about alternate
routes. The reasoning is not restricted to only using current events, but can also relate the
past to the present. For example, an application can determine how many times in the past
week has a certain car been parked at a given parking lot. If there is a suspicious pattern
in the parking behavior, the application can raise an alarm.

2.1 Surveillance

Various projects have looked at automating different aspects of surveillance. In LOTS ([21]),
the authors present an omni-directional tracking algorithm that tracks multiple objects in
an image from an omni-directional camera. The paracamera is used to obtain the image and
then a tracking algorithm works directly with the geometry of the paraimage. The system
provides windows for human users to view perspective-corrected forms of the paraimages.
Tracking is a prime example of reasoning about events based on time, location, and identity.
The process involves identifying objects in an environment and following their movement through location over time. The LOTS system tracks camouflaged and slowly moving soldiers over a terrain, that is, it concentrates on a specific tracking instance. In terms of the generic application (Figure 5) LOTS has a single processor entity that is directly connected to the input entities (cameras), and an output entity (display). The processor entity implements a specific tracking algorithm.

In MPI-Video ([22]) the system allows a user to get data from disparate sources and interact with the data. An environment model (EM) represents the dynamic state of the environment, and helps applications to structure algorithms that determine the environment state. EM consists of layers organized by the level of abstraction; the bottom layer the least abstract, and the top layer the most abstract. Each layer represents an object that interacts with objects in the other layers. In terms of the generic application (Figure 5) MPI-Video provides a way to organize the algorithms in a processor entity. It does not address the issues related to the input and output entities. Real-world applications the environment model will be part of an application that will need to handle temporal stream data, and reason about events using time, location, and identity.

Davis et al. ([28]) present work on surveillance of human activity: detecting independently moving objects from a moving ground camera, a system to track multiple moving people using sequences taken from a stationary camera, and classification of recovered motion into various activity classes. They concentrate on the image processing issues involved in detecting, tracking and classifying activity. With respect to the generic application (Figure 5) their work uses a single input entity (camera), a single processing entity, and a single output entity (display). Their system can generate input data for an application that uses a reasoning system to process the data and drive application behavior.

CyberARIES ([31]) is an agent-based system for collaborative, distributed surveillance. Independent agents are in charge of individual sensor systems, and the agents collaborate to reach a common decision about the environment. CyberARIES provides a framework for coordinated sensing, processing, and communication among the agents. An agent is defined as a software entity that can accept stimuli from other agents, maintain steady-state
behavior in the absence of stimuli, and can provide stimuli to other agents. A distribution layer is responsible for the stimuli communication between agents, and regulating the stimuli flow in the system. A distribution layer sets up the connections among the agents that are necessary to communicate the stimuli. For flow control, the distribution layer asks a sending agent to decrease its transmission rate when it notices a disconnect between the arrival rate and processing rate at the receiving agent. CyberARIES maps to the generic application (Figure 5) with agents playing the role of input, processor, and output entities. CyberARIES does not define the processing that agents perform on the stimuli. A system that the agents can use to reason about event using time, location, and identity will help them process the stimuli.

Ivanov et al. ([44]) present a system for surveillance of human-car interactions. The system consists of three components: tracker, event generator and parser. The tracker processes camera images, identifies objects in the view, tracks them, and collects object movement data into partial tracks. The event generator generates discrete events from the tracks based on an environment map that encodes the knowledge about the environment. These events help determine when an object entered and exited a mapped environment. The parser analyzes the events according to a grammar that describes possible activities. The tracker plays the role of an input entity (Figure 5), and the event generator and parser act as processor entities. The human-car system reasons about events in a limited sense; it deals with the present time, in a specific location, and a minimum notion of identity. To build a complete application requires an ability to archive the events, so that the parser can relate present ones to those from the past. An ability to reason about events using time, location, and identity will help make the parser more powerful. Further, a system to handle temporal streams will allow the trackers to gather data from multiple cameras.

The VSAM project ([64]) presents a multi-camera surveillance system. It detects people and vehicles, tracks them using cooperative sensors, determines their three-dimensional locations using a geospatial model, and presents the information to an operator using a graphical interface. The sensors are modular units that can be added or removed without affecting other sensors in the network. Each sensor performs real-time processing of video
streams to transform them into symbolic representation of objects and events. The data from disparate sensors is fused within a three-dimensional scene coordinate system to obtain a complete representation of the environment. Every camera observation is mapped to a three-dimensional geodetic coordinate: latitude, longitude, and elevation. In terms of the generic application (Figure 5), the cameras map to input entities, and the process that determines the location maps to a processor entity, and the display maps to an output entity. Currently the VSAM project provides no support for an application to make inferences about the surveyed objects. A full-fledged application can use VSAM along with a system to handle temporal streams, and a system to archive and reason about events. A temporal stream system can provide support for handling input from the cameras. VSAM can provide the representation for objects in the physical space. A system to archive and reason about the events will allow an application to relate past and present, and to draw inferences about the surveyed objects.

Ng, et al. ([57]) present a system that uses a large number of omni-directional cameras to survey an area. Their system has two mains goals: robust tracking and profiling of human activities, and dynamic synthesis of virtual views for observing the environment from arbitrary vantage points. The tracker uses multiple Omni-Directional Vision Sensors (ODVS) that populate the environment. The tracker detects people, measures azimuth angles, and triangulates to localize people in the scene. The views that the system synthesizes are of two types: views that follow a person, and views that are seen from the walking person’s perspective. Ellis and Black ([32]) present a system that uses multiple cameras to track objects. The data streams are sent to a server that temporally aligns the streams to compensate for the different processing rates, and integrates data from multiple streams to attain a common view. The cameras in both projects (Ng, Ellis) act as the input entities (Figure 5), the tracker in Ng and the server in Ellis as the processor entity, and the display as the output entity. Like other automated surveillance systems, these project focus on the image processing aspects of the problem. The camera images are directly passed to specific processor entities. The reasoning is limited to a specific location and time, with no real identity aspect. To build a complete application will require systems to handle temporal
streams, and to archive and reason about events. A temporal stream system will provide
abstractions for dynamically shipping camera data to arbitrary processor entities during
extension. An event archival and reasoning system will assist an application in inferring
behavior patterns from event observances.

BioSTORM ([24]) is a system to quickly detect an epidemic resulting from a bio-
terrorism agent. The surveillance performed by BioSTORM is different from that of the
systems we have just considered. The system takes diagnostic data from various sources
(hospitals, clinics), checks the data to ascertain the outbreak of a disease, then analyzes
the data to detect an epidemic. The analysis involves checking for statistical patterns in
the data that point to significant aberrations in the time-space pattern of the diseases. The
focus of the work is building a knowledge base about the normal spatial-temporal patterns
of diseases, and formatting the input data for statistical analysis. The project does not
deal with the input and output issues that will map it to the generic application (Figure 5).
BioSTORM performs a specific case of reasoning about events, based on time, location, and
identity. It does not provide a middleware that can be used by applications in a spectrum
of domains.

All the projects listed above handle continuous input streams from cameras (except
BioSTORM) without high-level abstractions to support the usage. Further, they all perform
a limited form of reasoning about event on an individual level. None of them (except
CyberARIES) is a distributed system, as they connect all the input and an output to
a single processor. Thus, to enable truly distributed and pervasive applications such as
homeland security ([69, 46]), there is a need to handle temporal stream data, and to archive
and reason about events using time, location, and identity.

2.2 Traffic Management

The typical infrastructure for traffic control in a metro area can be described as follows.
There are detectors on major roads that record various measures such as speed (of individual
vehicles) and flow (number of vehicles in a unit time). The sensors are placed at fixed
intervals on the roadways, and data arrive periodically at the control center. The center
also receives the current state of control devices such as signals at intersections, traffic signals at on-ramps, message signs, radio broadcast systems, and reversible lanes. In the control center, operators interpret the information to determine the state of the traffic. They try to detect problems, figure out their possible causes, and take actions to solve or reduce the severity of the problems (congestion, accidents, stalled vehicles). Some actions they can take are change the timing of the signals, or change the signs to inform the people in the vehicles. The mapping of the typical traffic-control infrastructure to the generic application (Figure 5) is as follows: the sensors are the input entities, the human operators are the processor entities, and the signals and signs are the output entities.

The current systems with their dependence on human operators, inherently limits the number of roads under observation. First, only the important roads are monitored resulting in problems on unmonitored roads not getting detected for long times. Second, most of the solutions human operators devise are local in nature. Third, during non-peak hours such as late night or wee hours of the morning, the number of operators present is minimal. Some projects have tried to automate the various aspects of traffic management to assist the operators. The goal in the automation projects is to replace the humans with programs as processor entities.

In the Traffic Sign Classifier (TSC, [61]) an in-car system to recognize traffic signs was developed. A traffic sign is defined by its color, form and inclusion of certain forms and colors. The TSC matches the characteristics of a candidate object with a feature space consisting of characteristics of all the signs, to determine whether the object is a sign, and the identity of a sign (if it is one). A fuzzy control system does a hierarchical graph traversal to reach the sign decision; it takes into account the probabilities of individual decisions when reaching the final decision. The camera reading the sign is the input entity, the classifier is the processor entity, and the output entities are undefined (Figure 5). Here the system does a very limited form of reasoning: determining the identity of an observed object without taking into account either location or time. A reasoning middleware can allow an integrated car system to match the car speed with a speed limit sign, and ask the driver to slow down if he is going too fast.
In ESCORT ([67]) a platform to integrate devices that provide traffic control at an intersection was developed. The platform called Abstract Model of Intersection (AMI) consists of four packages: static, logical, management, and control. The static package contains a topological model of the intersection (lanes, gates, etc.), and does not change during program execution. The logical package contains representation of dynamic information where state can be observed (e.g. traffic flow) or controlled (e.g. signal timing). Management package represents the physical devices whose functions are represented by the logical package. The control package deals with high level traffic management policies. The goal in ESCORT is to provide a framework for building intersection control applications. The management package refers to the input entities, and the other packages (static, logical, and control) to the processor entities (Figure 5). The main focus in ESCORT is on how to represent the information related to traffic control at the intersection. The mechanics of how to control the intersection is left to the applications. The processor entities can reason about time, location, and identity information to do the control.

In Traffic Scene Analysis ([48]) the system detects stalls in video images of the traffic on a highway. There are two components to the system: machine vision, symbolic reasoning. The machine vision component uses a tracker and motion model to extract vehicle trajectories over a sequence of traffic images. The symbolic reasoning component uses a dynamic belief network to make inferences about traffic events (lane changes, stalls). Here the tracker is the input entity, and the reasoning component is the processor entity, and a display is the output entity (Figure 5). A reasoning middleware will help make a more general system than the specific instance currently existing. A system that archives events and allows application to reason about them using time, location, and identity, can provide the infrastructure to build multiple belief networks. Further, a system to handle temporal streams can help an application to manage a many-to-many relationship among producers and consumers of camera streams.

Naumann and Rasche ([56]) present an approach to prevent collisions at an intersection. When entering an intersection a vehicle receives intersection geometry and management information from a beacon. Using its velocity (speed and direction) the vehicle calculates
its path through the intersection. Based on the path, it obtains individual tokens to pass through various parts of the intersection. The tokens regulate the flow of traffic through the intersection, and help to avoid collisions. The work deals only with the algorithm for collision-avoidance that can map to a processor entity in Figure 5, and not with creating a real application. In building a real application for multiple intersections, a system that allows the application to handle the stream data about traffic, and a reasoning system that helps the application to reason about the data will help.

In ALLONS-D ([60]) the authors give a scheme that uses control theory optimizations to minimize the total delay experienced by traffic at each intersection in a network. The scheme presents a model to calculate the delay experienced by each vehicle at an intersection. Using the model, the algorithm then tries to minimize the delay of all the cars at the intersection and comes up with the signal plan for the intersection. As in the case of Naumann and Rasche ([56]), the algorithm will map to a processor entity in Figure 5. Building a real application for multiple intersection, will require a system to handle stream data from input entities, and a system to reason about events using time, location, and identity to control output entities.

Monitorix ([12]) presents a traffic surveillance system that integrates video analysis and agent technology to detect abnormal events in traffic. Monitorix consists of four tiers: sensors and effectors, objective description, application assistant, user assistant. The sensors and effectors tier represents the software that handles the sensor and effector hardware. It receives sensor output and provides control information to the effectors. The objective description layer uses sensor information to generate higher-level semantic information. The information relates to an environment state and application objective. The application assistant layer analyzes the information it receives from sensors and effectors tier, and the objective description tier. The analysis is application-specific and helps drive the behavior. The user assistant tier generates profiles that can be used to study application behavior. The system provides mechanisms for the application to add or remove agents, and for agents in different tiers to communicate. The sensors and effectors tier maps to the input and output entities respectively in Figure 5. The other tiers, apart from the sensors and effectors tier,
map to processor tiers. The authors have implemented agents that reason about specific
events but there is no framework for the agents to reason about event information. Further,
there is no framework to handle the stream data that is generated by the sensors. Each
application independently decides how to communicate the information.

Molina et al. ([54]) present a decision-support system for traffic control. It helps opera-
tors in detecting traffic problems and in choosing appropriate control actions. The system
provides a methodology for building a knowledge model for the domain. Then it uses the
problem-solving method (from AI) to answer questions about the domain. Their focus is
on what would be the processor entities in Figure 5, and they do not look at building a
real application. The decision-support system is a specific project, whose concepts can be
used by other projects. A middleware that helps reason about events will make it easier to
apply the concepts to other systems.

As with the surveillance projects, none of the traffic management projects are distributed
systems. In fact most of the traffic management projects just concentrate on the algorithms
needed to operate on the traffic data. They completely ignore the issues in building real
traffic-management applications. A complete automated system has multiple benefits but
it will require an integration of many of the above-mentioned projects. Thus, building
complete applications will require systems to handle the stream data generated by the
sensors, and to reason about traffic events using time, location, and identity.

2.3 Mobile Commerce

Mobile commerce (m-commerce) has many definitions. Tsilagatidou et al. ([74]) define m-
commerce as “any type of transaction of an economic value that is conducted through a
mobile terminal that uses a wireless telecommunications network for communication with
the e-commerce infrastructure.” A more general definition given by mobileinfo.com [53]
is “any electronic transaction or information interaction conducted using a mobile device
and mobile networks (wireless or switched public network) that leads to transfer of real or
perceived value in exchange for information, services or goods.”

We first look at some existing m-commerce systems, and then show how the ability
to reason about time, location, and identity will benefit these systems. M-commerce is a suitable domain for reasoning about events using time, location, and identity. The reason is that the devices in use (e.g. cellphones, vending machines) most readily provide time, location, and identity information for a transaction event. The cellphones map to the input and output entities in Figure 5. Currently, to the best of our knowledge, no projects exist that have components similar to the processor entities. By processor entities we mean high performance computational resources that can perform heavy-duty tasks. These entities are distributed and together perform high-level data analysis. An example of the analysis is drawing inferences of consumer behavior from the transaction data.

The most successful m-commerce systems in the world is probably I-Mode [4], a wireless Internet service over cellular phones provided by NTT DoCoMo in Japan. It has attracted more than 36 million subscribers since its introduction in February 1999, and provides access to more than 62,000 specially formatted sites. It provides always-on connection and people use it for various activities (email, shopping, banking, entertainment, etc.). Users are charged based on the data they download, and not on the time spent on the download. Some other examples of m-commerce systems using the cell phone are: soda vending machines in Finland [2] and Slovenia [10], parking in Estonia and Norway [6], text messaging (SMS), and downloading ring tones. Wireless LANs [11], and RFID technology [9] provide more avenues for m-commerce.

Now let us consider the possibilities when we can reason about events based on time, location, and identity. For an individual consumer we can track the time and location of his shopping, and send him relevant offers. For example, in the parking application we can determine the usage times and and offer him a personalized parking plan. Based on the purchasing behavior of a group, a merchant can provide a service that helps people with common interests to interact. For the vending machines, a much more accurate stocking schedule can be determined, that will assist in inventory management. The vending machines or parking lots will constantly generate a stream of data about user behavior. A system to handle the temporal stream data will assist application developers in handling the input. The input can be transmitted to different processor entities that analyze the
data. Further, an ability to archive events and reason about them using time, location, and identity in a generic manner will help business application developers to analyze the data, and hence user behavior. The analysis may lead to special offers, promotions, and deals with the objective of increasing profitability.

2.4 Summary

After surveying the domains of surveillance, traffic management, and m-commerce we see the following:

- The applications are not really distributed and pervasive.

- Individual applications handle their live stream data in an ad-hoc manner.

- Applications do not organize historical data, and at best perform limited reasoning on current data

- The reasoning about information is application-specific, and not usable by other applications.

Thus, to build pervasive distributed computing application in surveillance, traffic management, and m-commerce requires the following:

- A system to help applications handle the temporal stream data generated by the environment.

- A system to reason about events using time, location, and identity so that applications can analyze the data streams and take some action.
CHAPTER 3

REQUIREMENTS

In this chapter we describe some of the requirements for a middleware that supports distributed pervasive computing applications. The requirements are derived from the survey of applications from the domains of surveillance, traffic management, and m-commerce in Chapter 2. The generic structure of distributed pervasive applications from the previous chapter (Figure 5) is shown again in Figure 6. To enable the applications represented by Figure 6 requires the following: stream handling, event reasoning, distributed heterogeneous components, dynamic join/leave, plumbing, dynamic resource management, and failure handling. The list we present is not a complete one; it just contains the most fundamental requirements.

