# A Distributed, Object Oriented Architecture for Pattern Analysis

Brian Shand

Submitted to the Department of Electrical Engineering,
University of Cape Town, in partial fulfilment of the requirements
for the degree of Master of Science in Engineering.
September 1999

To Olivia

# Declaration

I, Brian Ninham Shand, declare that this dissertation is my own work. It is being submitted for the degree of Master of Science in Engineering at the University of Cape Town. It has not been submitted before for any degree or examination at this or any other university.

B. N. Shand

# Abstract

This thesis presents a new distributed object architecture that allows pattern analysis to be shared among many computers.

Techniques of pattern recognition, probabilistic decision making, and distributed computing are examined as foundations for the architecture.

The architecture employs both active and passive object migration between computers on a network. It is also fully object oriented, allowing objects to dictate novel migration strategies independently. Another important feature of this architecture is a new technique which allows objects to be truly distributed, by enabling them to reside in many places simultaneously.

The advantages of this distributed architecture include lower computing costs, enhanced fault tolerance, and fast real-time information processing. A battery of tests was developed to confirm that the architecture performed according to its specifications.

The 'Smart Building' is an application designed to demonstrate the system. It uses distributed objects to reconstruct workers' behaviour from movements sensed by cameras inside its rooms. Distributed objects in the 'Smart Building' mirror reality: as people in the building move between rooms, distributed objects migrate between computers. Thus, the 'Smart Building' takes advantage of the geographical proximity of interrelated data sources, to enhance efficient distributed pattern analysis.

# Acknowledgements

I would like to thank the following for their contribution towards this thesis.

Professor Gerhard de Jager, my supervisor, for his enthusiasm and guidance.

DebTech and the National Research Foundation for financial support.

The UCT Digital Image Processing group for their friendly encouragement.

My family for their unflagging interest.

Sun and Java are registered trademarks of Sun Microsystems, Inc.

Microsoft, C++, Win32, Windows and Windows NT are registered trademarks of Microsoft Corporation.

# Contents

# List of Figures

# Part I

# Introduction and Literature Review

# Chapter 1

# Introduction

The objective of this thesis is to present a distributed object architecture that allows pattern analysis to be shared amongst many computers — as demonstrated by a 'Smart Building'

Real time pattern analysis requires high speed computers, with high bandwidth connections to the data sources. Analysing this data centrally requires an immensely powerful computer, and a very high speed network; this is the traditional solution. However, distributing the processing of information to each data source would enable a simple network of smaller computers to perform the same task at a lower cost — and with a reduced processing delay[1].[1]

Object-oriented design allows the information processing to be delegated to the objects themselves. Furthermore, a group of distributed objects on different computers can act together, as if they were a single object, using a technique developed in chapter 8. This allows a single task to be shared by many computers.

An architectural specification defines the environment which distributed objects inhabit. It clearly defines the respective responsibilities of the distributed object architecture, and the objects within it. Because the design is object-oriented, the architecture primarily facilitates interactions between objects. Since each distributed object is able to define its own migration strategy, novel distribution techniques are made possible, without needing to modify the architecture.

A 'Smart Building' is a concrete example of the usefulness of distributed pattern recog-

[1] Distributed systems reduce delays by eliminating the communication lag of centralized systems, for local computations.

nition. A Smart Building is a computerised building that is aware of what is happening inside it. It can use cameras to follow the movements of the people in the building, and other devices to assist them wherever possible. Since most activity within a particular room of the building does not affect the other rooms, each room can have its own computer performing most of its data analysis. The rooms then need to communicate with each other only when a person moves between rooms. This can all be implemented using a distributed object architecture — thus proving the usefulness of the architecture for distributing pattern analysis.

Part I of this dissertation surveys concepts in pattern recognition and distributed computing, which form an essential background for developing a distributed object architecture. It begins with a literature survey of pattern recognition techniques, with special emphasis on symbolic techniques which can be used for high level abstractions, in chapter 2. Chapter 3 is an introduction to probabilistic techniques, which enable a computer system to reason with inexact information. Finally, chapter 4 explores the objectives and requirements of distributed computing.

Part II outlines the development of a distributed object architecture. This environment enables distributed objects to cooperate, in recognising patterns of information. In addition, a suite of experiments are used to test the architecture which was developed.

Part III presents a simulated building which aims to provide a rich and powerful source of data for the Smart Building.

Part IV demonstrates the strength of the architecture developed in part II in a real application. Using the techniques of part I, and the building simulator of part III, it gives a concrete example of distributed pattern analysis, in a computerised Smart Building.

# Chapter 2

# Pattern Recognition

In this chapter, techniques of data analysis and pattern recognition are explored, in order to evaluate their suitability for distributed pattern recognition.

Computers can react sensibly to the information which they receive only by correlating it with information which they already have. This requires a system of knowledge representation, and a way to compare old and new information. To do this, many techniques have been developed, each with its own particular area of application.

Context free grammars offer a syntactic approach to pattern recognition, by describing complex patterns through a set of reduction rules. The computer language Prolog is designed around these rules, and it is a standard language of Artificial Intelligence (AI) research.

At a lower level, statistical techniques are often used for pattern recognition; they directly reflect the probabilistic correlation between the data input and the resulting recognition.

Apart from the techniques which explicitly use statistical data, there are other techniques which draw implicit associations: neural networks 'learn' to associate example outputs with example inputs, by building an internal model which attempts to represent the correct data dependencies. Fuzzy logic and stochastic search algorithms provide alternative internal data representations, which can be optimized to reflect a set of training data.

## 2.1 Context Free Grammars

Context free grammars (CFG's) can be used to recognize certain types of pattern. A simple system of rules is used to describe which patterns a grammar will recognize. For example, the following grammar [2, p. 32] describes simple arithmetic expressions:

$$\begin{aligned} expr &\rightarrow expr + term \quad | \quad expr - term \quad | \quad term \\ term &\rightarrow term * factor \quad | \quad term \,/\, factor \quad | \quad factor \\ factor &\rightarrow \textbf{digit} \quad | \quad (\, expr\,) \end{aligned}$$

The first symbol in the grammar is known as the start symbol, and legal sentences in the language can be derived only by iteratively expanding non-terminals (*expr*, *term*, *factor*) according to the production rules. For example, the following set of productions proves that **9 − 2 ∗ 3** is a valid sentence in this grammar:

$$\begin{aligned} expr &\rightarrow expr - term \\ &\rightarrow expr - term * factor \\ &\rightarrow expr - factor * factor \\ &\vdots \\ &\rightarrow \textbf{9} - \textbf{2} * \textbf{3} \end{aligned}$$

One of the greatest achievements of context free grammars and machine understanding is the compiler, which understands the structure of a computer language sufficiently to translate it from a human-readable form into machine code for running on a computer.

CFG's are extremely useful for recognizing well-formed patterns, with a high degree of structure. However, they cannot easily detect approximate patterns, or patterns close to the sentences in the grammar. Even compilers are very ineffective at detecting errors, and worse at correcting them automatically. They are therefore useful for abstract problems, where information is represented symbolically, but are seldom used directly on lower level sensor data.

## 2.2 Prolog

Prolog[1] is a computer language designed primarily for Artificial Intelligence (AI) research, and automated theorem proving [3],[4].