![Generic Structure of a Distributed Pervasive Application](image)

**Figure 6:** Generic Structure of a Distributed Pervasive Application

The input entities constantly generate temporal streams using sensors that monitor the environment. The processor entities need to analyze the data streams to figure out the events, and then reason about the events to govern application behavior. The application entities are distributed over an environment and they need to be able to form and maintain dynamic interactions (join/leave, plumbing). The entities can be heterogeneous in terms of
the hardware and software capabilities. As the application entities are dynamic so are their resource demands. Further, entities can fail for various reasons, so the application needs to handle failures.

3.1 Stream Handling

Today video and audio sensors are common. Consider gesture and speech as inputs for a pervasive application. A gesture is a sequence of images, and speech is a sequence of audio samples. The import of a word would depend on the associated gesture. Each stream is a sequence of some basic basic element (e.g. a frame of pixels).

As the cost of hardware drops, other sensors would become widespread. Consider a house as the target environment. Each room will have sensors to continuously monitor light and temperature. These data streams would be fed to a computational device, that would try to maintain the light and temperature at a user-defined level. There may be sensors, that monitor the health of the people in the room. They would feed an analyzer that would raise an alarm, if it detects an anomaly in the data stream. There may be motion detectors in the yard. The system would raise an alarm, if everyone in a house is sleeping, and movement is detected in the data stream. As sensor based computing becomes more prevalent we expect there to be a preponderance of stream data. Thus there needs to be efficient support for streams.

Streams from different sources may need to be combined, correlating them temporally. For example, a stereo vision application would combine images captured at the same time from two different camera sensors, and stereo audio combines data from two or more microphones. Other analyzers may work multi-modally (by combining vision, audio, gestures and and other sensor inputs). For example, in a house the motion detector has to work with other sensors, to determine that everyone is asleep. These requirements suggest that application development would benefit considerably if the programming system offers some support for indexing data by time. Further, this form of indexing will allow applications to correlate data from different sources.
3.2 Event Reasoning

An event can be defined by its time, location, and identity; time refers to when it happened, location refers to where it happened, and identity refers to who or what was involved. Complex pervasive computing applications constantly take event input from the environment, process it, and act on it. To process the event information they need support to reason about events based on time, location, and identity to ease application development.

Consider the following examples of reasoning about events using time and identity. A surveillance application may want to find out if someone was in the building in the past hour. Or, a traffic application may want to determine if all accidents between SUV and sedans occur early in the morning. Or, a m-commerce application may try to figure out if a user takes soda from a vending machine during a specific time period everyday.

Similarly, there can be examples of reasoning about events using location and time. An application may want to know who was at a given location at the given time. Or, if most of the accidents in an area happened during late afternoon. Or, if trend of sales from vending machines in a given region over the course of a day.

An application may also want to reason about events using time and identity. For example, it may want to know a person’s location at a given time. Thus, we can see that reasoning about events based on time, location, and identity can take various forms.

![Diagram](image)

**Figure 7:** Reasoning Example
Reasoning about events using time, location, and identity can be viewed as a correlation. For example, consider trying to determine where a person was yesterday evening. Figure 7 (same as Figure 4) show a three-dimensional event space consisting of axes that relate to the time, location, and identity of people. A point in the space will represent a particular person, in a specific location, at a given time. A specific identity (person) and time (yesterday evening) instances, define an area in the identity-time plane. We look for a cluster of points that projects onto the specified planar area. If there is such a cluster, it will give the required location(s).

3.3 Distributed Heterogeneous Components

The very nature of the application suggests that the components of the system are going to be distributed and heterogeneous. To be pervasive an application has to exist everywhere in some form wherever a user can go. In other words, application components have to be executing on at different places and on different hardware and software platforms. The distributed components have to be able to communicate with one another.

The heterogeneity may be both at the level of the hardware as well as the system software that runs on the components. For example, a sensor may have an integrated Java virtual machine associated with it and JVM may be the only interface exported by the device to the application programmer. This sensor may be working with a microphone and camera connected to a desktop machine. Further the application programmer may want a choice in the language used for different components of the same application. For example, some compute intensive algorithms (such as a color tracker) may be coded in C but it may be more expedient to code some other part of the application (such as a video display perhaps) in a language like Java. All these application components should be able to communicate and have access to high-level abstractions.

3.4 Dynamic Join/Leave

There should be a natural way for components of the application to join and leave. The system should impose the minimum constraints on the order that components can join or
leave. For example, the system should not mandate that all application components have to start and terminate execution at the same time. The model supported should be based on the way human interaction occurs in the real world. For example, in a telepresence application a participant can join or leave anytime. That may further influence the join and leave of other components. A camera sensor may come alive when a participant joins a chat session and may go offline when the participant signs off. In other words the system may neither know, nor can it enforce, when a component joins or leaves an application.

3.5 Plumbing

Pervasive computing applications consist of several different computational modules that are connected to one another in complex fashion. Further, such topology may not always be statically definable. The programming system should allow for intuitive, efficient, and flexible way of dynamically instantiating such complex pipelines of computational modules. The software infrastructure should not impose restrictions on the application topology.

Apart from the topology, another plumbing issue is the data that the components are allowed to communicate. The infrastructure should allow the application components to communicate arbitrary data without any interpretation on its part. The middleware communication mechanisms should also be intuitive, efficient, and flexible and not burdensome for the application developer.

3.6 Dynamic Resource Management

There are two aspects to resource management in the context of such applications: Firstly, it may be necessary to dynamically provide more processing power for certain aspects of the application than others. For example, analyzing an image for objects of interest may require exploiting data parallelism. Secondly, efficient management and recycling of memory buffers that act as conduits between the computational modules is crucial for application scalability. These applications are continuous in nature. Sensors continually produce data that are passed on to subsequent stages for processing and higher order inferencing. To ease the programming burden the runtime system should facilitate automatic reclamation
of memory resources based on user level hints.

3.7 Failure Handling

There are various aspects to failure handling for these applications. We concentrate on two of the most fundamental ones. The first aspect derives from the continuous nature of these applications (i.e. they have to execute all the time). Temporary failures of individual components should not lead to failure of the entire application. The second aspect derives from the interactive nature of these applications (i.e. they involve components interacting with one another). When a set of components are using a high level abstraction to interact, and one of them fails, the others should be able to take corrective action. For example, if the consumer of a producer-consumer pair fails, the producer should be able to stop creating input. In other words, when a component fails all the other components interacting with it should be informed about its failure, so that they can take the necessary action.

There are other aspects of this application context that we do not look at, but they require support from the programming infrastructure as well. Among these are Security and Privacy. With people who may occupy these spaces unaware of the sensors, there is a need to address security and privacy concerns in pervasive computing applications. Mobility and Disconnection are issues too. Progress in wireless technology has enabled mobility to be incorporated in applications. Keeping track of moving components is an interesting topic. Mobility can result in a component getting disconnected from other application components. Continuous operation in the face of disconnection is another issue. Naming and Discovery mechanisms that enable resource addressing and acquisition are needed to support the dynamic nature of pervasive applications.
CHAPTER 4

RELATED WORK

In this chapter we present projects in different areas that are related to ours. Our goal is to provide a software infrastructure that allows pervasive computing applications handle temporal streams, and to reason about events using time, location, and identity. Thus, the projects that we look at are from the following areas:

- Software infrastructures for developing pervasive computing applications
- Data mining as it relates to the reasoning aspect of our system
- Programming models for general distributed computing applications
- Location technologies, services, and applications
- Overarching projects in pervasive computing domain

4.1 Pervasive Computing Middleware

4.1.1 Context Toolkit

The Context Toolkit ([66, 29]), is a system that provides a framework for building context aware applications. The framework provides the following support for context-aware applications: specification, handling, interpretation, communication, storage, availability, and discovery.

The context is specified using attribute-value pairs. The specification also allows association of multiple dimensions for context provided by a given device. The dimensions are single piece of context vs. multiple pieces of context, related vs. unrelated in case of multiple pieces, and unfiltered vs. filtered context.

For handling context they provide an abstraction called the context widget, that is analogous to the widgets in user interface toolkits. The widgets provide context using both
a query and a notification mechanism. The widgets help separate the low-level acquisition from the application-level use of context. Thus an application that uses a context does not have to change when the device providing it changes. Similarly a device providing the context is oblivious to the application using it.

To support context interpretation, the toolkit provides a mechanism to build a recursive, hierarchical chain of widgets. This mechanism helps when context information may have to go through multiple layers of processing to reach an application. For example, an application may only be interested in knowing if a meeting is going on among some specific set of people. At the bottom layer this consists of information about people in a room. The actual work to be done for the interpretation is left to the applications.

Transparent communication is provided for the case when the producer and consumer of the context are distributed in the environment. The context information can be generated through multiple devices that are distributed (e.g. a location sensing system). Multiple application running on other machines may want to use context provided by a device.

Finally, storage, availability, and discovery of context information are provided as a runtime option for context widgets. Storage allows a device to maintain historical context information, and can be used to establish trends and predict future values. Storage assists in maintaining availability of context information. As the context creators are independent of users, they have no idea when the context information will be needed. Thus, the information has to be constantly available. Discovery allows an application to determine what kind of context information is available in the environment.

Context Toolkit aids applications by providing the means to separate the acquisition of context from its use, but it leaves the use undefined. As the reader will see in Chapter 7 our system, in contrast, does not handle the issue of acquisition, but it provides support to reason about the primary types of context information: time, location, and identity. The relationship of our system to context toolkit can be thought of as a protocol stack, with the context toolkit the lower layer, and our system the higher layer. As an analogue consider networking applications. The TCP protocol is enough for distributed application components to communicate. Still, we have messaging protocols such as MPI and HTTP
based on TCP. These protocols ease the task of communication for various applications. MPI helps parallel scientific applications, and HTTP helps Internet applications.

### 4.1.2 Active Spaces

The Active Spaces project ([65, 3]) presents another software infrastructure for building pervasive computing applications. The project presents the notion of an active space, an active space meta-operating system called Gaia OS, and an application framework to build active space-aware applications. The project defines an active space as “a physical space coordinated by a responsive context-based software that enhances the ability of mobile users to interact and configure their physical and digital environment seamlessly.”

The Gaia OS consists of the following components: event manager, context services, presence service, space repository, and context file system. The event manager is responsible for event distribution in an active space using a decoupled communication model. The communication model consists of one or more suppliers that provide information to a channel, and one or more consumers that receive the information from the channel. The context service provides a registry that allows applications to query and register for context information. Context is represented as a 4-ary predicate defined as follows: $(<\text{ContextType}>,<\text{Subject}>,<\text{Relator}>,<\text{Object}>)$. Complex context types can be constructed by performing first order logic operations on context predicates. The presence service detects digital and physical entities present in an Active Space. The service uses a heartbeat mechanism to maintain soft-state about the presence of entities. The space repository stores information about all the software and hardware entities in an Active Space. The repository provides the ability for attribute-based retrieval of entity information. The space repository uses the presence service to learn about arriving and departing a given space. The file system helps manage user tasks that depend on context information. The tasks are: automatically making personal data available to applications, organizing the data so that an application can locate it, and retrieving data in appropriate format based on user preferences or device characteristics.

The application framework consists of the basic building blocks and mechanisms to
construct active spaces: model, presentation, controller, adapter, and coordinator. Model refers to the component that implements application logic and exports an interface to access and manage application state. The model can be be implemented as a single component, or a set of distributed components. The presentation transforms the application state into an external representation that affects the user environment and can be perceived by the user. Presentations are dynamically attached to and detached from the model. The controller is any entity capable of altering application state. Controllers receive notifications from the model to synchronize the application state. The adapter is a mediator and adds a level of indirection between a controller and a model. A controller invokes methods on an adapter, and the adapter calls the appropriate model. A coordinator manages an active space application consisting of a model, presentation, and controllers. The coordinator manages the application composition, fault tolerance, status monitoring, and life cycle.

Similar to Context Toolkit, the focus in Active Spaces is on issues of handling context information (representation, communication, notification), while leaving the use of context information undefined. As context naturally involves time, location, and identity information Active Spaces applications can use our reasoning system (Chapter 7) to drive application behavior. The sensors that generate input can use our temporal stream handling system (Chapter 6) to communicate with Active Space components that create context information. In other words, our work complements the work in Active Spaces. Another difference is that we focus on pervasive applications for large scale environments (campus surveillance), whereas Active Spaces focuses on local scale environments (moving between office and lab).

4.2 Data Mining

Data mining ([72, 73, 27, 14]) is a technique that allows users to discover patterns and relationships in data, using a variety of analyses, and to make predictions based on the discoveries. The basic steps of data mining for knowledge discovery are: define problem, build data mining database, explore data, prepare data for modeling, build model, evaluate model, and deploy model. We describe each of the steps in detail in the following paragraphs.
Defining the problem deals with knowing the exact objectives that you want to meet with the data mining exercise. For example, the goal could be to increase the response rate of mail offers. A component of the definition will include a justification (e.g. cost versus benefit) for the exercise, and a method to measure the results.

Building the data mining database consists of seven parts. One, collect the data, because there will most likely be various sources of data. Two, describe the data. Three, select a subset of data to mine. Four, assess the quality of data and cleanse it, to ensure correctness and consistency. For example, some fields may have wrong or missing values. Five, integrate and consolidate the data by combining data from different sources into a single database. Six, construct the meta data, to provide information for construction of the physical database. Seventh, construct and load the data mining database.

Exploring the data involves understanding the data, and figuring out the fields that are most important to the problem. Some techniques that can assist exploration are visualization, clustering, and link analysis. Visualization is graphically representing the data. Clustering is dividing the data into groups that are different from one another. Link analysis is discovering associations among values in a database, where an association is a rule of type $A \Rightarrow B$. There are two aspects to associations. First, support or prevalence, that refers to the frequency with which an association appears in a database (i.e. number of entries with both A and B, divided by the total number of entries in the database). Second, confidence, that is the conditional probability of B given A (i.e. the frequency of A and B, divided by the frequency of B).

Preparing the data requires the following. One, select the variables, because not all variable are relevant. Two, select a sample of rows that represent the whole database, as this can speed up the process. Three, transform variables to more suitable representation, so that they are easier to deal with.

Building the data mining model consists of choosing, training, and validating the model type to make the prediction. The models are defined by the algorithms used to process the data. Examples of model types are neural networks, and decision trees. The model is trained using a subset of the data, and then tested against another subset of the data.
Evaluating the model consists of deciding whether it can provide results that are of some use. The evaluation is dependent on the problem that was specified. Finally, if a model successfully passes the previous steps, it can be deployed. The model can be used in two ways. The first is to take actions based on viewing the results of the model. The second is to apply the model to different data sets.

Our work on reasoning about events using time, location, and identity (Chapter 7) and data mining have a similarity; both try to locate a relevant subset of data from a larger collection, by borrowing ideas from Mathematics. (The relevance of a subset is dependent on an application.) Our work differs from data mining in a number of ways. First our system targets the domains in pervasive computing (surveillance, traffic management), whereas data mining targets business-transaction analysis. Second, data mining tries to find random patterns in data that are not obvious to the user. An example pattern is that on weekends beer and diaper sales show correlation. Our work tries to find clusters of events (using time, location and identity), when the application has some idea about the cluster. An example is the locations where a user spent time yesterday evening. Third, we have taken some specific features that may be present in general data mining systems, and built a runtime system around them. For example, we reason about three specific attributes of events, whereas data mining systems may handle multiple attributes. Fourth, we provide generic interfaces that are usable across a spectrum of target applications. Data mining packages are very specific to, and embedded in business applications. Thus, arbitrary distributed pervasive applications cannot use a data mining package directly for their reasoning needs. Fifth, data mining is an offline process, and our reasoning system is online. Our system targets applications that do not need all the data mining features. The limited functionality required allows our system to be available online.

4.3 Programming Models

Parallel Virtual Machine (PVM) [71] provides communication and coordination primitives for parallel scientific applications with support for operating system heterogeneity. Message Passing Interface (MPI) [35] provides a standard for application messaging. It mandates
the semantics of communication and coordination operations among application entities. MPICH [49] and LAM [37] are the most popular implementations for MPI. MPI and PVM are different from our system for handling temporal stream data (Chapter 6) due to a few reasons. One, we target pervasive computing applications, whereas MPI and PVM target parallel scientific applications. Two, we provide efficient support for handling temporal stream data in a distributed environment by allowing applications to index individual stream items. In MPI and PVM the communication primitives only deal with sending and receiving a set of raw bytes. Three, we assume a dynamic join and leave model of application participation, whereas MPI and PVM fix the number of entities at the start of a program. Four, we allow application entities to continue execution when some entities fail, whereas a single entity failure terminates the entire PVM and MPI application. Although PVM and MPI provide a communication mechanism and participation protocol, they do not meet all the needs of distributed pervasive applications. We use the communication facilities of MPI for our system that handles temporal streams. The basic send and receive primitives of MPI help us implement our higher-level abstractions.

For procedural programs Remote Procedure Call (RPC) [19] provides a mechanism to build distributed applications. A server entity exports an interface that multiple client entities can call as if they are calling local functions. In the object-oriented space Common Object Request Broker Architecture (CORBA) [41] and Remote Method Invocation (RMI) [52] provide mechanisms that mimic RPC. CORBA and RMI specify the framework that allows an object to access the methods of a remote object. CORBA specifies an Interface Definition Language (IDL) that an IDL compiler maps to a target language (e.g. C++), and RMI is an integral part of the Java language. RPC, CORBA, and RMI provide transparent communication among distributed entities, and can be used to implement pervasive computing middleware. In fact for our system that handles temporal streams, we use RPC-like communication to implement the higher-level abstractions.

Linda [38, 15, 25], TSpaces [77, 50], and JavaSpaces [36] provide the tuple space model of building distributed systems. In this model producers store tuples of attributes and values in a tuple space. Consumers retrieve tuples they are interested in by providing a
template to the tuple space to filter the tuples. This model of programming provides an anonymous and asynchronous communication among entities. Such systems provide fairly generic programming capabilities for data sharing and synchronization. In theory tuple spaces can be used to build any distributed computing application. However, in practice they do not have the right level of abstraction for application development in different domains. For example, the pervasive applications that we target need abstractions that help applications to access items in a temporal stream, or allow the applications to reason about events using time, location, and identity. A middleware providing the required abstractions for a domain will be easier to use for application development than tuple spaces. Tuple spaces can be used as an implementation platform for the pervasive computing middleware.

4.4 Location Technologies, Services, and Applications

Various location-sensing systems have been developed. The Global Positioning System (GPS) [39] triangulates signals from satellites to determine geographic position. The Active Badge system [75] uses a badge placed on an object, that periodically emits an infrared signal. A server collects this data from sensors in the environment, and provides location information. In the Active Bat system [43] people and objects carry wireless devices called Bats that emit ultrasonic pulses. Ceiling-mounted receivers measure the times of flight for a pulse, and a controller uses this information to determine the location. For accuracy, the controller synchronizes the Bats and the receivers, by sending a radio frequency signal to the Bats and a reset signal to the receiver using a wired serial link.

The RADAR system [16] uses IEEE 802.11 WaveLAN technology to determine location. It measures, at a base station, the signal strength and signal-to-noise ratio of signals sent by wireless devices, and uses this data to compute 2D position. The Cricket Location Support system [62] uses ultrasound signals sent by emitters in the environment. A mobile object uses a receiver to obtain signals from multiple emitters to determine its position. Cricket uses a radio frequency signal to synchronize the senders and receivers. Our reasoning system assumes the availability of location information for events that occur during an application lifetime. The systems listed above provide the location information required
by an application using our reasoning middleware that the reader will see in Chapter 7. The fact that multiple technologies provide location information affects the design of our middleware. An application should be free to use any one of the location systems, and the middleware should not restrict the choice.

Next we consider systems that provide location information as a service to applications. In the location system of QoSDREAM project [55] the emphasis is on presenting location in a technology-independent manner. The information is modeled as regions and interaction between regions, and applications can choose the granularity of detail they require. Mobile Shadow [33] provides an architecture for building location-aware services. It consists of a virtual representation of physical space, users and services. When a user changes location, his virtual representation does too, and this activates the services in that location. In the Rover project [17], the system tracks the location of users and adapts application-level information to different link-layer technologies and client-device capabilities. Leonardi and Kubach [51] present the architecture for a large-scale distributed location service. The location data are replicated on various servers with varying accuracy levels. More accurate copies are nearer to the current position of a mobile object. Less accurate copies are nearer to the applications using the location data. The location services provide location information that is one of the defining attributes of an event, and our system that helps applications reason about events (Chapter 7) assumes the presence of location information. Thus, applications can use these location services in conjunction with our reasoning system.

We next discuss some applications that use location information. The GUIDE project [26] is used to aid tourists in Lancaster. It uses location data to determine the information to provide to the tourists. The Active Office project [42] uses the Active Badge system to obtain location information, and it provides a distributed service that allows access to the information. The simplest application provides information about people or objects based on a query (textual or graphical). Other applications help in choosing a communication mechanism based on the locations of the parties. The Sentient Computing project [13] uses the Active Bat system to obtain location information. The simplest application shows a model of the environment with people and objects in it. Other applications allow interfaces
to follow a user whether it is his desktop, his phone, or cameras tracking his movement. These applications are based on a specific location technology, and reason about location information in an application-specific manner. They determine the location of a user, and that serves as a trigger for change in the input and output mechanisms available to the user. Our reasoning system in Chapter 7 is not based on a specific technology, helps applications reason about events along three axes (time, location, and identity), and allows reasoning in an application-independent manner.

4.5 Grand Projects

Projects such as MIT Oxygen [7], Berkeley Endevour [45], and Georgia Tech Infosphere [5] share our high level objectives, namely, to support pervasive computing. We differ in specific research goals from them. Oxygen’s focus is to develop the fundamental technology for a pervasive computing fabric, and address the issues of networking them, and developing adaptive applications on top of them. Endevour’s focus is to develop scalable services such as file systems on planetary scales on top of pervasive infrastructure with varied network connectivity. Infosphere’s focus is to devise middleware that will help the end user combat with the explosive growth of information on the Internet and allow the information infrastructure to scale as the World Wide Web grows. Our research goal is a seamless distributed reasoning system, which is complementary to these projects.
CHAPTER 5

BRANCHING WITH D-STAMPEDE AND CREST

In the previous chapters we have built the case for the intellectual contributions of this thesis. Chapter 2 surveys the domains of surveillance, traffic management, and mobile commerce. The survey shows the following common features among the applications in the domains of our interest.

- The applications are not really distributed and pervasive.
- Applications handle their live stream data in an ad-hoc manner
- Applications do not organize historical data, and at best perform limited reasoning on current data
- The reasoning is application-specific, and unusable by other applications

The survey concludes that distributed pervasive computing applications need the support of middleware to handle temporal streams, and reason about events. Chapter 3 presents the requirements for the middleware to support the applications. The requirements are as follows:

- Handling live temporal streams by allowing applications to index the stream data using a notion of time
- Organizing historical data to reason about events using time, location, and identity to drive application behavior
- Supporting heterogeneous components, so that they can work together even with different hardware and software
- Allowing application entities to join and leave arbitrarily so that they can create and maintain dynamic interactions
• Providing the plumbing so that application entities can instantiate, use, and modify complex communication pipelines

• Managing resources (processing power, memory, etc.) in the face of constantly changing application behavior

• Handling failures by providing two things: an ability to execute in the face of partial failure, and failure notification

Chapter 4 looks at other related work in the following domains:

• Other software infrastructures for developing pervasive computing applications, namely Context Toolkit and Active Spaces

• Data mining that detects patterns in data, as it relates to the reasoning that our system does for events using time, location, and identity

• Programming models for general distributed computing applications such as PVM, MPI, CORBA, RMI, and Tuple Spaces (Linda, TSpaces, and Java Spaces)

• Location technologies (GPS, Active Badge, Active Bat, RADAR, etc.), services (Qos-DREAM, Mobile Shadow, and Rover), and applications (Guide, Active Office, and Sentient Computing)

• Overarching projects in pervasive computing domain such as MIT Oxygen, Berkeley Endeavour, and Georgia Tech Infosphere.

None of the projects meets our goals for distributed pervasive applications: handling temporal streams, and organizing historical data to reason about events. In Chapter 6 we present our first intellectual contribution: D-Stampede system that allows application to handle temporal streams. In Chapter 7 we present our second intellectual contribution: Crest system that allows applications to reason about events, using time, location, and identity. D-Stampede and Crest together provide the necessary middleware infrastructure for distributed pervasive computing applications.
CHAPTER 6

D-STAMPEDE: HANDLING TEMPORAL STREAMS

In this chapter we present the D-Stampede programming system. The main feature is that it allows an application to handle temporal streams.

![Octopus Hardware Model](image)

**Figure 8:** Octopus Hardware Model

The hardware model assumed by D-Stampede is the Octopus shown in Figure 8 (same as Figure 1). Different kinds of sensors (and data aggregators located near them) collect raw data and perhaps do limited processing such as filtering. However, extraction of higher order information content from such raw data requires significantly more processing power. For example, microphones may collect audio data, but higher order processing is needed for voice recognition. Thus there is a continuum of computation and communication resources that a pervasive computing application spans.

The computational model supported by D-Stampede is pictorially presented by the thread-channel graph in Figure 9 (same as Figure 2). The model captures a huge class of inherently distributed applications that are emerging in the context of pervasive computing. The threads map to computing devices ranging from small to high-end processing elements.
scattered in a pervasive computing environment. Channels serve as the application level conduits among the threads, specifically tuned to handle time-sequenced streaming data. D-Stampede provides APIs for the dynamic creation of threads and channels, and for the dynamic establishment (and removal) of the plumbing among them.

6.1 Architecture

The D-Stampede architecture has the following main components: *Channels, Queues, Threads, Garbage Collection, Handler Functions, Complex Data Structures, Real-time Synchrony, Nameserver, Fault Tolerance*, and support for *Heterogeneity*. D-Stampede borrows some of its architectural components from Stampede ([59]). The borrowed components are channels, queues, threads, garbage collection, handler functions, complex data structures, and real-time synchrony. D-Stampede extends Stampede in the following ways. First, is the observation that the abstractions Stampede provides for parallel multimedia applications are applicable to distributed pervasive applications that D-Stampede targets. Second, Stampede limits the existence of the computational model, the dynamic thread channel graph of Figure 9, to a cluster. Thus, executing a distributed pervasive application on Stampede will present an infeasible requirement; connecting all sensors to nodes of a cluster. D-Stampede extends the computational model to the whole Octopus (Figure 8). Finally, D-Stampede adds the following features that are important for distributed pervasive applications: *name
server, fault tolerance, and heterogeneity support.

6.1.1 Threads, Channels, and Queues

D-Stampede provides a uniform set of computational abstractions across the entire hardware continuum: threads, channels, and queues. Stampede threads are POSIX-like and can be created in different protection domains (address spaces) for memory isolation purposes [59]. Channels and queues are system-wide unique names, and serve as containers for storing time-sequenced data items produced by threads. They facilitate inter-thread communication and synchronization regardless of the physical location of the threads, channels, and queues. A thread (dynamically) ‘attaches’ to a channel (or a queue) for input and/or output. Once connected, a thread can do I/O (in the form of get/put items) on the channel (or queue). The items represent some application-defined splitting of stream data (for e.g. frames of video, audio samples, etc.). The timestamps associated with an item in a channel (or queue) is user defined (for e.g. timestamps may be the frame number of video generated in a telepresence application). The collection of time-sequenced data in the channels and queues is referred to as space-time memory [63].

While the channel allows random access by a thread for items of interest (based on the timestamp value associated with an item), a queue, as the name suggests allows FIFO access to items contained in it. Another difference is that in a channel there can only be one item with a particular timestamp, whereas in a queue there can be multiple items with the same timestamp. The queue abstraction is primarily designed to exploit any data parallelism in an application. For example, if it is desired to analyze a given frame of video for objects of interest, then the frame can be partitioned into frame-fragments (all having the same timestamp) and placed in a queue by a splitter thread. A distinct thread can analyze each frame-fragment for objects of interest. A joiner thread can then stitch together the composite analyzed outputs.

At a conceptual level, a D-Stampede computation with threads, channels, and queues is akin to a distributed computation specified by a set of processes connected by sockets.
This conceptual equivalence makes it an easy transition from socket-based distributed programming to channel-based D-Stampede programming. The power of D-Stampede is the ability afforded to the application for reasoning about program behavior based on time (an important feature in interactive distributed applications such as telepresence), and temporal correlation among data generated by different sources (e.g. audio and video coming from an avatar in telepresence).

6.1.2 Garbage Collection

D-Stampede aids the development of highly dynamic applications. The runtime mechanisms in D-Stampede mirror the dynamism in the applications. The human body uses selective attention as a means of filtering out inputs from unrelated sensors while performing a specific bodily function (take primate vision for example). In a similar vein, D-Stampede facilitates selective attention at two levels: first by allowing a thread to dynamically choose the set of channels and queues it wants to perform I/O on, and second by using timestamps as a filtering mechanism. API calls in D-Stampede facilitate a given thread to indicate that an item (or a set of items) in a channel or a queue is garbage so far as it is concerned. Using this per-thread knowledge, D-Stampede automatically performs distributed garbage collection [58] of timestamps (i.e. items with such timestamps) that are of no interest to any thread in the D-Stampede computation.

6.1.3 Handler Functions

D-Stampede allows association of handler functions with channels and queues for applying a user-defined function on an item in a channel (or a queue). The handler functions come in handy in a variety of situations. For e.g., if an item (which may be a complex user-defined data structure) has to be transported across address spaces (and/or machine boundaries), the user can define serialization and de-serialization handlers that D-Stampede will invoke as necessary to perform the API calls (such as a get from a thread in a remote address space). Similarly, once an item is determined to be garbage by the runtime system, a user-defined handler can be invoked to garbage collect memory buffers in the user space associated with that item.
6.1.4 Complex Data Structures

Often applications want to transfer complex data structures across machines. By complex, we refer to structures that may reside in memory non-contiguously, with different fragments connected by pointers or references. While it is not possible to have a generic function in the runtime system to serialize or de-serialize these structures, D-Stampede facilitates the handling of such objects in a structured manner. The application can install an object-specific serialization routine at the sender. Thereafter, every time such a complex object needs to be transported, the runtime calls this routine and transfers the resulting serialized form of the object. The corresponding de-serialization happens at the receiver using another application-installed function. This mode of transfer offers four advantages. First, it provides the application program a generic interface for mapping an object to a serial form. Second, the application does not need to bother about data-marshalling every time it wants to send or receive complex structures. Third, in the event multiple receivers of a single complex object are located in the same machine, the unpacking is done only once (by the runtime), thereby saving some processing. Fourth, the complex object is locally cached at a receiving node resulting in considerable sharing of the network bandwidth, when there are multiple receivers at a node.

6.1.5 Real-time Synchrony

The timestamp associated with an item is merely an indexing system for data items, and does not in itself have any direct connection with real time. For pacing a thread relative to real time, D-Stampede provides an API for loose temporal synchrony that is borrowed from the Beehive system [68]. A thread can declare a time interval at the end of which it wishes to synchronize with real time. The thread also specifies a tolerance margin and an exception handler. As the thread executes, at application-defined points, it performs a D-Stampede call attempting to synchronize with real time. If it is early, the thread waits until the declared time interval passes, and synchrony is achieved. If it is late by more than the specified tolerance, D-Stampede calls the thread’s registered exception handler that can attempt to recover from this slippage.
Using these mechanisms, for example, a camera in a telepresence application can pace itself to grab images and put them into its output channel at 30 frames per second. It will specify $(1/30)^{th}$ of a second as the interval and a tolerance of zero. After the thread puts each frame in the channel, it will make the call to synchronize with real time. If the thread takes more than $(1/30)^{th}$ of a second to handle one frame, the registered exception handler is executed. If the it thread takes less than $(1/30)^{th}$ of a second, it will wait till the time passes before processing the next frame. A possible way to handle the loss of synchrony may be to skip the next frame.

6.1.6 Name Server

D-Stampede infrastructure includes a name server. Application threads can register (and de-register) all pertinent information (such as names of channels and queues, as well as their intended use in the application) with this name server. Any new thread that starts up in the application anywhere in the entire network of the Octopus model can query this name server to determine resources of interest that it may want to connect to in the computation model (Figure 9). This facilitates the dynamic start/stop feature we alluded to in the requirements chapter (3). We do not mandate that only our name server be used – an application can use any arbitrary name server. All applications that utilize the name server know its location, and the mechanism for sharing this information is the responsibility of the applications and outside of D-Stampede.

6.1.7 Fault Tolerance

D-Stampede provides the following support for fault tolerance to pervasive-computing applications. First, if one component fails, the remaining components are able to continue. Second, among the remaining components, those that were interacting with the failed component, are informed about the failure.

To explain how our system provides fault tolerance, we briefly describe the structure of an application (a more detailed example is given when we describe our experiences with D-Stampede). An application consists of some components that execute at the head, and some at the tentacles of the Octopus model (Figure 8). An application starts as a set of
head components \((H_i \ i = 1, \ldots, n)\). Then a set of tentacle components \((T_k \ k = 1, \ldots, m)\) join the application. Each tentacle component \(T_k\) connects to a head component \(H_i\) (and only to one head component). Multiple tentacle components can connect to a given head component.

Now, consider our first requirement for fault tolerance. There are two types of component failures: tentacle and head. When a tentacle component fails, its connections to the head component breaks. The head component responsible for the failed tentacle component, updates the information about its connected components. When a head component fails, its connections to all its tentacle components break. The affected tentacle components can continue participating in the application by connecting to other head components. The head components do not have direct connection to each other, so they employ a heart beat mechanism to keep track of other head components. When a head component fails, its heart beat is lost, and the other head components update their knowledge of existing head components.