Prolog programs are based on a similar rule-based structure to CFG's; for example, the following program recognizes the same sentences as the grammar given above.

```
expr(E)  :- append(X,[+|Y],E), expr(X), term(Y).
expr(E)  :- append(X,[-|Y],E), expr(X), term(Y).
expr(E)  :- term(E).
term(E)  :- append(X,[*|Y],E), term(X), factor(Y).
term(E)  :- append(X,[/|Y],E), term(X), factor(Y).
term(E)  :- factor(E).
factor(E):- E = [X], digit(X).
factor(E):- append([<|X],[>],E), expr(X).
digit(X) :- X=0.  digit(X) :- X=1.
digit(X) :- X=2.  digit(X):-X=3. digit(X):-X=4. digit(X):-X=5.
digit(X) :- X=6.  digit(X):-X=7. digit(X):-X=8. digit(X):-X=9.
```

When the program is compiled, the following tests demonstrate that it corresponds to the grammar:

```
?- expr([9,-,2,*,3]).
yes
?- expr([5,*,<,6,+,7,>]).
yes
?- expr([5,6,+]).
no
```

In this program, the brackets are represented using < and >, since round brackets have intrinsic meaning for Prolog. These tests confirm that $\mathbf{9} - \mathbf{2} * \mathbf{3}$ and $\mathbf{5} * (\mathbf{6} + \mathbf{7})$ are both valid sentences in the grammar, while $\mathbf{5}\,\mathbf{6} +$ is not. This example also demonstrates how Prolog operates; a program consists of a collection of assumptions, which Prolog will assume to be true. The user then makes a statement, and Prolog decides whether

[1]Prolog stands for 'Programming in Logic', and was developed by Alain Colmerauer and Phillippe Roussel, from the Artificial Intelligence Group at the University of Aix-Marseille, together with Robert Kowalski from the University of Edinburgh.

the statement is true or false, based on the assumptions. Thus, once a program has been written, interaction with Prolog consists of a session of questions and answers.

A statement in Prolog is written as `result:-condition` — if `condition` is true, then `result` will be true too. Thus the statement in line 9 '`digit(X) :- X=0.`' declares that, if `X=0`, then `X` is a digit. Similarly, the statement in line 1 states that, if `E` consists of an expression followed by a plus followed by a term, then `E` will be an expression.

Prolog can recognize any context free grammar, but it can also solve a far wider range of problems — in fact, it is as general as C++ or any other programming language. The difference between languages is how elegantly different problems can be solved in them. For example, Prolog is very effective at recognizing patterns in syntactic data, while Matlab is most effective at problems which require vector arithmetic and matrix manipulation.

The greatest difficulty of programming in Prolog is firstly to guarantee that the given problem has a solution, and secondly to ensure that Prolog will find the solution in a finite amount of time. If the program were incorrectly designed, Prolog could explore an infinite number of dead-ends before reaching the correct solution.

Although Prolog decides only whether statements are true or false, it is possible to use 'side-effects' to generate extra information during the search for solutions. For example, the following statement will discover all expressions consisting of exactly five symbols.

```
?- length(E,5), expr(E), write(E), fail.
[0, +, 0, *, 0][0, +, 0, *, 1][0, +, 0, *, 2] ...
... [<, 1, >, /, 9][<, 2, >, /, 0][<, 2, >, /, 1] ...
... [<, <, 8, >, >][<, <, 9, >, >]
no
```

This shows some of the power of Prolog not evident in the earlier examples: Prolog's 'resolution' strategy for analyzing statements allows it to work backwards as well as forwards towards a goal — it can **generate** solutions, as well as testing them. In the same way, it can find patterns in many forms of data — or generate patterns which match the data.

This power is one of the reasons why Prolog-related languages are still used for AI research. However, Prolog is seldom used commercially because of the difficulties of guaranteeing output, and the rigidity of its formalism which makes Prolog programs hard to write or understand.

## 2.3 Statistical Pattern Recognition

Statistical pattern recognition is the classical method of pattern recognition. An item of data is converted into a 'feature vector'[2], which is then classified as belonging to a particular class. For example, a patient could be analyzed using two tests, to decide whether he/she is suffering from a particular disease. The results of a number of these tests could be plotted in two dimensions on one graph, with a symbol to indicate whether or not each patient actually had the disease. Then the test space could be partitioned according to the correspondence between test results and the presence of disease.

Figure 2.1 shows how a sample data set could be plotted in this situation. Here the class of each data point is shown using either a filled circle or an empty diamond. For this data, the dashed line clearly separates the two classes, and therefore it could be used to classify the data points. However, for real data, the best division between classes is seldom so simple.

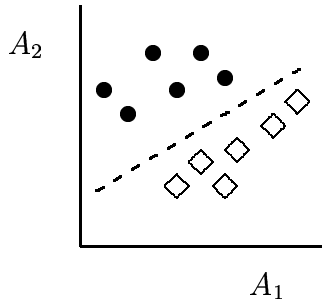



Figure 2.1: A simple classification problem

For some problems, such as image processing or face recognition [5], the feature vector has a very high dimensionality — every pixel in the image is an entry in the feature vector. In that case, the features are often projected into a lower dimensional space first [6], and classification is then performed on these new features.

Traditional pattern recognition methods all attempt to partition the feature space to produce the best possible classifications. Solutions include using hyper-planes to divide the space, classifying the regions near each point according to that point's classification ('nearest-neighbour classification'), and selecting a weighted subset of the features according to their statistical significance in classification. Weiss and Kulikowski [7] have written a review of the most commonly used statistical techniques of pattern recognition and machine learning, and of neural networks. The seminal justification of these

[2] A feature vector is an ordered list $(v_1, v_2, \ldots, v_n)$, with each element $v_i$ chosen from a predetermined set $A_i$

methods is by Duda and Hart [8].

## 2.4 Neural Networks

Neural networks offer a method of training a classifier directly, without explicitly deciding which of the features are most important. A neural network uses a mechanism analogous to that of a biological neuron in order to associate a given stimulus with a given response [9]. The most commonly used neural network is the multilayer perceptron, which is a feedforward network.

If the neural network is trained with enough data, it learns to produce correct or nearly correct responses. To do this, it adjusts the weights of its internal structure. Then, when the network is presented with data outside of the training set, the internal model calculates a response.

In an ideal case, the internal model correctly represents the essence of the desired correlation, and both training data and new data are correctly matched to responses. Because of the many weights in a typical neural network, and local linearities, enough training usually provides a good approximation to the correct model. Since neural networks can be automatically trained, and are reasonably efficient once trained, they are often used when a large amount of training data is available and when the underlying physical model is uncertain[3].

One disadvantage of neural networks is that they have an intractable internal model: the internal weights cannot be explained, nor can they be shown to correspond to any physical process. As a result, it is impossible to translate the internal model of a neural network into a comprehensible description, and there is no guarantee that a neural network will work consistently on all new data. For this reason, neural networks are sometimes used together with a traditional secondary control system, for safety.

In summary, neural networks provide a very general model which can be used in many classification problems. However they usually need time-consuming training on pre-classified data, and their internal models cannot be scrutinized; this limits their widespread adoption.

Support vector machines are a relatively new class of feedforward network. They require considerably less training time than traditional multilayer perceptrons [11, p. 345], and produce very general results, but their speed of operation can be slower than that of

[3] For example, neural networks are often used in face recognition applications [10]

ordinary neural networks.

## 2.5 Stochastic Search

A significant disadvantage of traditional methods of pattern recognition is that a large set of pre-classified training data is needed to produce the classification model.

Conversely, context free grammars and many other formal techniques can analyze only those problems which are phrased as a collection of rules. This makes it difficult for linguistic constraints to be combined with training data into a single model; although traditional models can be converted into a system of rules, the system is usually large and unwieldy, and too complex to be interpreted directly by a person.