Currently the state information stored at the failed head component is lost. To prevent the loss of state information, the head components can use some well-known mechanism (e.g. some checkpoint mechanism) to keep their state persistent. When a head component fails, the other nodes can be given responsibility for parts of the application state from the failed node. The rationale for this design is that the head components are expected to be more stable and less failure-prone than tentacle components. The head components are expected to run on a back-end cluster, while the client components are expected to run on end devices. For example, in a surveillance application, the client components can be the cameras, and the head components can be the parallel application analyzing the camera inputs. The cameras can easily be smashed by an intruder, whereas the cluster is less likely to be compromised.

Now, consider our second requirement for fault tolerance. The application components interact using the channel (queue) abstractions. For the notifications to work, all components using a channel (queue) use the following protocol. They have to register with the system that they are using the channel (queue). A component has to register separately for
every channel (queue) of interest. There are two effects if a component does not register that it is using a channel (queue). One, it will not get a notification when another component using the channel (queue) fails. Two, the other components using the channel (queue) will not get a notification when the unregistered component fails. When a component fails, the system uses the registration database to inform the other registered components about the failure.

The registration process is used by components to tell the system that they are using a particular channel (queue). By default the system does not keep track of the components using a channel (queue). This registration database consists of two types of information. One, the list of channels (queues) of interest to a tentacle component. Two, the list of components registered for a channel (queue). This registration database is distributed among the head components. Each head component maintains the information about the channels (queues) for each of the tentacle components that it handles. For a specific channel (queue) a head component acts as a home node. The home node maintains the list of clients interested in the channel (queue). When a tentacle component fails, its head node consults the channel (queue) list for the failed component. It then notifies the home nodes for the channels (of the failed component) about the failure. Finally, the home nodes use their component lists for the channels to send the failure notifications.

### 6.1.8 Heterogeneity

We made a conscious decision to develop D-Stampede as a runtime library on top of standard operating systems. Accordingly, we have architected the D-Stampede system as “client” libraries on the end devices (i.e. tentacles of the Octopus) and a “server” library on the cluster (i.e. the head of the Octopus). This organization has no bearing on the generality of the computational abstractions provided by D-Stampede. The API calls of D-Stampede are available to a thread regardless of where it is executing. To reiterate, an application can write a program, which has threads executing in the clients and the servers. We presume that the thread running on the server will be the ones handling heavy processing loads, whereas the threads on the clients will be mostly handling input and output.
D-Stampede accommodates heterogeneity of components in two ways. Firstly, at the operating systems level. The server library of D-Stampede has been ported to the following platforms: DEC Alpha-Tru64, x86-Linux, x86-Solaris, and x86-Windows2000. The client library has also been ported to multiple platforms: x86-Linux, StrongARM-Linux, and x86-Windows2000. Secondly, at the programming language level: the end devices (i.e. the clients), can be programmed in either C or Java. A D-Stampede application can have some parts written in C and some parts written in Java sharing the same data abstractions. An application can be created that has a server and client components using any of the supported platforms. For example, an application can have the server component running on x86-Solaris cluster, while the clients consist of an arbitrary mix of C and Java on the supported platforms. We elaborate on the implementation details in the next subsection.

6.2 Implementation

![Diagram of D-Stampede Implementation]

**Figure 10: D-Stampede Implementation**

D-Stampede is implemented as a runtime library that has two parts, a *server* written in C running on a high performance cluster, and *clients* (written in C or Java) that can run anywhere in the distributed system. Figure 10 shows the organization. Though the
programming model is uniform and does not distinguish between client and server, this
dichotomy exists for both historical reason (Stampede was originally developed as a cluster
computing library) and a reflection of the Octopus analogy we alluded to earlier (Figure 8).
The distributed end points (clients) are usually connected to sensors and in general serve
as the capture and access points in the environment, while heavy duty computation (such
as tracking and image analysis) is performed on the cluster (server).

6.2.1 Client Library

The D-Stampede APIs are exported to the distributed end points in a manner analogous to
exporting a procedure call using an RPC interface [19]. There are client libraries available
for both C and Java. The Java client library encapsulates the D-Stampede APIs as objects.
The application level D-Stampede programs running on the end devices (i.e. clients) can
be written in C or Java and they coexist as parts of a single application.

A TCP/IP socket is used as the transport for communication between the client and
the server libraries. The Java client library uses our own data representation to perform
the marshalling and unmarshalling of the arguments, while the C client library uses XDR
[70].

6.2.2 Server Library

The server library is implemented on top of a message-passing substrate called CLF, a
low level packet transport layer developed at Compaq CRL. CLF provides reliable, ordered
point-to-point packet transport between the D-Stampede address spaces within the cluster,
with the illusion of an infinite packet queue. It exploits shared memory within an SMP, and
any available network between the nodes of the cluster, including Digital Memory Channel
[40], Myrinet [20], and if none of these are available, UDP over a LAN.

There is a listener thread on the cluster (part of the server library) that listens to new
end devices joining a D-Stampede computation. Upon joining, a specific surrogate thread
(see Figure 10) is created on the cluster on behalf of the new end device. All subsequent
D-Stampede calls from this end device are fielded and carried out by this specific surrogate
thread. State information pertaining to an end device is maintained by the server library
via the associated surrogate thread. The surrogate thread ceases to exist when the end device goes away. The creation/annihilation of this surrogate thread on the cluster mirrors the joining and leaving of an end device.

Garbage collection is performed on the cluster concurrent with application execution. The surrogate threads participate in garbage collection on behalf of the end devices. The server library notifies the client libraries for any storage reclamation that has to happen on an end device as a result of garbage collection on the cluster.

6.2.3 Supporting Heterogeneity

Java clients can run on any end device that has a JVM. The server library (which is in C) and the C client library have been ported to the following platforms: DEC Alpha-True64 Unix, x86-Linux, x86-Solaris, and x86-Windows2000. The implementation supports combinations of 32-bit and 64-bit platforms for the end devices and the cluster. Any combination of end devices and a cluster platform can host a single D-Stampede application.

6.2.4 Supporting Handler Functions for End Devices

D-Stampede allows user-defined actions to be performed on an item in a channel or a queue via the handler function. Garbage collection is a good example for use of this mechanism. Upon the D-Stampede runtime determining that an item in a channel is garbage, it calls the user-defined handler to free up any memory in user space associated with that item. While this is quite naturally implemented for D-Stampede threads running in the cluster, special handling is needed for the end devices. When requested by an end device to install a handler, its surrogate installs a generic handler function on the cluster. When this generic handler is invoked by the D-Stampede runtime, it collects the information on behalf of the end device and communicates it to the end device at an opportune time (for e.g. when the next D-Stampede API call comes from the end device).

6.2.5 Complex Data Structures

We implemented the transport of complex data objects through an intermediate generic form which we call the chunk vector, an array describing the different consecutive blocks
of a complex object. Each array element has two fields: size of a block and a pointer to the actual physical chunk of that memory. The first element of the chunk vector always contains the number of such blocks present in that vector. The chunk vector is a very generic representation of a complex object, however, since the topology (or the connectivity) among different parts is only known to the application layer, it is the user who has to supply routines for converting a data structure into the chunk vectors and vice versa. The de-serialization routine takes care of any specific way the chunks need to be stitched together to recreate the complex object. Since the chunk vector is a generic structure, flattening of these chunk vectors into network-ready flat stream of bytes (and vice versa) is easily handled by the D-Stampede runtime.

Figure 11: Registration process for faults

6.2.6 Supporting Fault Tolerance

The system provides fault tolerance with the cooperation of two threads. One, the proxy thread, of which there is an instance for every client. Two, a failure-notification-handler thread, of which there is a single instance in every cluster address space.

For each channel (queue) there is a set of clients that may want a notification upon the failure of any one client in the set. Each client that wants failure notification has to
register its interest (Figure 11). The client sends the registration request to its proxy thread (Figure 11 step a and c). The client provides the proxy thread with the channel (queue) descriptor of interest, and the location (IP address and socket number) where it wants to receive the notification. The location provided by the client can be its own location or some other location. On receiving the request, the proxy thread adds this channel (queue) descriptor to a descriptor list (Figure 11 step b and f). This descriptor list contains all the channels (queues) that a given client has registered for notification. Further, using the channel (queue) descriptor, the proxy thread determines the home node for the channel (queue). The home node for a channel (queue) is the address space where it was created. The proxy thread sends a register message to the failure-notification-handler thread at the home node (Figure 11 step c and g). The message contains the location information for the client. On receiving the message, the failure-notification-handler thread at the home node, adds the information to a client list it maintains for the channel (queue) specified in the message (Figure 11 step d and h). For channels of interest the failure-notification-handler thread maintains a set of lists (one list per channel). For some channels there may not be any interest (there will not be any lists for these channels).

When a client fails, its connection to the proxy thread breaks. The proxy thread then
consults the descriptor list, to determine all the channels (queues) of interest to the failed client (Figure 12 step a). The proxy thread sends a failure notification to the home nodes of all the channels (Figure 12 step b). The failure-notification-handler thread at the home node, uses the channel (queue) information to determine the other clients registered for the same channel (queue) (Figure 12 step c). It then sends all those clients the information about the failure at their specified locations (Figure 12 step d). This information consists of the channel descriptor, and the failed-client identifier (IP address). In response, the clients are free to take any action (the system does not impose any requirement on the action).

We want to point out one aspect of the fault-tolerance implementation: its effect on system performance. The cost has two parts: one, the client call to register interest in a channel; and two, the cost of informing the remaining clients when a registered client fails. For each client that registers, these are one-time costs at the server. These costs can be considered to be negligibly small with respect to the life of an application. There is no other performance penalty at the server related to providing fault tolerance (there is no impact on the calls unrelated to fault tolerance).

6.3 Summary

In this chapter we have presented the first of our two main contributions: D-Stampede system to handle temporal streams in distributed pervasive applications. We have presented the hardware model that D-Stampede assumes, the computational model that it provides, the architecture components, and implementation details. The hardware model is an Octopus (Figure 8), and it represents a computation and communication continuum. The computational model is a dynamic thread channel graph (Figure 9) that maps to various distributed pervasive applications. The architecture consists of the following components: Channels, Queues, Threads, Garbage Collection, Handler Functions, Complex Data Structures, Real-time Synchrony, Nameserver, Fault Tolerance, and support for Heterogeneity. The implementation consists of two libraries: client, and server. We present a qualitative and quantitative evaluation of the D-Stampede system in Chapter 9.

51
CHAPTER 7

CREST: REASONING ABOUT EVENTS USING TIME, LOCATION, AND IDENTITY

Figure 13: Universe Model in Crest

In this chapter we present the Crest system that allows an application to reason about time, space, and identity of events. The Crest system assumes a model of the universe shown in Figure 13 (same as Figure 3). It consists of a set of participation servers and event stores, supporting a set of applications. Each application consists of a set of software entities working together. For example, in Figure 13 entities 1, 3, 4, 8, 9, and 10 participate in application 251. An entity can participate in more than one application (e.g. in Figure 13 entity 4 is a participant in three applications). The participation server helps a set of entities working together in an application to discover one another for the purposes of communication. Application entities can generate events, store them at an event store, and correlate events. The entities can communicate with one another directly, or indirectly through the event store.
7.1 Architecture

The architecture has the following components: events, reasoning operations, event store, participation protocol, and communication system.

7.1.1 Events

The application behavior consists of a sequence of events. Formally, an event is a 3-tuple and consists of time, location, and identity information. Time, location, and identity refer to the following: when did it take place, where did it take place, and who or what was involved in it. The time can be real (clock and calendar time) or virtual (some integer value). The location can be in different forms: GPS coordinates, application-specific notation, etc. An identity can represent anything: real world object (person, place, thing, etc.); a virtual identity (e.g. a module that fuses data from multiple sensors); data used in an application etc. An identity may also refer to a set of entities (e.g. a crowd is a set of people). Application behavior is governed by the interaction between the identities. For example, in a surveillance application a camera is an identity. The captured image is an identity, so is the object in the image. A room where a camera is placed is an identity.

An application may use multiple types of time, location, and identity. Thus an application can reason about multiple types of events. For example, if an application has two time types (TIME1, TIME2), one location type (LOCATION), and two identity types (IDENTITY1, IDENTITY2). It can can have multiple event types such as: EVENT1 = (TIME1, LOCATION, IDENTITY1), EVENT2 = (TIME1, LOCATION, IDENTITY2), EVENT3 = (TIME2, LOCATION, IDENTITY1), and EVENT4 = (TIME2, LOCATION, IDENTITY2). Also, multiple applications can reason about the same event type.

7.1.2 Reasoning Operations

Store: To reason about time, location, and identity we have to store events so that we can look them up in the future. The function to store an event in a database takes the event instance as an argument (Store (E)). Any entity, in the set of distributed application entities, can store event information; not necessarily the entity that generates the event.
The entity that stores an event has no idea about entities that may use the event. The event may have attributes that allow the user entity to infer about the producer entity.

**Find:** We need an operation to find past event-related information. There is a set of global find operations: Find (E, L), Find (E, T), Find (E, O), Find (E, T, L), Find (E, T, O), Find (E, L, O). E refers to the event class to search, L to the location instance, T to the time instance, O to the identity instance. The event class is an argument to the find functions because, by definition, multiple event classes can have an overlap of location, time, or identity classes as components. (Two event classes cannot have all of time, location, and identity component classes the same.)

A combination of store and find operations provide indirect communication between entities. The indirect method is in contrast to the direct communication between a sender and receiver. (Section 7.1.5 presents the direct communication between entities.) Multiple find operations can be called for a single store operation.

**Correlate:** We need operations to correlate events across time, location, and identity. There are two primary challenges in providing the correlation operations: simplicity, and extensibility. These challenges arise because the definition of correlation is dependent on the application. The middleware can provide a basic definition of correlation, but it cannot
mandate that applications follow the given definition.

Figure 14 presents a pictorial representation of the correlation operations. (Figure 14 is the same as Figure 4). An event definition in an application can be thought of as a three-dimensional space. (Each event instance is a point in the space.) Each of the event attributes (time, location, and identity) is an axis in the space. Each axis is only conceptually a line. Two factors prevent it from being a real line: one, a point on the axis can have multiple attributes; two, each axis is a collection of discrete rather than continuous values.

A correlate operation takes values in two event dimensions as arguments, and returns values in the third event dimension. The correlate operation works as follows:

- It determines the planar area defined by the values in the two input dimensions
- It determines the cluster of points in the three-dimensional space that projects onto this planar area
- If there is a projection, it returns values in the third dimension from the cluster as a result of the operation

7.1.3 Participation Protocol

There is a protocol for entities to participate in an application. In our protocol an entity explicitly joins or leaves an application. Initially an entity starts an application. It registers the application name with a participation server. Later on, when another entity wants to join the application, it first contacts the participation server. The participation server assigns a locally unique identifier to this joining entity, and also gives it information about all the other entities in the application. By locally unique we mean that the assigned identifier is unique with respect to all other entities in the given application. An entity can join multiple applications, and it will get a locally unique identifier for each application. An entity can also join a single application multiple times, and it will get locally unique identifiers for each join instance. Using this local identifier the entity contacts the current application entities, and informs them that it is joining the application. The joining process can be active or passive. In the active mode the joining entity informs all existing entities

55
about its presence. In the passive mode it informs an existing entity only when it wants to communicate with that entity. Figure 15 shows an active join for three entities. The messages are labeled with integers to show the temporal sequence they will follow.

The tuple consisting of the application name and the locally unique identifier provided by the participation server, forms a globally unique identifier for an entity. The middleware for its part is only concerned with the locally unique identifiers, not the globally unique ones, for the operations it performs on behalf of the entities. Applications are free to use the globally unique identities if they desire.

![Diagram](attachment:diagram.png)

**Figure 15:** Participation Protocol: Join

When an entity wants to leave an application, it informs the participation server and the other entities of its intent. Similar to a join, an entity can leave in an active or passive mode. Figure 16 shows an active leave for an entity among three participating entities. The messages are labeled with integers to show the temporal sequence they will follow. The contract for an entity is that when it leaves, it has to inform all the entities that it had contacted when it joined. When a new entity joins (leaves), the existing entities proceed as usual, except for an extra message they may receive from the joining (leaving) entity. The entities can join or leave an application in any sequence.

The participation protocol enables application entities to create and maintain a dynamic
interaction. The protocol does not impose any constraint on the number of applications an entity can join, or the number of times it can join a single application. Nor does the protocol impose any constraint on the number of entities in an application at any time, or the the connection topology set up by the application entities. (The connection topology is governed by the join and leave messages exchanged among entities.) The only requirement for an entity is to contact the participation server for joining and leaving an application.

![Diagram](image)

**Figure 16**: Participation Protocol: Leave

### 7.1.4 Event Store

The events that an application reasons about are archived in an event store. There can be multiple event stores in the universe, and each participation server knows about at least one event store. When an application starts, it is assigned an event store by the participation-protocol server. This assignment holds for the life of the application, that is from the time the participation server receives a start-application message to the time it receives a stop-application message. There are two design choices we make about event store assignment: who assigns the event store for an application, and do all entities of an application share an event store. Regarding the first choice, we decided on the participation server assigning the event store. An alternative was to allow one of the application entities to choose an arbitrary
event store. Our approach allows scheduling the event stores among different applications based on some policy (e.g. round robin). Also, our approach provides a guaranteed point of contact to propagate the event store assignment. As part of the protocol each joining entity has to contact the participation server, but it does not have to contact all the other application entities. Regarding the second choice, we decided on a single event store for an application. An alternative was to allow entities to store events locally with themselves. We chose a single store because an entity can leave arbitrarily; once an entity leaves, the event data will be unavailable for the remaining entities.