Stochastic search techniques offer a method of combining training data and pre-specified rules. Furthermore, stochastic techniques can produce data models which correspond to human representational systems. The disadvantage is that a stochastic solution cannot be guaranteed to be optimal, unlike a purely mathematical result.

Stochastic searches modify the parameter values of an arbitrary data model in order to find the model which best fits the training data. Because **any** data model can be used, the model can be designed to be intelligible to humans. For example, an air conditioner could be trained to produce a comfortable environment, from examples of temperatures and humidities and the corresponding actions. If the air conditioner had a heater and a humidifier, and also temperature and humidity sensors, then the data model could consist of rules of the following form: `If (comparison of temperature or humidity), then (perform action).` The system would then deduce rules such as:

```
If temperature < 16°C, then turn on the heater.
If humidity < 30%, then turn on the humidifier.
```

However, if the data model had rules of form `If (A*temperature + B*humidity < 1), then (perform action)` then more complex rules could be produced, such as:

```
If 0.06*temperature + 0.005*humidity < 1, then turn on the
   heater.
```

Now, if the day were very humid, the heater would turn on only at a higher temperature than normal, to keep the building occupants as comfortable as possible.

Because a stochastic search can be made for the parameters of any data model, additional predefined restrictions can be built into a model to integrate human experience with training data. Commonly used stochastic search techniques include genetic algorithms (GA's), population based incremental learning (PBIL), and ant colony optimizations [12].

Stochastic techniques can provide good results, especially when an appropriate data model is used. Although classical statistical techniques can produce solutions more quickly for standard models, stochastic techniques offer far greater convenience and flexibility of representation than any other model.

## 2.6 Fuzzy Logic and Expert Systems

Fuzzy logic extends classification from a binary problem to a real-valued problem. Instead of classifying a datum as belonging to one particular state or another, fuzzy logic assigns it a degree of membership of each state [13]. (The sum of all of these degrees is 1, to indicate that these are the only possible states.) This differs subtly from probabilistic interpretations, in the following way: Imagine that a person is classified as short or tall, according to their height. If I declare that $p$(Fred is tall) = 80% and $p$(Fred is short) = 20%, then Fred is either tall or short, but I would bet 4 to 1 that he is tall. If, however, I state that Fred has 80% membership of the tall state (and 20% membership of the short state), then it is not possible to assign him definitively to one state or the other; he is a partial member of both.

Fuzzy logic essentially blurs the distinction between states, allowing powerful classification systems to be created, with fewer states than in traditional methods. Furthermore, fuzzy controllers interpolate between their control actions more smoothly than traditional state-based controllers [14]. Fuzzy logic implementations are also easier for humans to interpret than state-based classifiers, so their solutions can be examined and verified.

Expert systems are computer systems which attempt to mimic the assessments of a human expert. They have been demonstrated extensively in medical scenarios, in which a list of symptoms and test results is provided, and a diagnosis is called for. The facts are often combined into probabilistic chains of cause and effect [15], which are used to

produce a judgement of the most likely diagnosis. The chain of reasoning is usually provided by a human expert (assisted by a 'knowledge engineer'), who can also help to assign the probabilities. Once this has been done, training data can be used to improve the probability values.

Fuzzy logic and expert systems both provide more easily understood schemes of knowledge representation than traditional classifiers, through high-level semantic representations of their information. They are also amenable to stochastic search and other machine learning techniques, but the simplicity of their models must be maintained for this to be useful [16, p. 13,27].

## 2.7 Conclusion

The number of pattern recognition techniques commonly used proves that a different technique is most useful for each type of problem. The crux of choosing a technique is deciding how the data and the classifier should be represented, based on a trade-off between mathematical and linguistic models of the data. In this dissertation, concepts from Prolog will be used extensively in the semantic analysis of combinations of paths in the Smart Building of part IV.

# Chapter 3

# Probability

Pattern recognition is a complex task that requires computer decision-making. Humans are often required to make decisions based on inaccurate or incomplete information. Similarly, computers must often process information from unreliable sensors in order to make a decision, by sensibly combining probabilistic events. This chapter examines the mathematics of probability and uncertain reasoning, in order to provide a basis for computer analysis of complex data.

## 3.1 Absolute and Relative Probability

When a computer must rely on uncertain data, a probability is often assigned to the data as a measure of its accuracy. There are two different ways to measure this probability: absolutely and relatively. An absolute measure of probability describes the likelihood of a statement as a percentage, e.g. a statement of 80% probability would be true 4 times out of 5, on average. In a relative measure, only comparative probabilities are needed. For example, when a program must recognize faces from a training set, relative probabilities enable the best match to be chosen.

In order for absolute probabilities to be used, laborious physical tests must be performed to calibrate the probabilities in the system. Furthermore, environmental changes may alter these values, invalidating the measurements. Even so, a poor probability estimate can be better than none at all, and it is sometimes also possible to obtain a probability estimate by using two independent sensors to test each other.

Relative probability suffers from the opposite problem; since relative probabilities have

no direct physical basis, it is possible that an exceedingly unlikely solution to a problem will be selected as the 'best solution'. In absolute terms, this solution might be only 1% certain, yet it would be selected as best simply because it had a higher probability than the next best solution.

A further refinement of absolute probability is to specify the probability of an event using two numbers instead of one. The second value could be used to denote the confidence in the first value. For example, a probability pair $(80\%, 0\%)$ would denote that there was an exact 80% chance of a particular event happening, with 0% variability. Similarly, the pair $(80\%, 10\%)$ could indicate a 95% confidence that the true probability lay between 70% and 90%. When performing calculations with many probabilities of this type, it would then be possible to provide upper and lower bounds for the true probability of a combination of events.

## 3.2 Conditional Probabilities

In many real situations, the sensors which are used to detect events are unreliable; conditional probabilities can be used to codify the properties of these detectors.

A simple probability represents the likelihood or certainty of a particular event occurring, in terms of Bayesian probability [17]; the probability that event $A$ occurs is denoted by $p(A)$, which is a real number between 0 and 1. Conditional probabilities represent relationships between probabilistic events; the conditional probability $p(A|B)$ is the likelihood that event $A$ occurred if event $B$ is known to have occurred. By definition, $p(A|B) = \frac{p(A \wedge B)}{p(B)}$

As a physical example, imagine that a motion detecting camera is in a room, and it generates an event '$B$' whenever it detects movement. Assume furthermore that it generates '$\neg B$' if it does not detect movement, and it scans once per second. Let 'A' be the event that there is actually someone in the room.

If the camera detects movement one fifth of the time (one second in every five), then we could write $p(B) = 0.2$ and similarly $p(\neg B) = 0.8$ — the probability of no movement being detected. If the camera correctly detects movements 9 times out of 10, then $p(B|A) = 0.9$. Similarly, if the camera incorrectly registers a spurious movement once every 100 seconds, then $p(B|\neg A) = 0.01$.

From $p(B)$, $p(B|A)$ and $p(B|\neg A)$, we can compute all of the remaining conditional probabilities, such as $p(A|B)$, which represents the likelihood that there is actually

someone in the room when movement is detected, as follows:

$p(\neg B|A) = 0.1$ - the chance of failing to detect a movement[1]
$p(\neg B|\neg A) = 0.99$ - the chance of correctly identifying an absence of movement

Now $p(B|\neg A) = \frac{p(\neg A \wedge B)}{p(\neg A)} = \frac{p(B)-p(A \wedge B)}{1-p(A)} = \frac{p(B)-p(B|A)p(A)}{1-p(A)}$,
therefore $\frac{0.2-0.9p(A)}{1-p(A)} = 0.01$ and so $p(A) = \frac{19}{89} \approx 0.21$

From this, we can deduce that $p(A|B) = \frac{19 \times 9}{89 \times 2} \approx 0.96$ and $p(A|\neg B) = \frac{19}{89 \times 8} \approx 0.03$. These are the two most useful probabilities in determining the reliability of the motion detector readings.

## 3.3 The Mathematics of Events Sequences

Conditional probabilities may be combined to determine the probability of a sequence of events. For example, let us consider two motion detections in succession, and analyze the probabilities of their corresponding to 0, 1 or 2 real events. Label the detections $B_1$ and $B_2$, and the presences of a person $A_1$ and $A_2$.

The probability that both detections are correct is
$p(A_1 \wedge A_2|B_1 \wedge B_2) = \frac{p(A_1 \wedge A_2 \wedge B_1 \wedge B_2)}{p(B_1 \wedge B_2)} = \frac{p(A_1 \wedge B_1)}{p(B_1)} . \frac{p(A_2 \wedge B_2)}{p(B_2)} = p(A_1|B_1).p(A_2|B_2)$, if the events are independent.

Thus $p(A_1 \wedge A_2|B_1 \wedge B_2) \approx 92\%$,
$p(\neg A_1 \wedge A_2|B_1 \wedge B_2) = p(A_1 \wedge \neg A_2|B_1 \wedge B_2) = p(A_1|B_1).p(\neg A_2|B_2) \approx 3.8\%$,
$p(\neg A_1 \wedge \neg A_2|B_1 \wedge B_2) \approx 0.2\%$.

Therefore, the probability that there were two real events in 92%, the probability of one real event is 7.6% ($= 3.8\% \times 2$), and the probability that both detections were spurious is 0.2%.[2] From this, it is clear that the chances of both movement detections being spurious becomes vanishingly small, even with an unreliable detector.

We can use similar techniques to compute the probability that someone is moving in the room over $n$ time units, given that they are detected only $k$ times, namely $p$(All the $A_i$'s are true, given that $k$ of the $B_i$'s are false).

When a large number of events are combined, a combinational explosion occurs. As a result, the probability of any particular result becomes infinitesimal - in this case,

[1] Given that $A$ has occurred, then $B$ must either occur or not occur, so $p(B|A) + p(\neg B|A) = 1$.
[2] These probabilities should sum to 100%; the error results from rounding to 2 significant figures

there are so many different situations where one or more of the $A_i$'s is false, that they overwhelm the case where all of the $A_i$'s are true.

This does not actually reflect the true situation; given a sequence of motion detections $A_1, A_2, \neg A_3, A_4$ (2 detections, a gap, and another detection), a person would probably deduce that $\neg A_3$ was a false negative, and that there had been a person in the room for the whole period. This means that the events are not independent of each other. Nevertheless, the assumption of independence allows us to place bounds on the probabilities of event combinations.

## 3.4 Conclusion

All of the above concepts are useful in analysing the movements of people within a Smart Building, to produce probable paths of motion within each room. This requires comprehensive modelling of the uncertainties introduced by the cameras that detect motion, and by human activities.

# Chapter 4

# Distributed Computing

A group of computers can often perform a task far more quickly than one computer acting alone [18, p. 316].[1] This is the promise of distributed computing, in which a group of independent computers work together on a common task. These computers do not share memory, but they are able to communicate with each other via a network.

In many cases, the computers of a distributed system cooperate as peers, and no one computer has precedence over the others. In contrast, multitasking is a technique which allows a single computer to switch rapidly between tasks, acting as if it were performing them simultaneously — each task seems to have a computer to itself. The difference between distributed computing and multitasking is that multitasking involves one computer executing many different tasks at once, while in distributed computing, many computers share a single task. These techniques can also be combined, to allow many tasks to be distributed across a network of computers.

Programming a distributed system differs from programming a stand-alone system in two major ways.

Firstly, a stand-alone system is usually limited in speed primarily by the rate at which computations can be performed. In a distributed system, a significant limiting factor is often the rate at which data can be communicated between the separate processors. Therefore a processor must sometimes wait, idle, for more data to become available. (As a result, distributed systems share many of the challenges and techniques of 'event driven' systems.) An effective distributed system needs to keep as many processors working productively as possible at a time. Large data buffers would allow this, but

[1] For example, one portion of a program may execute well on one type of machine, with a fast processor, while another may run fastest on another computer with large amounts of memory

they introduce a large lag into the system, which would make it ineffective for real-time problems. Therefore other, more creative techniques must be used.

Secondly, pathological data dependency patterns must be avoided. The worst of these is a 'deadlock', in which two processes are each stuck, waiting for the other to produce more data. Automatic deadlock-avoidance algorithms do exist [19, p. 227], but they require explicit management of all resources in the system, which makes them difficult to use. An alternative to run-time deadlock prevention is to analyse the algorithms used with data-flow techniques, to ensure that no deadlock can occur.

In a real situation, a distributed algorithm should also be able to recover from physical hardware failures — if one computer crashes, the entire network of computers should be able to continue with most operations. The network should also continue to operate sensibly, even if some communication links break down. For this to happen effectively, a fully distributed computer system is needed.

## 4.1 Fully Distributed Systems

A fully distributed computer system is one in which there is no central, controlling computer; instead, all computers are peers and must decide on their behaviour collectively. This is the ideal case for many distributed systems: there is no central server which could crash and disable the whole network; network scalability is also not limited by the bandwidth of one server. However, this complete distribution of control has its own bandwidth cost, and there are certain classes of problems which cannot be solved in this way.[2] Furthermore, in a real situation, it is convenient to be able to log the results of the system centrally for archiving and accounting purposes.

Starting a fully distributed system initially, or restarting it after a catastrophic failure also presents difficulties; no central computer should be needed to coordinate the bootstrapping process. However, it would be useful if the system could nevertheless re-assimilate the information which it produced during its earlier operation. A solution is to use one particular computer to backup the important data of the system, and to act as a source of information after a crash.[3] This technique of centralising certain tasks in an otherwise fully distributed system, can also be used for tasks such as deadlock prevention (using the banker's algorithm [19, p. 587]). Although this can limit the degree

[2] An example of this is the Byzantine generals problem [19, p. 598]

[3] If this backup server is not active during the restart, its information will not be available to the system — but the system should nevertheless be able to act sensibly without it, and assimilate the information at a later stage

of fault-tolerance of the system, a major malfunction will in any event require the use of crash recovery techniques and roll-back (process termination), so this is not a great disadvantage.

## 4.2 Event Driven Systems

Event driven systems share many features with distributed systems — in fact, many distributed systems are based on the message passing paradigm of event driven systems.

Traditionally, computer programs are written from a procedural point of view: e.g. read a file from disk; sort the numbers contained in the file; write the new sorted file back to disk. In this model, the computer program directs all operations, and dictates the speed at which everything must happen. This is very efficient (since the processor is busy all the time) for systems where all the data is already available, and there is no hurry to produce output.

However, many computer systems work the other way around: it is external occurrences which drive the system. For example, a washing machine spends most of its time just waiting for time to pass, so that it can proceed to a new cycle, or for someone to press a button. Another example could be an air conditioner in an office, which captures temperature and humidity data one hundred times a second, filters it to reduce noise errors, and uses it to adjust the output power, as well as periodically reporting to a central building management computer.