7.1.5 Communication System

The system provides asynchronous communication between entities. To begin communication, a receiver must notify a sender of its interest in receiving data from the sender. Further a receiver must install a handler to receive data messages from the sender. Whenever a data message arrives at the receiver, the sender-specific handler is invoked by the runtime. A receiver can install multiple handlers but cannot specify more than one handler for a specific sender. A sender will start sending messages to a receiver after it receives a notification of interest from the receiver. A sender can send messages to multiple entities that have registered an interest. The sender and receiver are free to exchange any data as the system does not interpret the message payload. Communication ends when the receiver notifies the sender that it is no longer interested in receiving data. An entity can play the role of both sender and receiver for all applications it joins.

The communication interface consists of the following functions at the receiver end:

- **Install (Application, Sender, Handler):** to install a handler for a specific sender entity in an application
- **Install (Application, Handler):** to install a handler for all possible sender entities in an application
- **Initiate (Application, Sender):** to notify a sender in an application to begin data transmission
- **Terminate (Application, Sender):** to notify a sender in an application to end data transmission

The communication interface consists of the following function at the sender end:

- **Send (Application, Receiver, Data):** to send data to a specific receiver in an application

- **Send (Application, Data):** to send data to all possible receivers in an application

The receivers for a send function are those that have sent an initiate notification to the sender who makes the call. The number of such receivers may be less than the total entities in an application. Further, the set of receivers interested in a sender will most likely vary for each sender.

The choice of asynchronous communication is governed by the application domain. In pervasive computing, unlike scientific computing, the application behavior is driven by environment events. The environment events by nature are asynchronous: someone enters or exits a room, the light turns on or off, etc. Thus, asynchronous communication fits pervasive computing applications better than a synchronous model with a matching send and receive operation for a message transmission.

In contrast to the direct message exchange between entities that we have just explained, the system also provides indirect communication using a combination of store and find operation. A sender can use the store operation to archive a message in the event store. A receiver can the find operation to obtain the message by specifying suitable attributes. The indirect method adds overhead to message delivery compared to the direct version. The overhead has two parts: number of hops, and event store processing. A message travels two hops (sender to event store, and store to receiver), compared to one hop (sender to receiver) for direct communication. The event store processing refers to the database operations involved in initially storing the message at the event store, and then later searching for the message. The indirect method is suitable for the case when a producer may generate data and leave before the consumer joins the application.
7.2 Reasoning Interface

In this section we list the interface that the system provides for reasoning operations. For each operation we provide a simple example of how it can be used. In the examples, L represents a location, T a time, and O an identity. The array form L[][] refers to a set of locations, T[][] to a set of times, and O[][] to a set of identities.

- L.CorrLate (O) gives the association between a location and an identity. It returns the set of times (T[]) that the identity was at the location. If L is an hourly parking lot, O is a blue car, then the set of returned times can be the duration of the car being parked. In the given scenario, if the car has been parked overnight, it may mean suspicious behavior.

- L.CorrLate (T) gives the association between a location and time. It returns the set of identities (O []) that were at location L at time T. If L refers to a meeting room, and T to a time during office hours, a non-null return set may mean a meeting.

- T.CorrLate (O) gives the association between an identity and time. It returns the location where the identity was at the given time. If T is a specific instant in time, then the answer will be a single location. If T is an interval then the answer will be a set of locations. This function can help find out where a person was or supposed to be at a given time.

- T.CorrLate (L) is another form of L.CorrLate (T) operation.

- O.CorrLate (L) is another form of L.CorrLate (O) operation.

- O.CorrLate (T) is another form of T.CorrLate (O) operation.

- L.CorrLate (O []) gives the association between a location and a set of identities. It returns the set of times (T []) that the given identities were at the specific location. This operation can help determine if multiple cars (O []) were parked in a parking lot (L) throughout the day. As another example, if the identities refer to people and it
returns the same time for all identities, then an inference that a group activity took place is likely.

- L.CorrLate (T []) gives the association between a location and a set of times. It returns the set of identities that were at location L and the given times. The time values to the function may map to a schedule, and the returned values may be the identities of the employees. The returned identities can then be matched against those related to a schedule to determine if the schedule is followed.

- T.CorrLate (O []) gives the association between a time and a set of identities. It returns the locations where the identities were at the given time. This function can help find out where a set of people were or supposed to be at a given time. In another instance, the function can be used to determine if a group of people were meeting.

- T.CorrLate (L []) gives the association between a time and a set of locations. It returns the set of identities that were present at the locations at the given time. This function can be used to determine if the lookout posts were manned by guards at a given time. As another example, the function can be used to determine if there were customers in the stores at a given time.

- O.CorrLate (L []) gives the association between an identity and a set of locations. It returns the set of times when the identity was observed at the given locations. This function can determine whether a guard is able to visit different locations as planned; the return values from this function should match the desired schedule. Alternately, an application can determine if a person visited some locations; the return values will be a non-empty set.

- O.CorrLate (T []) gives the association between an identity and a set of times. It returns the set of locations where the identity was at the given times. This function can help determine where a person was at different times. If an object such as a car in a parking lot has not moved, then all the location values returned will be the same. If the values returned are different, we can know about the mobility of an object.
The Crest system provides the correlate operation listed above as a minimum. An application is not limited to just the listed operations. It can define its own reasoning operations that take different arguments. For example, an application may define an operation of the form Correlate (T[], L[]). Similarly an application is not restricted to using the listed operations with their current semantics. It is free to modify the semantics of the operations by extending them using object inheritance. For example, an application may want to only return the first N matches for the L.Correlate (O) operation.

7.3 Implementation

The system is implemented as a runtime library that consists of the following components: database scheduler, participation protocol, communication system, table creation, event storage, and reasoning operations. The database scheduler implements the policy to assign an event store to an application. The participation protocol allows application entities to manage a dynamic interaction. The communication system allows the application entities to send and receive messages. The table creation module allows an application to create database tables in the event store for the reasoning classes (time, location, identity, event) that an application uses. The event storage module allows an application to store events that it may want to reason about later in the event store. The reasoning operations allow an application to reason about events using time, location, and identity.

The library is implemented in Java language because Java is object-oriented and supports heterogeneity. The notion of an event as a tuple consisting of time, location, and identity components naturally maps to objects. Thus an object oriented language is an ideal choice for the system implementation in contrast to a procedural language like C. Java runs on multiple hardware and software platforms, and thus supports the heterogeneity prevalent in pervasive computing applications. The heterogeneity support is the reason for our choice of Java as opposed to some other object-oriented language such as C++. The middleware uses Java Database Connectivity (JDBC) interface for event store operations, so any database that supports JDBC is usable.

We map the reasoning objects (time, location, identity, events) to relations and store
them in a relational database. Our implementation uses a PostgreSQL [8] database as the event store as PostgreSQL is freely available and has a JDBC driver. In contrast to our approach of mapping objects to relations, object-oriented databases [18] provide an alternative to storing events. Some of the object-oriented databases are IRIS [34], ORION [47], GEMSTONE [23], OZ+ [76], ODYSSEY [30]. Table 1 presents a comparison between object-oriented and relational databases. As we can see there is no absolute winner among the two, and any choice involves trade-offs. On the one hand, object-oriented databases can provide storage for normal objects, directly maintain object relationships, and easily navigate related objects. On the other hand, relational databases can perform better at database operations as objects of one class are in one table, have data independence due to simplicity in table structure, are easily available, have a standard language, and are widely used. In our case we went with object to relation mapping because of the following reasons: one, higher performance of database operations reduces the overhead of our system; two, standard language allows us to change the database (for example from PostgreSQL to Oracle) if needed; three, easy availability reduces a usage barrier; four, wide usage ensures greater familiarity.

7.3.1 Database Scheduler

As many applications can be expected to use the infrastructure, more than one database may be needed to support the scenario envisioned in the introduction (Chapter 1). The database scheduler takes as input a list of databases that are available for applications to use. Every time a new application starts, the participation server requests the database scheduler for an assignment. The scheduler chooses the next server to assign, based on some policy, and sends the information to the participation server. Currently our scheduler implementation follows a round-robin policy, but any policy can be used. To change the policy requires extending the class that implements the policy.

7.3.2 Participation Protocol

The participation protocol has two modules: server, and peer. The server module has a listener thread, a request queue, and a thread pool. The listener thread listens for all
Table 1: Table comparing object-oriented and relational databases

<table>
<thead>
<tr>
<th>OBJECT-ORIENTED DATABASES</th>
<th>RELATIONAL DATABASES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent objects appear as normal objects.</td>
<td>Tuples represent the image of an object.</td>
</tr>
<tr>
<td>Persistent objects have complete object relationships.</td>
<td>Relationship between values in tables model object relationships.</td>
</tr>
<tr>
<td>Efficient navigation between related objects via object references.</td>
<td>Navigation between related objects may require operations on multiple tables.</td>
</tr>
<tr>
<td>Objects of a class may scatter to different physical locations in a database, and slow down operations (query, update, etc.).</td>
<td>Objects of a class will be in one table, hence give higher performance for operations.</td>
</tr>
<tr>
<td>Data are tightly tied to the object design.</td>
<td>High level of data independence as tables are simple.</td>
</tr>
<tr>
<td>Can represent arbitrarily complex data.</td>
<td>Requires flattening of complex data into tables.</td>
</tr>
<tr>
<td>Extensible as new classes can inherit from exiting classes.</td>
<td>Harder to extend as there are limitations in inheriting tables.</td>
</tr>
<tr>
<td>No standard data manipulation language.</td>
<td>SQL is the standard data manipulation language.</td>
</tr>
<tr>
<td>Not easily available.</td>
<td>Easily available. Can even be freely downloaded.</td>
</tr>
<tr>
<td>Not widely used. Mostly a part of proprietary business applications.</td>
<td>Very widely used.</td>
</tr>
</tbody>
</table>

Incoming application requests (START, JOIN, LEAVE, STOP). Once the listener receives a request, it places the request on a queue. At any time a worker in the thread pool is either waiting for a request to arrive on the queue, or servicing a request. When a request arrives on the queue, a waiting worker picks up the request, checks the type, and appropriately services it. Once a worker completes servicing a request it goes back to waiting for the next request. The thread pool approach is in contrast to creating a new thread to service each incoming request. We chose to use a thread pool because it amortizes the cost of creating threads over many requests. Creating a thread for every request will add creation overhead for each request. In case the request servicing time grows too long, the thread pool can grow in size. If there is not enough work and most threads are waiting, then the thread pool can shrink in size. The participation server can grow or shrink the thread pool as needed.

The peer module provides application entities the interface to participate in an application. Participation operations are start, join, leave, or stop an application. For each
operation the peer module sends the appropriate message to the participation server, and obtains the server response and caches the data. In case of join the response data are the following: entity identifier, event store, and identifiers for other entities in the application. In case of leave the response data are identifiers for other entities in the application. As entities can join and leave at arbitrary times, there is a thread (control-message listener) for the join and leave messages from peers. On receiving a join message from a peer, the module adds the joining peer to the list of application entities it maintains. On receiving a leave message from a peer, the module removes the joining peer from the list of application entities it maintains. If a handler specific to the joining or leaving entity exists, the control-message listener invokes the handler. Instead of an entity-specific handler there can be an application-specific handler that the listener invokes. As an entity can participate in multiple applications, the peer module maintains all the participation information related to each application: entity identifier, the event store, and the control-message handlers that can be peer-specific or application-specific.

7.3.3 Communication System

Conceptually the communication system is orthogonal to the participation protocol; however in the implementation it is tightly integrated with the participation peer module. The reason is that the application name and the entity identifier for an application uniquely identifies an entity for communication, and this information is obtained from the participation protocol. To start communication a receiver sends a control message (initiate transfer) to the sender. The control-message listener thread at the sender extracts the receiver information from the initiate-transfer message. If an entity-specific or application-specific handler exists, the control-message handler thread invokes it. Using an application-level function the sender can then start data transmission. There is no matching application-level receive operation on the receiver side. There is a data-message listener thread that waits for any data on the receiver side. Whenever data arrives, the data-message handler thread invokes an entity-specific or application-specific handler, and passes the incoming data as an argument to the handler. To stop communication, a receiver sends a control
message (terminate transfer) to the sender. Once the sender receives the terminate-transfer message it stops transmitting to the particular receiver. As in the case of initiate-transfer message, the control-message handler thread at the sender invokes an entity-specific or application-specific handler, if such a handler exists.

7.3.4 Database Schema

There is a global table in every database that contains the event definitions. For example, Table 2 shows two event definitions, with EVENT class composed of TIME, LOCATION, and IDENTITY classes. Multiple event classes may have common component classes. For example, in Table 2 the event definition of EVENT1 shares two component classes with EVENT (TIME, IDENTITY). Each application-defined class (event, time, location, identity) maps to a table that stores the class instances. The name of the table is the same as the name of the application-defined class.

Table 2: Table to store event definitions

<table>
<thead>
<tr>
<th>DEFINITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Name</td>
</tr>
<tr>
<td>EVENT</td>
</tr>
<tr>
<td>EVENT1</td>
</tr>
</tbody>
</table>

There is a two-level structure to the tables related to each event class. At the leaf nodes are three tables, one each, for the time, location, and identity classes. Each row represents a specific instance of the class in the table.

At the root level is the event table that stores information about the individual component instances (time, location, and identity). Each event table row consists of six columns, with two columns each for the time, location, and identity instances. The two columns (entity id and unique val) for each component uniquely identify a row in the component table. For example, the entry for time columns in the event table points to some entry in the time table. Similarly the column entries for location, and identity point to entries in their respective tables. Thus, the entity id and unique val provide the mapping from event table to the individual tables.
An entity has to provide a unique identifier (entity id and unique val) for each of the time, location, and identity instances it creates. The unique identifier has two parts: identifier assigned by the participation protocol to the entity (entity id) when it joins an application, and a unique value generated by the entity itself (unique val). The entity identifier distinguishes among the entities that generate time, location, and identity instances. Thus, the entity identifier is same for all instances of time, location, and identity that are generated by a specific entity. The unique value distinguishes among the various time, location, and identity instances generated by a specific entity.

When an entity generates the unique identifier it does not have to synchronize with other application entities to store events. (Each entity can independently generate the unique identifier and store events.) In contrast, if individual entities did not generate unique identifiers, then a central authority would have to provide the mappings from event tables to component tables. Thus, multiple store operations would be serialized at the central authority, instead of progressing concurrently as they do now.

As an example, consider an application where the time, location, and identity are represented by classes TIME, LOCATION, and IDENTITY respectively. A sample set of tables for these classes may look as shown in Table 3, Table 4, Table 5, and Table 6.

Table 3: Table to represent class EVENT.

<table>
<thead>
<tr>
<th>EVENT</th>
<th>TIME</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTITY ID</td>
<td>UNIQUE VAL</td>
<td>ENTITY ID</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>35</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

An alternative to a two-level table structure for an event class is a single table that stores all the information for an event. For example, Table 7 represents the same information as Table 3, Table 4, Table 5, and Table 6 combined. (As Table 7 is only for illustration, we have excluded some of the columns in the individual tables.) There are many drawbacks to using a single-table structure. One, the system cannot reuse information. For example, if two
<table>
<thead>
<tr>
<th>ENTITY ID</th>
<th>UNIQUE VAL</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>250</td>
</tr>
</tbody>
</table>

Table 5: Table to represent class LOCATION.

<table>
<thead>
<tr>
<th>ENTITY ID</th>
<th>UNIQUE VAL</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>320 X St.</td>
<td>Atlanta</td>
<td>GA</td>
<td>USA</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>120 Y Av.</td>
<td>Portland</td>
<td>OR</td>
<td>USA</td>
</tr>
</tbody>
</table>

Events occur at the same location, we need only one entry in the location table, whereas the information will have to be repeated in the single-table structure. Two, there will be inefficient space utilization as information will be duplicated when events have overlapping component values. This inefficiency is not negligible because an application may deal with a huge number of events over time. (The second drawback is a corollary of the first.) Three, it will be harder to manage event information because in each event table we will have to keep track of how many columns refer to each of the time, location, and identity components. When each class is stored in its independent table, as in the two-level structure, the system does not have to keep track of the number of columns.

7.3.5 Table Creation

The system uses the Java interface mechanism for an application to specify the class attributes. For each class, an application provides the names and types of the attributes,

<table>
<thead>
<tr>
<th>ENTITY ID</th>
<th>UNIQUE VAL</th>
<th>First</th>
<th>Last</th>
<th>Eyes</th>
<th>Hair</th>
<th>Race</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>Sameer</td>
<td>Adhikari</td>
<td>Brown</td>
<td>Black</td>
<td>Asian</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>Arnab</td>
<td>Paul</td>
<td>Brown</td>
<td>Black</td>
<td>Asian</td>
</tr>
</tbody>
</table>
Table 7: Alternate table to represent class EVENT.