If all of these occurrences are translated into events, then events direct the operation of the system, producing an 'event driven system' [20]. In an event driven operating system, events can be used for all communication between processes, as well as for communication between processes and hardware.

Thus, distributed and event driven systems operate according to a message paradigm: each item of information in the system is represented as a message, which is automatically routed from source to destination by the operating system. In contrast, classical operating systems, such as UNIX, are founded on a file-based paradigm: every aspect of the operating system is a file — including even the main memory and the CPU itself — and the operating system primarily provides file access mechanisms.

Just as the UNIX kernel is extended with file processing utilities, so distributed systems are extended with message handling utilities. For example, `sort` is a UNIX utility which reads the contents of a given file, and produces a new, sorted version. Similarly,

an Optical Character Recognition (OCR) process on a distributed system could consume scanned images (packaged into messages), and produce messages in which the characters have been recognised.

Distributed systems enable a group of computers to coordinate their activities, and thus solve far more complicated problems than a single computer could, if it acted alone. This is particularly useful for real-time applications, where speed is critical [21]. In fact, a centralised approach is often impossible for a real-time system because of the communication delays between the central computer and the remote sensors, and because of the enormous number of sensors in a large network; a distributed system is essential.

## 4.3 Object Oriented Programming

Object Oriented Programming (OOP) is a computer program design philosophy which sees complex computer systems as composed of many smaller components, called objects. An object essentially binds together related data and the actions which can be performed on that data. The actions are known as the object's interface, and objects interact by using each other's interfaces [22].

To illustrate this, consider the example of a first-in first-out queue. The concept is similar to a supermarket queue — there are two basic operations: adding data to the tail of the queue is like someone joining the shopping queue, and removing data from the head of the queue is like someone reaching the cashier. There are many different ways to program a classic data structure like this, but the functionality remains the same. All that one can do with a queue is add data to the tail and remove data from the head (and somehow check that there is actually data to remove).

Of course, one can have more than one queue, and although the queues will all have the same functionality, each will be storing different data. In computer science, these queue objects are called instances of the queue class, and their add and remove operations form the class interface.

In addition, one can have a slightly different class of queue, which exhibits the same basic behaviour as the original, but gives additional features as well. This is called a derived class, a child of the original class, and is roughly analogous to an express queue in a supermarket. Another example is the class of cars, derived from the class of vehicles — one would expect a car to exhibit all the same modes of behaviour as a more general

vehicle, and also to add extra functionality.

OOP is a philosophy of hiding the inner workings of objects from each other. If the implementation of an object is hidden, then other programs can no longer be dependant on its inner workings. This means that the inner workings can be modified without having to change the code of any other objects. In the supermarket, this is analogous to upgrading the cash registers, without having to inform every customer.

This data hiding is particularly useful in distributed systems, where OOP can shield ordinary objects from the inner workings of the system, but still give them access to the features which they need to communicate or migrate between computers.

## 4.4 Existing Distributed Systems

Distributed computing environments already exist for certain classes of problems; the most well known of these are CORBA, PVM and DCOM [23].

CORBA is the Common Object Request Broker Architecture, maintained by the Object Management Group (OMG). This is a standard interface architecture, which facilitates communication between objects in heterogeneous computer environments. CORBA is exceedingly useful for providing common interfaces for databases and other information services, but it is currently very restrictive when transmitting richly structured data between computers — only simple data can be sent with ease.

PVM (Parallel Virtual Machine) provides programmers with the illusion that a group of computers is actually a single machine. While CORBA emphasises the separateness of the computers and the unreliability of the network between them, PVM hides the network's very existence. This is most useful for problems requiring huge computational resources, such as ray-tracing algorithms, but not so useful for collating information, since this is a meaningless concept on the single virtual computer.

DCOM is Microsoft's Distributed Component Object Model. It allows objects on different computers to communicate with each other, and pass parameters to each other's methods, as if they were on the same computer. DCOM is primarily a communication mechanism, Microsoft's 'TCP/IP of objects' [24]; it is useful for linking applications which are already running on two computers, but it does not itself provide for remote creation of new objects on a computer.

The World Wide Web (WWW) is also a distributed computing environment, in a sense.

However, the greatest part of its communication is unidirectional, from web servers to computer users. In other words, it is ideal for client-server applications, but it does not allow for direct peer to peer communication.

## 4.5 Conclusion

Distributed computing systems allow independent computers to work together. However, for security and for commercial reasons, most architectures limit interactions to simple, passive data transmission. The next part describes the advantages of allowing more sophisticated interactions, where active processes can also move between computers.

# Chapter 5

# Conclusion

This part has shown current challenges in data processing. Firstly, methods of pattern recognition must be adapted for distributed environments where not all data is available. Secondly, great flexibility in probabilistic decision making is demanded by real-time processing, in order to generate and analyse many possible scenarios simultaneously. Finally, a distributed system must provide the correct level of abstraction for the objects that use it, to facilitate distributed pattern recognition.

The above survey of current methods of pattern analysis, probabilistic techniques, and distributed computing systems, suggests that there is a need for a new distributed architecture that allows efficient high-speed pattern analysis.

# Part II

# A Java Architecture for the Smart Building

# Chapter 6

# Introduction

Distributed objects require a computer environment which enables them to migrate between different computers. Part II of this thesis shows the development of a computer architecture, for distributed pattern analysis applications — such as a 'Smart Building'.

This part describes a new architecture which can be used to distribute the analysis of patterns to the source of the data. In this way, a network of smaller computers could be used to perform the task of a central computer at a far lower cost — since those computers would already be needed to acquire the data. The advantages of this system would include lower cost, greatly enhanced fault tolerance, and a reduction in lag between the production and processing of data, especially in very large networks — facilitating real-time control of the system.

This distributed architecture is Java based; the object-oriented data hierarchy allows data to migrate transparently and *actively* between computer systems of different types and processing abilities. This organisation allows both data and the agents which process the data to move, enabling both data and processing to be distributed across the system. In this way, those computers with related information could pool their resources to provide better results — while avoiding the information overload which would result if all systems shared all of their information.

Chapter 7 motivates why this architecture should be object-oriented, and why it should be written in the Java language. Next, chapter 8 explains a new technique which allows a group of objects to act together as a single object. Chapter 9 gives a precise specification of the requirements of the architecture, while chapter 10 describes in detail the implementation of a suitable environment. Finally, chapter 11 lists experiments which demonstrate the strength of the system developed.

# Chapter 7

# Motivation for a Distributed Object System

The problem with existing methods of real time pattern analysis is that they require high speed computers, with high bandwidth connections to their data sources. Traditional solutions which centralize the data before analysing it therefore require an immensely powerful, central computer, with very high bandwidth links to all of its sources of data, possibly extending over long distances. Distributed systems, on the other hand, can use low bandwidth network links to span long distances, with a computer connected to each data source.

In control and automation applications, a large network of sensors is often created. These sensors are then monitored in order to decide on the actions which the system must perform. Although there are many ways in which to make these decisions (such as centralizing all of the results, or performing the decision making locally at the computer nearest to each sensor), some strategies may be more efficient than others.

The ideal communication strategy depends on the lag and bandwidth of the network connecting the computers and the speeds of the computers, and also on the number of sensors and how quickly action must be taken.

This chapter first details why a distributed system is needed in order to coordinate this information efficiently and flexibly, and why objects are the best way to represent both the information and the agents which process it. Next is an explanation of the mechanisms which are needed for a distributed file system, which is an important first step towards a distributed object system. Finally, Java is shown to be an ideal language

for implementing these distributed systems.