<table>
<thead>
<tr>
<th>TIME</th>
<th>LOCATION</th>
<th>IDENTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Val</td>
<td>Address</td>
<td>City</td>
</tr>
<tr>
<td>7</td>
<td>320 X. St</td>
<td>Atlanta</td>
</tr>
<tr>
<td>1</td>
<td>120 Y. Av</td>
<td>Portland</td>
</tr>
</tbody>
</table>

and the system creates the table. Currently the attribute types are restricted to those supported by relational databases. After the individual classes, the application informs the system about the event tuples. Based on all this information the system creates the tables in the database. As table creation only happens once for each class, it is not a big overhead. Any entity of the distributed application can call the create function. The application can ask the system to create the tables at any time. There is only one condition: reasoning operations involving any class take place only after the corresponding table creation.

For example, referring to Table 4 in case of class TIME the application informs the system that each entry has an attribute Value whose type is Integer. The table for LOCATION and IDENTITY classes are similarly created. Then, the table for EVENT class (Table 3) is created with the help of the system. Finally, an entry is made in the definition table (Table 2) that says that EVENT class consists of TIME, LOCATION and IDENTITY classes.

7.3.6 Event Storage

When an application invokes an event store operation, the system does the following. It checks if the given time, location, and identity values already exist in their respective tables. If they do not, it makes entries in the associated tables. If they already exist, it just extracts the unique identifiers (entry id, unique val). (For an explanation of the unique identifiers refer to Section 7.1.4.) For example, the time, and location values might exist, whereas the identity value may not. Then the system only inserts the identity value in the table for identity values. Next it checks the event table to see if the same entry already exists, using the unique identifiers it has from the individual tables. Only if this is a new instance, it makes an entry in the event table (pointing to the entries in the individual tables).
7.3.7 Reasoning Operations

Each reasoning operation maps to a multi-level query for the event database. Consider the case when an application correlates instances of TIME and LOCATION classes. The system queries the event-definition table to determine all the event classes that have the given TIME and LOCATION classes as components. By definition of an event class, the classes returned by the database will differ in the identity classes. For each event class the system does the following. First, it queries the event table using the unique identifiers (entity id and unique val) for the given TIME and LOCATION instances. Second, from the resulting events it extracts unique identifiers to entries in IDENTITY. Third, using the unique identifier values from the second step, the system queries the identity class to get the attribute value pairs. Fourth, it uses the attribute values to create instances of the particular identity class. After, the system has processed all event classes, it returns the identity instances. Note that the set of returned identity instances can potentially belong to different classes.

7.4 Summary

In this chapter we have presented the second of our two main contributions: Crest system to archive events and reason about them, using time, location, and identity. We have presented the universe model that Crest system assumes, its architecture components, and its implementation details. The universe model (Figure 13) consists of a set of participation servers and event stores, supporting a set of applications. Each application consists of a set of software entities working together. The architecture components are: events, reasoning operations, event store, participation protocol, and communication system. The implementation consists of a package of Java modules for the various architecture components. We present an evaluation of the Crest system in Chapter 9.
CHAPTER 8

CONVERGING WITH D-STAMPEDE AND CREST

Figure 17: Integration of D-Stampede and Crest in an application

Figure 17 shows how to integrate D-Stampede and Crest together in pervasive computing applications. An application entity can use D-Stampede to handle temporal streams by using a notion of time to index stream data. Sensors that constantly monitor the environment can provide streams of ephemeral data. From the stream data the application entity can generate event information that it can archive using Crest. The application entity can also use Crest to correlate current events with historical data.

An application entity has equal access to both D-Stampede and Crest; using one does not prevent the entity from using the other. Though an application may be structured such that some entities use D-Stampede, some use Crest, and some use both. This application structuring will help divide various tasks among nodes. Consider the generic distributed pervasive application in Figure 18 (same as Figure 5. The input entities connect to sensors, and the output entities connect to actuators. The sensors constantly gather data from the environment (cameras, temperature sensors, light sensors, etc.). Some of the actuators constantly modify the environment (e.g. thermostat). The input and output entities that control the sensors that constantly interact with the environment, can use D-Stampede
to pass the stream data to some of the processor entities. The processor entities that receive stream data can determine the events that occur (e.g. person enters or exits a room), and use Crest to archive the event information. Some of the processor entities can use the current events to drive the actuators (e.g. switch on the lights when someone enters a room). Some of the other processor entities can use Crest to correlate different events and affect application behavior. For example, start the computer with the same settings that a user had when he last logged off. Thus, we observe that for distributed pervasive computing applications, D-Stampeede and Crest provide powerful capabilities, handling temporal streams and reasoning about events respectively, that complement each other.
CHAPTER 9

EVALUATION

In this chapter we present an evaluation of the systems developed in this thesis: D-Stampede and Crest. The overall goal is to determine the suitability of our systems for pervasive computing applications. We evaluate the systems using both qualitative and quantitative metrics. For the qualitative evaluation, we focus on how a programmer can use the systems to build applications by discussing the applications we have built. The goal of the qualitative evaluation is to show the ease of programming, when our systems are used to build pervasive applications. For the quantitative evaluation there are two types of numbers: micro-measurements, application measurements. The micro-measurements quantify the cost of the interfaces each system provides. The application measurements study how the systems affect application performance.

9.1 D-Stampede: Handling Temporal Streams

9.1.1 Programming using D-Stampede

In this subsection we do two things. One, we describe a video-conferencing application we have built using D-Stampede. Video conferencing is an appropriate choice to study temporal stream handling in pervasive computing applications. Each participant in a conference continuously generates a video sequence, that is a temporal stream of images, for the life of the application. Further, each participant receives a continuous stream related to all the other participants. Thus, a video-conferencing application handles a plethora of temporal streams. This abundance of streams is characteristic of pervasive computing applications. Two, using the video-conferencing example we show how D-Stampede eases the development of similar applications.
9.1.1.1 Video Conferencing

Consider a video conferencing application. Conceptually, this application involves combining streams of audio and video data from multiple participants and sending the composite streams back out to the participants. To keep the discussion simple, we will only consider video streams. An implementation of this application using D-Stampede will have the following components:

- A server program on the cluster that has a set of channels (one per end device) for storing the video streams; a mixer thread that takes corresponding timestamped frames from these channels to create a composite video output; and a channel for placing the composite video output of the mixer thread.

- A camera and a display as end devices for each participant. A client program on each end device that is comprised of a producer thread that ‘puts’ its timestamped video stream on its assigned channel; and a display thread that ‘gets’ the composite video and displays it to the participant.

Figure 19 pictorially shows the structure of this application. When the server program is started on the cluster the following events happen:

1) The server program creates multiple address spaces $N_1, N_2 \ldots N_k$ in the cluster.
2) The server library spawns a listener thread in each address space.

3) The server program creates address space $N_M$, where a mixer thread is spawned and a channel $C_o$ is created.

4) The id of channel $C_j$ created by $j$-th client ($1 \leq j \leq m$) in $N_i$ ($1 \leq i \leq k$) is made available to $N_M$ via the nameserver.

5) The id of channel $C_o$ in $N_M$ is made available to each client via the nameserver.

The mixer thread does the following:

1) Create input connections to channels $C_j$ ($1 \leq j \leq m$).

2) Create output connection to the channel $C_o$.

3) Get correspondingly timestamped items (images) from each $C_i$.

4) Create the composite item (image).

5) Put composite item on channel $C_o$.

When the $j$-th client program is started on an end device, the following events happen:

1) The client library implicitly communicates with a listener thread in one of the address spaces $N_i$ ($1 \leq i \leq k$).

2) The client program creates a channel $C_j$ in address space $N_i$.

3) The client program starts a producer thread which creates an output connection to channel $C_j$ and puts images in $C_j$.

4) The client program starts a display thread, which creates an input connection to channel $C_o$ and gets composite images from $C_o$.

### 9.1.1.2 Ease of Use

D-Stampede eases the task of developing such an application by providing high level computational abstractions. As can be seen from the above example, the features in D-Stampede – such as specifying an arbitrary number of address spaces, creating any number of threads
in different address spaces, creating system-wide unique channels, transporting stream data that can be split in any arbitrary user-defined manner, time-stamping the data items, and time-correlating different data streams – come in very handy for developing such interactive stream-oriented applications.

To further highlight the ease of use, we contrast using D-Stampede with using TCP sockets, for writing this application. We had to do this as part of our performance evaluation. Below is a sample of the limitations we came across when trying to use TCP sockets.

- In TCP sockets, each client has to create and maintain connections to \( N - 1 \) clients. In D-Stampede a client has to only get items from a specific channel.

- The data arriving on each connection is an arbitrary set of bytes. In D-Stampede the application can put each frame as an item in a channel.

- Sometimes there is a speed mismatch between the data capture rates at a client, and the composite delivery rates. Instead of delivering all frames, the application may want to skip a few frames. In D-Stampede this can be done by just picking up desired items, and telling the system to discard the rest. In TCP sockets every frame will have to be built from the bytes, and only then can they be discarded.

- To generate the composite, we have to correlate frames from multiple sources. The timestamps in D-Stampede provide a natural way to do this. Trying to do this with the bytes of a TCP connection, will require code that will distract from the main task of the correlation.

### 9.1.2 Performance

We have carried out an experimental study to evaluate the performance of the D-Stampede system. We have measured the performance of the system at two levels. First, at the micro-level to determine the latency of D-Stampede operations, and the cost of communicating failure notifications. Second, at the level of a video-conferencing application that we discussed in the previous subsection. The micromeasurements and one set of application
measurements are on a homogeneous platform. One set of application measurements are on a heterogeneous platform.

The hardware setup for the homogeneous platform study is as follows: A cluster consisting of 17 eight-way SMPs interconnected by Gigabit Ethernet. Each processor is a 550MHz Pentium III Xeon. Each node in the cluster has 4GB RAM and 18GB SCSI disk. The nodes run RedHat Linux 7.1. Note that even in experiments that involve the client libraries for C and Java, one of the cluster node acts as an end device. This experimental setup ensures that the measurements reflect the impact of the D-Stampede software libraries and are not perturbed by the vagaries in the end device hardware capabilities.

For the heterogeneous-platform experiments some components that use the client libraries ran on an IPAQ H3800 (206 MHz Intel SA-1110 32-bit RISC, 64-MB SDRAM, 32-MB Flash ROM). The IPAQs use Lucent 802.11b wireless cards (Orinoco) to connect to the network, and use the WEP for security. The IPAQs run Familiar Linux distribution [1].

9.1.2.1 Micro Measurements

![Figure 20: Experiment 1](image1)

![Figure 21: Experiment 2, Configuration 1](image2)

The micro-level consist of four experiments. The first three experiments compare an end-to-end data exchange between two D-Stampede threads against a similar data exchange
using the underlying messaging layer. The intent is to show that the overhead incurred for the high-level data abstractions in D-Stampede is minimal compared to raw messaging. For raw messaging, this data exchange amounts to a send-receive combination. A put-get combination is the logical equivalent for D-Stampede. We conduct three experiments, each involving a pair of producer-consumer threads. In each of the three experiments we compare different configurations of producer-consumer pair performing a data exchange in D-Stampede against a commensurate send-receive based data exchange. In the graphs that correspond to these experiments (Figures 24, 25, and 26), the message size (in bytes) is plotted along the X-axis, and the latency (in microseconds) for the data exchange is plotted along the Y-axis. The readings shown are for data sizes ranging from 1000 to 60000 bytes, in steps of 1000 bytes. We restricted our readings to 60000 bytes because UDP does not allow messages greater than 64 KB. The fourth experiment evaluates the cost of communicating failure notification as a function of the number of clients interested in the notification. In the graph for this experiment (Figure 27), the number of clients to be informed is plotted along the X-axis, and the time taken to send the notifications along the Y-axis.

**Experiment 1:**
Figure 24: Intra-Cluster Results (Experiment 1)

In this experiment (pictorially shown in Figure 20) the producer and consumer threads are on different nodes (and therefore different address spaces) within the cluster. The channel used for communication is located in the consumer’s address space. The producer puts items on the channel. The consumer gets items from the channel. We orchestrate the experiment such that the put and get do not overlap. Latency is measured as the sum of the put and get operations.

This D-Stampede configuration is compared against two other alternative scenarios. One alternative uses UDP for communication and the other TCP/IP between the producer-consumer pair. Within the cluster, D-Stampede uses CLF, a reliable packet transport layer built on top of UDP. CLF does not have the connection management overheads of TCP/IP, and hence the two alternative scenarios being compared against D-Stampede. To ensure that the send and the receive do not overlap for the two alternative scenarios, we do the following: The producer sends a message and the consumer receives it. The consumer sends a message and the producer receives it. The exchange latency is assumed to be half the time taken for this cycle. The producer-consumer programs for this experiment are all written in C. The performance results are shown in Figure 24.

As can be seen, data exchange using D-Stampede adds an overhead (that ranges from 700 microseconds at 10 KB payload to 1200 microseconds at 60 KB payload) compared
to the UDP alternative. The overhead compared to the TCP/IP alternative is much less (starts from around 700 microseconds at 10 KB and with the increase of message size falls to 400 microseconds at 60 KB). The TCP/IP based producer-consumer measurements have some spikes that are due to the inherent congestion control properties of TCP/IP protocol. Overall, the overhead incurred for D-Stampede is fairly low at reasonably high payloads (less than 2X compared to UDP; at best almost the same or better than TCP/IP and at worst within 1.5X compared to TCP/IP).

**Experiment 2:**

This experiment involves the use of the C client library of D-Stampede. The producer thread runs on an end device as a client program. There are three configurations used in this experiment, each differing in the location of the consumer thread.

- **Configuration 1:** As shown in Figure 21, the consumer thread is co-located with the channel on the cluster. This configuration involves one “end device to cluster network” traversal. The get operation is local to the cluster node due to the co-location of the channel. Recall that the client library uses TCP/IP to communicate with the server library. Thus, this configuration shows the exact overhead that D-Stampede runtime adds to TCP/IP. For example, for a data size of 55000 bytes, TCP/IP latency is 2500 \( \mu s \), and D-Stampede latency is 3300 \( \mu s \).
**Figure 26:** Java based End Device and Cluster Results (Experiment 3)

- **Configuration 2:** As shown in Figure 22, the consumer thread is located on the cluster. The channel is in a different address space from that of the consumer. This configuration involves one “end device to cluster network” traversal (for the put operation) and one “intra-cluster network” traversal (for the get operation). Thus, this configuration is expected to have more overhead, compared to the previous configuration. For 55000 bytes payload, the D-Stampede latency is around 5000 μs.

- **Configuration 3:** As shown in Figure 23, the consumer thread is located on an end device. This configuration involves two “end device to cluster network” traversals, one for each of the put and get operations. Thus, this configuration has the maximum overhead. For 55000 bytes payload, the D-Stampede latency is around 6100 μs.

Since all the D-Stampede configurations used in this experiment involve the client library (which uses TCP/IP for communicating with the server library), we use a TCP/IP based producer-consumer pair written in C for comparison with the D-Stampede results. As before, we orchestrate the experiment to ensure that the producer and consumer do not overlap in their communication. The results for this experiment are shown in Figure 25. As expected, the shape of the D-Stampede curves track the TCP/IP curve for all the configurations. For configuration 1 (wherein there is only one “end device to cluster network” traversal) the D-Stampede runtime overhead over TCP/IP is fairly nominal (at best less
Figure 27: Failure Notification Results (Experiment 4)

than 12%).

**Experiment 3:**

This experiment is the same as the previous one with the only difference that the Java client library is used for D-Stampede, and the TCP/IP based producer-consumer program for comparison is also written in Java. The results for this experiment are shown in Figure 26. For a payload of 55000 bytes, the D-Stampede latency is approximately 11000 µs for configuration 1 (corresponds to Figure 21), approximately 12600 µs for configuration 2 (corresponds to Figure 22), and approximately 21700 µs for configuration 3 (corresponds to Figure 23).

The results of the three experiments are summarized below:

**Result 1:** For D-Stampede, intra-cluster data exchange (Figure 24) performs slightly better than C-based end device to cluster data exchange (Figure 25), which in turn performs better than Java-based end device to cluster data exchange (Figure 26). For example, the latency for a data size of 35000 bytes, is 2580 µs in the first case, 3200 µs in the second and 10700 µs for the third case. Since the hardware is the same for all these data exchanges, the disparity is simply due to the software. As we mentioned earlier, D-Stampede uses CLF for intra-cluster communication, which has less overhead compared to TCP/IP.

**Result 2:** For TCP/IP based data exchange, the C producer-consumer program results
(Figure 25), and Java producer-consumer program results (Figure 26) are similar. However, the D-Stampede data exchange is much better in C compared to Java. In C, marshalling and unmarshalling arguments involve mostly pointer manipulation, while in Java they involve construction of objects. Hence the disparity between the two.