## 7.1 Advantages of Distributed Systems

The simplest way to analyse distributed information is to send all of the information to a central server. This server can then hold all of the information, and make well-informed decisions. This centralised solution does have the advantage that all incoming information can be used for each decision.

A disadvantage of this solution is that it may be very inefficient, as discussed in chapter 4, the limiting factors being the processor speed of the computer and the bandwidth of its connection to the network.

An alternative solution is to have a computer connected to each sensor, to perform as much analysis as possible at each data source. Still, the decisions at one sensor will be dependent on results from other sensors. Thus a mechanism is needed for sensors to share relevant information, but which will limit the spread of unnecessary information, such as purely local information which has little or no effect on other computers' decisions. This is a distributed system, as there is no central point of control, and all of the computers are essentially peers.

Such a distributed system has the great advantage of fault tolerance, over a conventional system: there is no central server which could crash and disable the whole network, and if one computer crashes, the remaining computers should be able to continue with most operations. The system should also be able to operate sensibly, even if some communication links break down, and the network is temporarily divided into two disjoint subnetworks.

A distributed solution should also be considerably cheaper to implement than a centralised one: sensors are often connected to a network through small computers that preprocess their data. The underused processing power of these machines could be used to operate a distributed control system; now ultra-high speed network links and an enormously powerful central computer are no longer needed.

Distributed computing enables a network of lesser computers to cooperate to exceed the power of one huge central computer, by each working on a different part of a problem. The challenge is deciding how to subdivide a problem, and what information each computer should communicate to other computers.[1]

[1]Because of network communication overheads, suitable subdivisions should minimize intersite com-

## 7.2 Object-Oriented Distribution

For computers in a distributed network to be able to share their information, a standard system of representation is required. In simple systems, the possible data types and their characteristics are all known in advance, when the system is designed. However, more complex systems must be able to handle many heterogeneous classes of data, and they must also enable new data types to be added without redesigning the entire system [24].

For example, consider a robot designed as a distributed network of microcomputers — one for each joint, and one to process visual input from a pair of cameras. If the robot were to try to pick up a ball, the joint computers could receive information from the eyes detailing where the ball is relative to the robot's left arm. The left arm computer would then decide how the arm should move, and give instructions to perform the motion. Furthermore, it might communicate an outline of the movement to the right arm, to prevent the two arms from accidentally colliding with each other.

Traditional solutions for representing the data, as discussed in part I, can be overspecific, or excessively general, e.g. CORBA. If all data must be represented within a common framework, then the possibilities are limited for extending the system with radically new data types (since the entire system must be reprogrammed to accept new types). On the other hand, if a completely general and abstract representation system is used, then a huge overhead is incurred in communicating and interpreting both the data and its meaning together.

A solution to this problem is to represent each item of data as an object[2]. This has a number of advantages. One of the greatest of these is that code is directly associated with each data type. Therefore, if new data types are added to the system, the bulk of the existing program code is unchanged.

For example, in the robot analogy, if pressure sensors were added to the robot arm, then extra objects could be created to represent this information. If the old control program were used, then this new data would simply be ignored. However a new arm motor control program could take pressure data into account and stop moving the arm earlier than expected, if necessary. In other respects (such as communicating with the other arm), the new program could operate as before; in object-oriented terms, it could inherit its behaviour from its parent, the old program.

---

munication [25, p. 105]. In contrast, when multitasking on a single computer, communications overheads are seldom a significant cost.

[2] An object essentially binds together related data and the actions which can be performed on that data.

An object-oriented design also allows conceptually similar data items to share a common interface, so that on some level they could all be interrogated in the same manner. For example, both pressure and temperature sensors on a robot arm could provide objects with a `warning` method, in case the arm was being overtaxed.

### 7.2.1 Migration of Objects

Another major advantage of using objects for data is that the agents which process the data can themselves be objects. This is very important, because both information and agents can then migrate through the network. Thus the distribution of data and of processing can both be achieved in the same way.

For example, an object could represent the robot's intention to hold a ball in its hand [26]. This object could start in the vision processor, while searching and reaching for the ball, then migrate to the hand once the ball was grasped. If the ball were transferred to the other hand, the object could again move to follow it.

This example demonstrates an important design philosophy for distributed control systems: the internal state of the control system should mirror the actual state of the physical system, especially in spatial terms. Thus the 'ball' object would serve as an electronic avatar for the real ball; by allowing the object to follow the actual ball, the system would operate efficiently, and long communication paths could be avoided.

The question that arises is that of responsibility for the migration of objects. There are two answers to this question, corresponding to push and pull technologies on the Internet. In the case of an agent, it is clear that the object can anticipate where it should go, and it should be able to move itself automatically (through a bootstrapping hook into the new host computer). Passive data, on the other hand, would have to be pulled to another computer, if it was thought to be useful. Even so, an observer object would be needed to locate this data, and to inform the objects interested in it.

### 7.2.2 Duplication of Objects

It may be inefficient to have only one copy of a particular object in the system, particularly if objects on two different computers both need access to it. This problem may be resolved by duplicating the object, but care must be taken to ensure that the objects are not modified separately, which could result in inconsistent information propagating through the system. There are many different ways in which an object can be dupli-

cated; the most efficient way will depend on how the object is used. For example, certain types of information are 'read-only' and will never change, therefore it is safe to copy them freely without any restrictions except the assurance that at least one copy always remains. Section 7.3 gives more detail about various modes of data access.

For agents, even more advanced duplication modes are possible. In the simplest case, a single agent can move to follow useful data wherever possible. However, sometimes two sets of data may both seem useful simultaneously. In this case, the agent can either try to follow both sets at once, or it can split into two separate agents — one for each set of data. At a later stage, the agents may be able to recombine their separate findings, using hindsight to decipher the most probable set of external events. For example, if the robot's visual system were following the movement of a ball, the object following the movement of the ball might confuse the ball and its shadow and therefore split. Only at a later stage would it become clear, which of the two shapes was actually the ball, and then the objects could merge again.

Because there are so many different ways in which objects can migrate or duplicate themselves, it would be unrealistic to prescribe an exclusive list of modes to which objects must conform. Instead, objects should provide special methods to perform these actions themselves; the default behaviour would be for only the simplest of objects[3]. By allowing objects to copy themselves, the mechanisms for distribution remain simple and accessible, yet the system will be flexible enough to allow for advanced distribution strategies[4].

A distributed object system provides a powerful and general framework for programming distributed applications, yet it simplifies their design by enforcing the independence of different types of data. The distributed object system must provide only a few basic services, such as transmitting a given object to a remote computer and activating it, and locating objects already on a remote computer. This already provides enough power for simple distributed applications; more advanced programs could extend the basic mechanisms for extra flexibility.

[3]Standard classes could be designed for distributing data according to typical modes (e.g. read-only data); specific classes could then be derived from these, which would exhibit the same functionality.

[4]Existing object-oriented systems (such as CORBA) provide a much more restrictive architecture, in which objects cannot move, but must always be queried remotely. There are also standard distributed computing algorithms, such as PVM, which transparently treat a network of computers as a single computer — but these ignore the physical localisation of data in real control problems.

## 7.3 A Distributed File System

A distributed filing system illustrates many of the features of a distributed object system — a distributed filing system is essentially a distributed object system in which each object is a file. This section describes the design of a distributed filing system, as a first step towards creating a distributed object system.