**Experiment 4:**

In this experiment we create as many server address spaces as there are clients. Each client connects to one address space, and no two clients connect to the same server address space. Each client address space executes on the machine where the server address space that it connects to executes. The clients register their interest in receiving failure notifications for a channel created in one of the server address spaces. We kill one of the clients to simulate a failure and measure the time taken by the server to post the failure notifications to the remaining clients. The time measurement starts when the failure-notification-handler thread receives a client-failed message from a proxy thread. The time measurement ends when the failure-notification-handler thread completes sending out failure notifications to all the registered clients. We wish to point out that this is the time to send out the notifications in contrast to the time it takes the clients to get the notifications. The latency of communication will get added to these times, when considering the time for the clients to get the notifications. The results of this experiment are shown in Figure 27. The time taken to send notifications to the clients, is roughly linear in the number of clients. For example, the system takes 1844 $\mu$s to notify six clients 2172 $\mu$s to notify seven clients (approximately 310 $\mu$s per client).

**9.1.2.2 Application Level Measurements**

We use the application described in Section 9.1.1 for this study. We abstract out the camera and display from the application to make the study a controlled experiment for evaluating scalability with respect to the D-Stampede system. The producer thread in the client program reads a “virtual” camera (a memory buffer) and sends it to the server program continuously. Similarly the display thread in the client program simply absorbs the composite output from the mixer without really displaying it. This structure allows us
to stress the communication infrastructure of D-Stampede at the maximum possible rate
the application pipeline can generate video without interfacing to any external I/O devices.
Sustained frame rate is the performance metric of interest in this application.

We have developed three versions of this application. The first version uses Unix TCP/IP
socket for communication between the client programs and the server program. The mixer (a
single thread) obtains images from each client one after the other, generates the composite,
and sends it to the clients one after the other. The second version is almost the same as the
first version, except that it uses a D-Stampede channel between each client program and
the server program (i.e. in Figure 19, the mixer in address space $N_M$ is single threaded).
The third version has a multi-threaded mixer using D-Stampede channels (i.e. the mixer
in address space $N_M$ is multi-threaded in Figure 19). There is one thread in the mixer for
each client program. Each thread obtains an input image from the associated client, and
performs its part of the composite image generation. Once the image is fully constructed,
it is placed in the channel by a designated thread in the mixer for the client programs to
pick up for display.

![Frame Rate vs Data Size](image)

**Figure 28:** Single Threaded Version performance

In Figures 28 and 29, we have only shown readings when the sustained frame rate at the
display thread is higher than 10 frames/sec. We feel that any lesser frame rate at the display
thread than this threshold would be unacceptable for a video conferencing application.
Figure 29: Multi-threaded Version performance

Table 8: Delivered Bandwidth (MBps) as a function of data size and number of clients.

<table>
<thead>
<tr>
<th>Data size (KB)</th>
<th>Delivered Bandwidth (MBps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Clients</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>74</td>
<td>11</td>
</tr>
<tr>
<td>89</td>
<td>11</td>
</tr>
<tr>
<td>125</td>
<td>13</td>
</tr>
<tr>
<td>145</td>
<td>14</td>
</tr>
<tr>
<td>190</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 28 shows the performance of the first two versions (both single threaded), for two clients\(^1\), and for image sizes from 74 KB to 190 KB per client. This figure plots the sustained frame rate (on the Y-axis) as a function of the image data size (X-axis). As can be seen from the figure, the performance of the socket version\(^2\) and the D-Stampede channel version are comparable for the most part. For example, for a data size of 110 KB, they both deliver 18 frames/second sustained frame rate. This exercise proved two things: 1) Due to the complexity of this application, writing it using sockets required much more effort compared to D-Stampede, 2) The performance of D-Stampede version is comparable to the socket

\(^1\)The other configurations of the single threaded version that met our threshold are: a) 3 participants with images sizes 74 KB, 89 KB and 106 KB, b) 4 participants with image size 74 KB. No configuration with 5 or more participants met our threshold of 10 frames/sec.

\(^2\)We chose to implement the application using TCP/IP sockets for comparison with D-Stampede channels since other alternatives such as Java RMI [52] or CORBA [41] have significantly higher overhead.
version.

In Figure 29, the performance of the multi-threaded version is shown by plotting the sustained frame rate (Y-axis) as a function of the number of participants (X-axis). Each line in the graph is for a different client image size. Therefore, for an instance of the application with $K$ clients, the display thread at each client receives a frame $K$ times bigger than the client image size. The sustained frame rates measured at the different participants varied in a narrow band for each client image size. We chose the slowest display frame rate among the participants as the representative for each client image size.

Comparing Figures 28 and 29, we see that the multithreaded version performs better than the single threaded versions. For a client image size of 74KB, the single threaded version delivers approximately 20 frames/sec. In contrast the multithreaded version delivers approximately 40 frames/second. Clearly, thread parallelism in the mixer helps to boost the sustained frame rate seen at the display threads.

Despite the thread parallelism in the mixer, it can be seen from Figure 29 that the sustained frame rate achieved at a display thread is a function of the number of participating clients, the per-client image data size, and the available network bandwidth between the cluster and the end devices. For an image size of 74 KB, we see a frame rate of around 40 frames/sec for 2 clients, which drops to approximately 30 frames/sec for 3 clients. Similarly, for 2 clients, with an 89 KB image size, we get approximately 34 frames/sec, which drops to approximately 27 frames/sec for 125 KB image size.

In this version of the application, all the threads of the mixer run in one node (an 8-way SMP) of the cluster. Thus all the client display threads feed out of this one node to get the composite image data from the channel. The available network bandwidth at this cluster node is a complex function of the simultaneity of the requests from the display threads, the bandwidth of the memory subsystem, and other vagaries such as thread scheduling by the operating system. Since mixing is the most compute intensive operation in this application pipeline, it is highly likely that the requests from the display threads for the composite image are serviced simultaneously. Therefore, for $K$ clients, with a per-client image size of $S$, and a frame rate $F$, the required bandwidth at this cluster node is $K^2 SF$. This is because
each client receives a composite of size $KS$, and there are $K$ clients. Table 8 summarizes the actual delivered bandwidth from this cluster node for the various client image sizes and number of clients. This table is derived from the measurements which are plotted in Figure 29. From the table, it can be seen that the sustained frame rate falls below the 10 frames/sec threshold when the required bandwidth exceeds 50 MBps, suggesting that this is perhaps the maximum available network bandwidth out of the cluster node. This situation happens with 5 clients when the image size is 190KB, and 7 clients for the other lesser image sizes. We point this out to emphasize that the observed limit to the scalability is the application structure and not any limitation in the D-Stampede implementation.

**Table 9:** Delivered Frame Rate (fps) as a function of data size and number of wireless clients.

<table>
<thead>
<tr>
<th>Data size (KB)</th>
<th>Frame Rate Number of Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>7.63</td>
</tr>
<tr>
<td>96</td>
<td>1.86</td>
</tr>
</tbody>
</table>

**Table 10:** Delivered Bandwidth (KBps) as a function of data size and number of wireless clients.

<table>
<thead>
<tr>
<th>Data size (KB)</th>
<th>Bandwidth Number of Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>366.2</td>
</tr>
<tr>
<td>96</td>
<td>357.1</td>
</tr>
</tbody>
</table>

### 9.1.2.3 Heterogeneous Application Measurements

For the heterogeneous-platform experiments we use the multi-threaded mixer. The mixer runs on a cluster node, and the clients run on wireless IPAsQs. Table 9 shows the frame rate as a function of the data size and the number of clients. For example, for 3 cameras sending a 24 KB image each, the frame rate obtained at a display is 2.62 fps. Similarly, for 4 cameras sending 24KB image each, the bandwidth obtained at a display is 331.20 KBps. Table 10 shows the bandwidth as a function of the data size and the number of clients. The
bandwidth is directly derived from the frame rate.

For the homogeneous case we set a threshold of 10 fps for accepting a reading. In case of wireless we were not able to get a single reading above that threshold. The result can be seen if we compare the delivered bandwidths in Table 8 and 10. Whereas in the homogeneous case we notice saturation around 50 MBps, in the heterogeneous case the best bandwidth achieved is 366 KBps. The reason for the observed result is pretty straightforward. The wireless network is much slower than the cluster backplane. Further, for the wireless we are using Wireless Encryption Protocol, whereas there is no encryption in the wired case. Further, the processor and memory capacity of the IPAs is also much less. Here, as in the homogeneous case, the application performance is limited by factors outside of the D-Stampede implementation.

9.2 Crest: Reasoning about Time, Location, and Identity

9.2.1 Programming using Crest

In this subsection we discuss the ease of programming using a hypothetical surveillance application. Next we consider a complex application pipeline that is a generic representation of most applications in our domain.

9.2.1.1 Hypothetical Surveillance Application

Once again consider the scenario presented in the introduction (Chapter 1).

A video camera used for boundary surveillance in a high-security compound captures a car parked nearby on successive days. A program determines the duration of the car’s presence at the location to be a week long. The application then finds out that the car has been at various high-security locations over the past month. There are many things the application does next in parallel. One, it notifies guards, who have wireless handheld devices, to check the car. Two, it informs the applications in the other high-security compounds in the region to look out for the car. Three, it checks if this car was observed during some recent incident in the city. If it was, a police application then checks if there is anything suspicious about the car (reported stolen), or its owner (criminal record).

Consider an application that consist of the participating entities that represent the
following: high security locations, and guards. The entities use the participation protocol to join the application. The pseudo code for the application that implements the scenario may look as follows:

    // Define variables.
    CarIdentityClass Car;
    GuardIdentity Guards [];
    SecurityLocationClass ThisSecurityLoc, OtherLoc [], HighSecurityLocs [];
    DayClass CarTimes [], Week [7], Month [30];
    EntityIdentifier Entity [];

    // Initialize Variables.
    Days = last seven days;
    Month = last 30 days;
    HighSecurityLocs = high security locations in the surrounding region;
    Guards = identities of guards in this high security location;
    Car = car information from camera;

    // Find out the duration of the car at current location.
    CarTimes = Correlate (Car, ThisSecurityLoc);
    if (CarTimes < Week) return;
    // As car has been here for more than a week,
    // check where has it been in the past one month.
    OtherLoc = Correlate (Car, Month);
    if (OtherLoc ≠ HighSecurityLocs) return;
    // The car suspiciously has been to other high security
    // locations in the past month. So, take some action.
    Message = "'Car' is suspicious;
    // Send a message to some guards to check the car.
    for (i = 0; i < Some number of Guards; i++)
Entity [i] = Entity identifier for Guards [i];

Send (Entity [i], Message);

// Send a message to other high security locations
// warning them about the suspicious behavior.
for (i = 0; i < Number of HighSecurityLocs; i++)
    Entity [i] = Entity identifier for HighSecurityLocs [i];
    Send (Entity [i], Message);

9.2.1.2 Ease of Use

The security application that we have shown as an example shows the ease of using Crest. The high-level abstraction provided by the participation protocol, communication system, and reasoning operations aid application development. The participation protocol allows various entities to start and maintain the dynamic interaction. The correlate operations allow the application to draw inferences ("car has been here for more than a week") from the data. The communication system allows the entities to send messages ("car is suspicious") to one another.

9.2.1.3 Generic Application Pipeline

Next we describe the application skeleton (shown in Figure 30) that we have developed for measuring the system performance. It consists of three classes of nodes: producers, mediators, consumers. The communication topology is shown by the arrows connecting the various entities in Figure 30. The entity that starts the application, by informing the participation server, has no other role to play. Similarly, the entity that stops the application has no other role to play. As the names suggest, a producer generates input data, a mediator processes the data, and a consumer uses the data. All the producers, mediators, and consumers synchronize, using control messages, before the data transfer starts. In the steady state, each producer sends data to all the mediators, the mediator creates a composite of the data it receives from all producers, and each consumer receives
the composite from all the mediators. The number of producers, mediators, and consumers is variable.

Each producer entity does the following:

1) Sends a join message to the participation server.
2) Installs a control-message handler to receive the incoming control messages.
3) For all mediators, it waits for the control message to initiate data transfer.
4) Sends the data it generates to all the mediators.
5) For all mediators, it waits for the control message to terminate data transfer.
6) Sends a leave message to the participation server.

Each mediator entity does the following:

1) For each producer, it creates a queue to store the incoming data.
2) Sends a join message to the participation server.
3) Installs a data-message handler to receive the incoming data from the producers.
   The handler puts each incoming item in the producer-specific queue.
4) Installs a control-message handler to receive the incoming control messages from the consumers.
5) For all consumers, it waits for the control message to initiate data transfer.
6) For all producers, it sends out a control message to initiate data transfer.
7) While there is data to be processed the mediator does the following.
   a) It cycles through the producer queues to create a composite.
      In the $N^{th}$ cycle it picks up the $N^{th}$ item from every queue.
   b) It performs a correlate operation unrelated to the composite attributes.
   c) It sends the composite to each consumer.
8) For all consumers, it waits for the control message to terminate data transfer.
9) For all producers, it sends out a control message to terminate data transfer.
10) Sends a leave message to the participation server.

Each consumer entity does the following:

1) Sends a join message to the participation server.
2) Installs a data-message handler to receive the incoming data from the mediators.
3) For all mediators, it sends out a control message to initiate data transfer.
4) Then it receives all the incoming data.
5) For all mediators, it sends out a control message to terminate data transfer.
6) Sends a leave message to the participation server.

The main work in our application is done by the mediators: correlate operation, and composite creation. The composite creation represents some application processing of the data. The correlate operation is unrelated to the composite that the mediator creates. The correlate operation just goes to the event store and returns without any match. (We choose the parameters of the correlate function accordingly.) The reason for an unrelated correlation is that we later use this application to study the overhead of correlate operation on an application pipeline. The prototype application we have presented is a generic kernel for applications in our domains (surveillance, traffic management). For example, this prototype
can easily be used for a surveillance scenario by inserting processing modules at the producer, mediator, and consumer. At the producer a tracker module can generate information to send to the mediator. At the mediator a module can use the correlate operations can do more meaningful analysis. At the consumer a module that displays video or plays audio can help a user.

Like the hypothetical application we discussed, this application shows the ease of programming using a Crest. Here we have built a complex communication pipeline; the producers, mediators, and consumers are able to synchronize with each other.

9.2.2 Performance

In this section we present the measurements of various Crest operations and an application pipeline that uses Crest operations. The hardware setup for the following experiments is almost the same as the one used for D-Stampede. The experiments were run on nodes of a cluster consisting of 17 eight-way SMPs interconnected by Gigabit Ethernet. Each processor is a 550MHz Pentium III Xeon. Each node in the cluster has 4GB RAM and 18GB SCSI disk. The nodes run RedHat Linux 9.0 (7.1 for D-Stampede).

9.2.2.1 Participation Operations

The main participation operations are join and leave. To study the cost of the join and leave operations we perform the following experiment. When an entity joins or leaves it first contacts the participation server, and then all the other application entities. We measure the time it takes an entity to contact the server and the peers, when there are already some existing entities. We vary the number of existing entities from zero to nine. As the number of existing entities increases we expect the cost of informing them to increase. At the same time we expect the cost of contacting the server to remain constant, as it is independent of the number of existing entities.

Figure 31 shows the latency of sending the messages to the server and other peers for join and leave operations. In the case of join operation a value $N$ on the $x$ axis means an entity joining an application with $N - 1$ existing entities. In the case of leave operation a value $N$ on the $x$ axis means an entity leaving an application with $N$ existing entities. In
both cases (leaving or joining) the entity has to send messages to \( N - 1 \) entities apart from the participation server. When the \( 6^{th} \) entity joins an application with 5 entities, it takes the entity 10\( ms \) to inform the server, and 230\( ms \) to inform the other 5 entities. When an entity leaves an application with 7 entities it takes the entity 8\( ms \) to inform the server, and 305\( ms \) to inform the other 6 entities. The lines in the graph conform to the expected form. It takes nearly constant time for an entity to inform the server irrespective of the number of entities already present in the application. It take more time to inform \( N \) peers than to inform \( N - 1 \) peers whether an entity joins or leaves an application.

### 9.2.2.2 Communication Operations

The experiment to study the communication operations involved a producer and a consumer. The consumer on start up installs a handler to receive data from the producer. Then it sends a message to the producer to initiate the data transfer. Every time a data item arrives at the consumer, the handler is invoked by the runtime. The only work done by the handler is the bookkeeping to calculate the attained bandwidth. It does not process the message data in any manner. When the consumer receives all the items, it sends a message to the producer to terminate the data transfer. Figure 32 shows the bandwidth attained at the consumer. For example, for a message size of 50000 bytes, the bandwidth is 15 MBps. Here again we see expected behavior. The bandwidth delivered increases with message size, and
Figure 32: Bandwidth delivered to the consumer

saturates around 18 MBps.

In the application experiments for D-Stampe (Table 8, Section 9.1.2.2) we see that the cluster nodes are capable of delivering up to 50 MBps. The discrepancy between available (50 MBps) and delivered (18 MBps) bandwidth can be explained by the fact that in the former case we have C executables, and in the latter Java. Java has much greater overhead than C due to its partially interpreted execution, and automatic garbage collection. Although, just-in-time compilation prevents completely interpreted execution, the short life of our programs does not significantly reduce the overhead of interpretation. (In the micro measurements for D-Stampe we see that Java has much greater overhead compared to C (Result 2, Section 9.1.2.1).) Thus, the producer-consumer pair is not able to use the full available bandwidth.