Any filing system must provide a few basic operations [19]; a minimal set could be: obtain a list of files, read a named file, and write a named file. Underlying these three operations are two data structures: a data structure for naming files, and a structure for holding file contents. This is the only contact that processes need with the filing system; any other activities of the filing system should be transparent to the users.

Traditionally, files are stored on a disc which is physically attached to the computer which is accessing them. In this architecture, however, files are stored in the memories of the computers participating in a network, to produce the distributed filing system.

### Solution 1: A centralized file server

The simplest system to implement would scarcely be distributed at all. Rather, one computer would hold all files, and provide them to other computers on request.

Each computer would need to know the address of the file server in advance in order to access files, or the address could be broadcast periodically if the network supports broadcasting. For a file to be transferred, there would have to be both a file server program and a file receiving program. In loading a file, the client program would request a file (by name), and then wait for a reply message containing the file. Saving would be the reverse process.

This system would be very inefficient, especially if a file were often changed by the same process, or if the file server were a long distance away, or if only a small portion of the file needed to be changed. This situation could be improved upon if local copies of the most recently read files were retained. (However, before serving the local copy, the client program would first have to send a short message to the file server, to confirm that the original file had not changed since the copy was made.[5]) A more fundamental problem is that the entire file system would cease to function if the file server or its network connection failed.

[5] If certain classes of files were known to be definitively archived, and would never change, this confirmation step could sometimes be avoided.

### Solution 2: Passive distributed file servers

An obvious extension to a single file server system is a system in which every computer can function as a file server. This means that files can potentially reside nearer to the computers which access them, reducing network congestion, and no single computer would need to handle the burden of storing and transmitting all of the files.

The decision of where to store a particular file now becomes significant; the speed advantage to be gained from an efficient strategy is considerable. A simple solution would be to store the file on the computer which created it. This assumes that all computers have an equal need to create files, and an equal ability to store and transmit them. It also assumes that the creating computer will have the greatest need to use its own files. This is the simplest extension to the centralized file server model, and it has the great advantage that file naming is simple; if every file name includes the name of the computer on which it was created, then finding a file by name becomes a trivial task.

The major limitation of this scheme is that files are placed on computers at creation time, and then remain static within the network. Therefore, this system suffers from many of the disadvantages of the centralized approach when files are repeatedly accessed from a great distance. Furthermore, the network load might be very unequally distributed, particularly if certain computers primarily produced files, while others primarily consumed them.

For example, a few outlying computers might collect physical data for all the other computers to process. The links to these few computers might then be overwhelmed with all of the requests for the files. Again, caching recently used files could improve the performance. However, it would be better still if the files could migrate dynamically to where they were most needed.

### Solution 3: Active distributed file servers

When a file is accessed, the computer which receives it can make a copy for later use. Provided that the original file is not changed, the copy can always be used, saving network bandwidth. The essential trade off in a distributed filing system is between the bandwidth savings possible in keeping many copies of the same file, and the bandwidth costs resulting from the extra overheads of keeping the files synchronized. (Certain types of files are known never to change, making them particularly useful to cache.)

One method of reducing overheads is to permanently move a file from one computer to

another — the new computer can then serve the file immediately, without needing to request permission first. If one computer accounts for most of the access to a particular file, then transferring ownership could be profitable.

Another method of reducing network congestion, while still limiting the number of copies of a given file, would be to attempt to use the caches of nearby computers to access the file, before requesting it from the original server. This would be particularly useful if the computers were connected with a tree-like network, in which communications to local computers consume bandwidth only on local links, while communications with distant computers consume bandwidth on local and long-distance links.

Many files are read from far more often than they are written to. This fact can be used in the design of a distributed file system: previously, it was suggested that the owner of a file should be consulted before a cached copy is offered for reading or writing. This requires one message to the owner for each file read, and one message for each file write operation. An alternative solution is to place the onus on the owner, to warn the holders of duplicates whenever a file is about to be modified. In this case, no message is required when a file is read. However, when a file is about to be written, messages must be sent to purge each cached copy, and when a file is cached, the owner must be informed.[6] The bandwidth cost (and delay) of writing to a file becomes very large, especially if it is widely cached — but this is offset by the fact that each cached copy represents a saving of at least one read request for the file.

Clearly, both the client-controlled and the owner-controlled caching systems can be used simultaneously without interfering with each other. This suggests that the following caching strategy could be used: when a file is first read, the client retains a copy for caching, but does not inform the owner. When the file is accessed for a second time the client checks with the owner that the original has not been changed. If it is unchanged, then the cached copy is served, and the client informs the owner that it will henceforth assume its duplicate to be current, until it receives a purge message.

There is one major problem with this scheme: if just one of the registered caching computers (or its network connection) fails, then the cached file could never be written to again. Although such a failure is very unlikely, a safeguard must therefore be built into the system. The following timeout protocol restores stability; its only assumption is that all computers share a common time scale (not a common clock[7]):

[6] A reply is required for each of these messages, if the network is unreliable, especially if an outdated copy of a file could affect system consistency — though in certain cases, this is not a consideration.

[7] Systems share a common clock if they have a common absolute time reference. This is typical only for computers which are directly attached to each other. However, distant computers without a common clock can still have a common time scale; they can both pause for 60 seconds.

Within a pre-specified period of time, each client must transmit an '*I am alive*' message to its server. Then, the server responds with a '*You are alive*' message. Unless the client receives the response within the time limit, it must assume that its copy is no longer valid (because of a network or computer failure). Similarly, if the server receives no message, it may conclude that the client no longer believes itself to have a valid copy of the file. Therefore in the case of an error, the file will ultimately revert to a single-owner state[8].

This example illustrates the careful planning needed to ensure that a distributed system is stable, especially when its communication links are unreliable.

If the cached files were not purged, it might be possible to have two different versions of the same file on the network simultaneously. In certain situations, this is not a problem, for example if the files contain successive approximations to the solution of a mathematical problem — when an improved version becomes available, the older one can be replaced. A more difficult decision results when two changed versions exist of one file; an intelligent agent must decide whether to accept one or the other, or attempt to merge the two into a new solution.[9]

### Conclusions about distributed files

From the discussion above, it is clear that different types of files are used in fundamentally different ways. As a result, it would be simplistic to treat the filing system as a collection of homogeneous files. Instead, when a process creates a file, it should be able to specify how the file will be used. Table 7.1 lists the file types which were identified in the last section.

The ideal way to implement these different file types is by treating each file as an object, with all files descended from a common abstract ancestor class. Each subclass could then have its own specialized file handing functions. Furthermore, this object-oriented approach simplifies the extension of the distributed filing system to a distributed persistent object system.

[8] In the case of a distributed-object system, the original object would be the server, while each cached copy would be a client.