Even though the delivered bandwidth is less than half of the available bandwidth, it is suitable for Crest. For the micro measurements, the producer and consumer exchanged raw bytes. In real applications, we expect the exchange to consist of events, because the focus in Crest is reasoning about them. With 18 MBps bandwidth, a producer-consumer pair will be able to exchange a number of events quickly (e.g. 100,000 events of size 180 bytes each).
9.2.2.3 Correlation Operations

The experiment to study correlation operations required creating an event store and then performing the operations on the event store. To create the event store we have done the following. We have created the time, location and identity classes with each class having a single integer-value attribute. By varying the value of the integer-value attribute from one to 100 for each class, we are able to enumerate a million different events. For each event, with two-third probability we decide to store the event. (That is, with one-third probability to not store the event.) Thus we end up with an event store of approximately 670,000 events. It takes more than seven hours to insert the events in the database.

The reason for choosing events in this manner is that we want a variable number of results when we execute the correlate operations. If we store all the million possible events, then every time we correlate an instance of one class (e.g. time) with an instance of another class (e.g. location), we would always get 100 instances of the third class (identity) as a result of the operation.

Next we perform the correlate operations. We perform 10,000 operations by enumerating all the possible combinations of 100 time instances and 100 location instances. Each correlate operation returns a certain number of identity instances as the result. Figure 33

Figure 33: Frequency distribution of correlate operations
shows the frequency distribution of the number of return values for the correlation operations. Consider a value \( Y \) on the y-axis and \( X \) on the x-axis. The pair \( (X, Y) \) means that \( Y \) out of the 10,000 correlate operations (involving an instance of time and location) return \( X \) identity instances as a result of the correlation operation. For example, 103 correlation operations return 57 identity instances. The correlation operations as a whole returns 46-85 identity instances.

![Correlate Latency](image)

**Figure 34: Latency of correlate operations**

Figure 34 shows the latency of the correlate operations. The latency is measured as the time from when the correlate operation is issued by an application to the time when the system returns the results to the application. The time spent by the application to process the results is not included in the measurement. The x-axis refers to the number of identity instances returned by the correlate operation. As can be seen in Figure 33, multiple correlate operations may return the same number of results. Thus, the y-axis shows the minimum, median, average, and maximum time taken by correlate operations that return the same number of results. For example, when correlate operations return 58 identity instances, the minimum time taken is 284 ms, the median time is 300 ms, the average time is 325 ms, and the maximum time is 526 ms.

In terms of trends, all the lines show expected behavior. That is, the greater the number of returned results, the greater is the time taken for the operation. Except at the boundaries
where there is only one correlate operation that returns a given number of results. \( X = 46, 48, 81, 83, 85. \) In these cases, the minimum, median, average, and maximum values are the same, and they distort the trends.

There are some other points to note about the readings. One, the time values suggest that the database is completely in-memory. The time taken to return the results for a correlate operation is less than what it takes to return results from disk. Two, the maximum time for a given number of results is about double the minimum, whereas the median and average values are close to the minimum. The reason for this wide variation most likely is that the measurements for the maximum time correspond to the moment when Java performs garbage collection.

### 9.2.2.4 Application Measurements

We use the application described in Section 9.2.1 and shown in Figure 30 for this study. We vary the number of producers (consumers) from two to ten, and the number of mediators from one to five. (The number of consumers is always equal to the number of producers.) The number of producers (consumers) and mediators define the instance of an application. The goal is to see how the correlate operations affect application behavior. The metric we measure is the bandwidth delivered at the consumers for each application instance. As there are multiple consumers in every application, we pick the median bandwidth value in each run. (In most cases the bandwidth values were similar up to two places after the decimal point.) The bandwidth reading pertains to two scenarios: no correlate operation performed by the mediator, and correlate operation performed by the mediator. In the correlation scenarios, each mediator performs a correlate operation before sending the composite to all the consumers. The correlate operation performed by the mediators can be thought of as a null operation. The inputs to the operation are chosen such that the operation just queries the database without returning any results. Further, the database is very small (very few events) to ensure that the query processing takes a very short time.

Figures 35, 36, 37, and 38 present the results of the experiments. Figure 35 shows the bandwidth as a function of the number of producers (consumers), when there are one
Figure 35: Application bandwidth for 1 and 2 mediators

and two mediators. The data are shown for both with and without correlate operations. For example, with seven producers (consumers) and one mediator, without correlation the bandwidth is 0.45 MBps, and with correlation it is 0.33 MBps. Figure 36 is the same as Figure 35, except that it deals with three and four mediators. For example, with six producers (consumers) and four mediators, without correlation the bandwidth is 1.37 MBps, and with correlation it is 0.59 MBps. Figure 37 presents the bandwidth as a function of the number of mediators, when there are five and six producers. The data are shown for both with and without correlate operations. For example, with three mediators and five producers (consumers), without correlation the bandwidth is 0.78 MBps, and with correlation it is 0.40 MBps. Figure 38 is the same as Figure 37, except that it deals with seven and eight producers. For example, with five mediators and eight producers (consumers), without correlation the bandwidth is 1.50 MBps, and with correlation it is 0.72 MBps.

The obvious conclusion that can be drawn from these graphs is that adding a correlate operation in an application pipeline adds overhead and decreases bandwidth delivered at the consumers. In Figure 35 and Figure 36, as the number of producers (consumers) increases, so does the bandwidth when mediators perform correlation. The reason for this behavior is that a mediator performs one correlate operation before sending a composite to all the consumers. In essence, the cost of the correlate operation is amortized over more
**Figure 36:** Application bandwidth for 3 and 4 mediators

communication operations. The decreasing cost of correlation with increasing number of producers point to the convergence of the lines for data with and without correlation. In Figure 37 and Figure 38, as the number of mediators increases, so does the bandwidth when the mediator performs correlation. Even though, an increase in mediator numbers means more correlation, at the same time it also means greater parallelism in communication with consumers. Potentially when some mediators are performing correlations, others are sending data to the consumer. Further, the lines for bandwidth with correlation track the behavior of the corresponding lines without correlation though with lesser values.

### 9.3 Summary

In this chapter we have evaluated the D-Stampede and Crest systems. To qualitatively study D-Stampede, we describe a video-conferencing application, and contrast it with developing the same application using sockets. In the video conference, producers generate streams, send them to a mixer that creates a composite, and the consumers display the composite. The producers, mixer, and consumers use the D-Stampede *channels* to communicate stream data. To quantitatively study D-Stampede, we measure the performance of the *put, get* operations for a channel under different configurations. We also study the performance of the failure notification mechanisms. Further, we measure the performance of the video-conferencing application by varying the number of participants, and observe the frame rate
Figure 37: Application bandwidth for 5 and 6 producers

that the application delivers, on homogeneous and heterogeneous platforms.

To quantitatively study Crest, we describe how one can build a hypothetical surveillance application. Further, we describe a surveillance-application skeleton we have built. The skeleton consists of multiple producers, mediators, and consumers, connected by a many-to-many topology. The producers generate data streams and pass them on to the mediators. The mediators mix the streams, perform correlate operations, and pass them on to the consumers. The consumer are responsible for the stream output. To quantitatively study Crest, we measure the performance of the participation, communication, and correlation operations. For participation, we measure the latency of the join and leave operations as a function of the number of application entities. For communication, we measure the bandwidth that a producer-consumer pair delivers as a function of the message size. For correlation, we measure the latency of a correlate operation as a function of the number of matches for each call. Further, we measure the application performance by varying the number of producers, mediators, and consumers, and observe the bandwidth that the consumers attain. The qualitative and quantitative study of D-Stampede and Crest shows that they are suitable platforms for developing distributed pervasive computing applications.
Figure 38: Application bandwidth for 7 and 8 producers
CHAPTER 10

CONCLUSION

In this chapter we conclude by describing the contributions of this dissertation and directions for future research.

10.1 Research Contributions

In this dissertation we started with a survey of projects in the domains of surveillance, traffic management, and mobile commerce. The results of the survey are that the applications are not really distributed and pervasive, they handle their live stream data in an ad-hoc manner, they do not organize historical data, and at best perform limited, application-specific reasoning on current event data. The survey showed the need for systems to help distributed pervasive applications handle two fundamental requirements. One, an ability to access continuous live stream data from multiple sources using time as an index. Two, an ability to organize historical data to reason about events, using time, location, and identity. Further, we have qualitatively and quantitatively evaluated the two systems.

10.1.1 D-Stampede

We have developed the D-Stampede system to handle temporal streams in distributed pervasive applications. D-Stampede assumes a particular hardware model that is representative of the prevalent computation and communication continuum, and provides a computational model. The hardware model is an Octopus (Chapter 6, Figure 8), and the computational model is a dynamic thread channel graph (Chapter 6, Figure 9) that maps to various distributed pervasive applications. The architecture of D-Stampede consists of the following components: Channels, Queues, Threads, Garbage Collection, Handler Functions, Complex Data Structures, Real-time Synchrony, Nameserver, Fault Tolerance, and support for Heterogeneity. Threads are basic mechanism for executing tasks, and can be created anywhere in the system. Channels and queues are system-wide unique containers that allow threads
to put and get items using a time index. D-Stampede automatically performs distributed
garbage collection of time-sequenced items based on information that a thread provides.
A thread can use handler functions with channels for two reasons. One, to automatically
execute serialization and de-serialization routines for transporting items. Two, to notify a
thread about items that have become garbage.

D-Stampede provides a generic mechanism to help threads transport complex data struc-
tures through channels and queues. Real-time synchrony helps threads to pace themselves
with respect to clock time. The nameserver allows threads to exchange information with
each other; there is no restriction on the type of information. D-Stampede supports fault
tolerance in two ways. One, if one component fails, the remaining components are able
to continue. Two, among the remaining components, those that were interacting with the
failed component, are informed about the failure. D-Stampede accommodates heterogeneity
of components at the operating system (Linux, Unix, Windows) and programming language
(C, Java) level. An application can have components in various combinations of operating
systems and languages, and they all are able to work together.

10.1.2 Crest

We have developed the Crest system that allows applications to archive events, and to
reason about them using time, location, and identity. Crest assumes a universe model
for distributed pervasive computing applications, and its architecture enables the model.
The universe model (Chapter 7, Figure 13) consists of a set of participation servers and
event stores, supporting a set of applications. Each application consists of a set of software
entities working together. The architecture components are: events, reasoning operations,
event store, participation protocol, and communication system. An event is a 3-tuple and
consists of time, location, and identity information. Time, location, and identity refer to
the following: when did it take place, where did it take place, and who or what was involved
in it. An application is free to define any type of time, location, or identity, and hence, any
type of event.

The reasoning operations allow an application to correlate events using time, location,
and identity. An event definition in an application can be thought of as a three-dimensional space. (Each event instance is a point in the space.) A correlate operation takes values in two dimensions as arguments, and returns values in the third event dimension. The correlate operation works in three steps. One, it determines the planar area defined by the values in the two input dimensions. Two, it determines the cluster of points in the three-dimensional space that projects onto this planar area. Three, if there is a projection, it returns values in the third dimension from the cluster as a result of the operation.

The participation protocol enables application entities to create and maintain a dynamic interaction. When an entity wants to join an application, it first contacts the participation server. The participation server assigns an identifier to this joining entity, and also gives it information about all the other entities in the application. Using this identifier the entity may contact the current application entities, and inform them that it is joining the application. The mechanics, but not the semantics, of the leave process are the same as of the join. An entity may join multiple applications; it might even join the same application multiple times.

The events that an application reasons about are archived in an event store. There can be multiple event stores in the universe, and each participation server knows about at least one event store. When an application starts, it is assigned an event store by the participation-protocol server. The system provide asynchronous communication between entities. To begin communication a receiver must install a handler, and notify a sender to start data transmission. Whenever a message arrives at a receiver, the system invokes the sender-specific handler. Communication ends when the receiver notifies the sender that it is no longer interested in receiving data.

10.1.3 Evaluation

We have qualitatively and quantitatively evaluated D-Stampede and Crest. To qualitatively evaluate D-Stampede we describe our experience in building a prototype video-conferencing application. In the video conference multiple producers are connected to a mediator, and it is in turn connected to multiple consumers. To qualitatively evaluate Crest we describe
our experience in building a skeletal surveillance application. In the surveillance application multiple producers, mediators, and consumers are connected in a many-to-many topology. Based on our experience, we argue that it is easier to build distributed pervasive computing applications using the abstractions provided by D-Stampede and Crest.

We take micro and application measurements to quantitatively evaluate D-Stampede and Crest. The micro measurements measure the cost of the functions provided by our system abstractions in isolation. The goal is to find the minimum overhead that our system imposes on applications. The measurements involve the same applications that we use for qualitative evaluation of the system. In D-Stampede the number of producers and consumers defines a configuration. In Crest the number of producers, consumers, and mediators together define a configuration. For each application we measure the bandwidth that each consumer attains for different configurations. The goal is to see how an application performs, when it uses our system. The evaluation of D-Stampede and Crest shows that they are suitable platforms for developing distributed pervasive computing applications.

10.2 Future Work

Our current work can be extended in multiple directions: correlate operations, application study, new features.

10.2.1 Correlate Operations

Currently the correlate operations return all results that match the simple input criteria. A possible avenue for exploration is making the correlate operations return a subset of the matches. An application may specify more complicated criteria for selecting the subset. Some example criteria are as follows: only $X$ percent of the results that are closest to the cluster center, values that are not more than $Y$ units away from the center, first $N$ values that are closest to the median value, values that are less than distance $D$ from the mean value, values that lie in the volume given by a sphere with center $(T, L, I)$ and radius $R$, and possibly many more.

To return a subset of values will require at least the following: determine the criteria that are important to applications, a method to allow applications to specify how to compare
values in event space, and a mechanism for applications to specify the criterion to apply for a correlate operation.

10.2.2 Application Study

Another avenue for exploration is developing full-fledged applications that use our system, and studying their performance. For example, we can develop a real surveillance application that uses sensors for input, executes various algorithms on the data, and uses actuators for output. (Currently, we have used application pipelines to study our system.) Once we have real applications using our system, we can study their performance. The study will help determine whether our system is ready for the real world. If not, the study will help us figure out bottlenecks and limitations in our system. The easiest way to proceed is to take existing applications, developed by others, and port them to our system. Porting existing applications has the advantage of providing ready-made benchmarks. We can compare the performance of the applications, for the cases that use our system, versus those that do not.

![Surveillance Application Diagram]

**Figure 39:** Surveillance Application

Figure 39 shows a sample surveillance application. It consists of sensors that survey the environment. The sensors can be of various types: cameras, microphones, motion...
detectors that use ultrasound, presence detectors that use infrared, and many others. A set of sensors is controlled by an entity that uses the D-Stampede mechanisms to pass data streams to processor entities. (There are many such sensor sets in the environment.) There are various processor entities that differ from each other in their roles. Some of the entities are generator, classifier, analyzer, tracker, and controller. A generator entity creates event information from the data stream. A classifier entity classifies the events according to some application-defined criteria. A tracker entity performs object tracking using information about events. A tracker can follow an object based on color, or shape, or some other feature. An analyzer tries to determine the behavior of the surveyed entities. For example, whether a person has been returning to a given location every day at a specific time. A controller entity may manage the flow of information among the application entities. For example, it may try to control the data rate of sensors. The processors we have listed are not exhaustive; the key point is that an application can consist of arbitrary types and number of processors. The common theme among processors is that they reason about historical and current event data to drive application behavior. For their reasoning the processors use the mechanisms that Crest provides. The processor entities then pass on information to other entities that control the output devices. The output devices can be of various types: displays, speakers, alarms, and many others. The output devices are used to send signals to human observers, to take some actions, as defined by the application. Some output devices (displays) get continuous stream data using D-Stampede mechanisms.

An application, such as that we have just outlined, will be invaluable for the study of D-Stampede and Crest due to two reasons. One, it will thoroughly test the ease of programming that we have aimed for in our work. It will help determine if the abstractions of D-Stampede and Crest are suitable and sufficient for the applications. Two, it will also thoroughly determine whether our system has acceptable performance for real applications. There are various ways to measure the performance overhead of our system, when we port a real application. For example, the reduction in the frame rate at an operator display. Another example, is the increase in the latency to raise an alarm.
10.2.3 New Features

Naturally, D-Stampede and Crest do not meet all the requirements of pervasive computing applications. We can extend our systems to tackle some other requirements such as mobility, disconnection, security, privacy, and many more. Mobility impacts the routing of information, and application topology may have to change to accommodate the effects. Sometimes a user might not be in a well-connected environment, and a pervasive application has to adapt accordingly. Maybe the application can reduce its functionality, or cache changes to take care of disconnection. Pervasive availability also implies pervasive vulnerability. Sometimes an application may execute over untrusted networks and hosts. Thus, maintaining security in pervasive environments is a challenge. A pervasive computing environment means continuous monitoring of people. Thus, the question of privacy takes on paramount importance. The programming idioms and runtime mechanism that will allow real-world pervasive applications to tackle mobility, disconnection, security, and privacy issues are future challenges.
REFERENCES