[9] This is an example of the relativistic difficulties of distributed networks; it is impossible in general for a group of computers on a network to agree on a common time scale. Some event pairs A and B can be ordered — for example, it might be possible to state that A definitely occurred before B. However, other pairs cannot possibly be ordered; there will always be events for which it will be impossible to determine whether A or B occurred first.

| |
|---|
| **Frequently changed files** May not be cached, unless the owner is consulted every time to confirm that the file is unchanged. |
| **Frequently used, occasionally changed files** May be cached, and the caches should register that they hold a copy of the file. (A timeout scheme is needed for consistency.) |
| **Read only files** Will never be modified, and can spread freely through the network. However, an owner is still needed to ensure that at least one copy of each file remains. |
| **Incremental files** Will not cause inconsistencies if outdated versions remain on the network, and are updated late. (This is very useful under heavy network congestion, or when a failure breaks the network into two subnetworks, and is cheap on bandwidth.) Changes are always made to the original file so that versions of the file form a linear sequence. (This would be achieved by migrating the file to the changing computer.) |
| **Mergable, incremental files** Can be updated anywhere, by anyone. Local consistency is achieved with a merging algorithm, which intelligently converts two changed versions of a common ancestor into a new solution. (Each file of this type would need its own merging algorithm.) |

Table 7.1: File types in a distributed system

## 7.4 Distribution with Java

Java is a computer language which allows the same computer program to run on a wide range of machines, from mainframes to terminals, and even embedded devices such as pagers and microwave ovens. For this to happen, each machine must run a 'Java Virtual Machine' (JVM) — although a different JVM must be produced for each type of computer, the same main program (compiled into 'Java Byte Code') can operate anywhere.

Furthermore, if a number of computers are linked together with a network, and they are all running JVM's, then they can use the network to share information. In conventional systems, data and objects must be translated laboriously into intermediate representations before being transferred to another computer. However, since all JVM's share a common representation for each type of object, Java objects can be transmitted directly from one computer to another. This is achieved using a mechanism known as 'object serialization'.

This makes Java an ideal language for producing a distributed object system, which can

operate across a heterogeneous network of computers.

Object serialization allows any object to be encapsulated into a stream of bytes, which can then be stored or transmitted across a network for later use [22]. It then allows the object to be reconstructed perfectly into its original form wherever it is received. If two computers agree to communicate across a network, then a program could be used to transmit an object from one computer to the other. However this does not take place transparently: a program is required on each computer to either package and transmit the object, or receive and reconstruct it.

Remote Method Invocation (RMI) is an extension to object serialization, which allows procedures and methods on a remote computer to be called directly from the local computer.[10] The parameters are then automatically transmitted across the network and reconstructed at the other end. RMI is essentially a pre-configured interface to object serialization, which makes it simple to create client/server and distributed applications.

A second service which RMI offers is a mechanism for identifying an RMI server by name, rather than by its physical address. This is another requirement for most distributed systems; there must be a mechanism which the computers on a network can use to identify each other on an ongoing basis. A common solution is to dedicate a process on one computer to identifying all the others, but this works only if the network is reliable. If the underlying network supports it, then server name broadcast is also effective, and there are even algorithms in which all computers will attempt to provide name facilities, which are especially useful when the network is occasionally split into disjoint subnetworks by physical interruptions.

Thus RMI provides the two essentials for a distributed system: mechanisms for throwing and catching events, and also a feature for broadcasting server names. Object serialization can then be used to migrate objects between computers, through these channels.

## 7.5 Conclusion

Centralised control is often impossible for many real-time control and analysis applications, since the volume of data produced and the timing constraints on the output would be overwhelming. The alternative is to distribute the processing of data nearer to the data source.

The chief difficulty in programming a distributed system is ensuring that information

[10] This is similar to Remote Procedure Calls (RPC) under UNIX

is channelled to the computers that most need it. Instead of creating prescriptive and restrictive high-level mechanisms for this, an architecture is proposed here in which the data can guide its own migration. In this architecture, objects are used for all data. This gives far greater flexibility, with little extra programming cost, since specific classes of data can be derived from abstract superclasses which represent their desired migration policy.

This architecture has the considerable added advantage that both the data and the agents that operate on the data can be treated as objects. As a result, both data and processing can be distributed in the same way, since agents can control their own migration and propagation through the network.

# Chapter 8

# Truly Distributed Objects

Truly distributed objects should not be restricted to being in only one place at a time. This chapter develops an approach which enables a family of ordinary migrating objects to act together as a single object. This technique is explicitly designed into the distributed object architecture of chapter 9, but it can be used in any distributed object system.

## 8.1 Making Distribution Explicit

Just as object-oriented design discipline can be used with non object-oriented languages (such as C), so too can truly distributed objects be built without explicit support from the programming language. Nevertheless, an object-oriented language (such as Java) helps to hide the details with simple syntax, while also providing built-in support for remote access and externalization of objects (through Remote Method Invocation and object serialization).

The challenge of distributed objects is deciding how explicit the distribution should be. As discussed in section 4.4, architectures such as PVM [27] hide distribution completely, treating a network of distributed computers as a single computer. In contrast, CORBA makes explicit which calls affect remote objects (but object migration facilities in CORBA are still very limited). These approaches are different because they make different assumptions about the network connecting computers: PVM implicitly assumes that the network is usually reliable, and that computers are relatively close to each other with high bandwidth, low lag connections, while CORBA assumes that the network is unreliable and unstable.

The speed of the network affects the granularity[1] at which parallelism and distribution is useful; a fast and reliable network (relative to CPU speed) is needed to make fine grained parallelism worthwhile [25]. When programmers write software for a particular distributed architecture, they choose a design which reflects the expected network effectiveness. Therefore, the programming architecture should allow the programmer to explicitly distinguish between local and remote objects.

It would be ideal to be able to treat local and remote objects in exactly the same way. However, an imperfect network makes this an unattainable goal. Remote objects can fail in ways which are impossible for local objects. Furthermore, it is impossible to evaluate whether a local or remote call would be more efficient, without knowing the time overhead associated with the remote call, in advance.

## 8.2 Comparing Mobile Objects and Distributed Objects

The phrase 'Distributed Object' is often used to refer to an object in a computer system which can operate somewhere other than on the server computer. The implication here is that a distributed object is an object which can operate anywhere. This is extremely useful for dividing business applications into a three-tier hierarchy of clients, middle-ware business logic, and servers [23] — the same code can be reused in all of these locations. In the same sense, Java applets[2] can also be referred to as distributed objects, when they are in fact simply objects that can begin their execution anywhere. In Object Management Group (OMG) terminology, they are 'stationary agents' [28, p. 7].

More complicated than these are objects which begin execution on one computer, and can then move to another computer. An ability to do this requires that extra features be provided by the operating system or system architecture, to enable remote injection of objects into a computer. According to the OMG, these 'mobile agent' objects require capabilities which many current distributed object systems do not offer.

Both systems described above assume that an object can be in only one place at any one time. However, just as it is useful for a multi-threaded program to operate on more than one CPU at a time, it is also useful for a distributed object to use more than one computer at once. A system which enables such true distribution is proposed below.

[1] Granularity is the optimal size of code fragments to be shared between computers.
[2] Tiny programs inserted into web pages on the internet

## 8.3 Truly Distributed Objects

A family of distributed objects can cooperate, to act together as a single object.

An important assumption of object oriented programming is that objects are completely specified by their interfaces. In other words, the internal state of an object is completely hidden from other objects, except where it is revealed by the methods (or public data members) of the object. This means that the code for an object could be rewritten completely, and the other objects would remain entirely unaware of this, provided the behaviour of the interfaces did not change.

This assumption is used by the Java RMI classes — when access to a remote object is needed, a local proxy [29, p. 207] object is created which passes the method call parameters across the network. The original object then performs the required computation, and the result is returned via the proxy object. Here, both the original object and the proxy behave in the same way, and from the perspective of the class interfaces, they might as well be the same object. In a sense, they are both simply interfaces onto the same conceptual object.

A 'truly distributed object' is the name used here for such a conceptual object. It is distributed in that it is spread across many computers. It may have many smaller constituent objects, but they all provide the same interface behaviour, and are thus simply representatives of the larger whole. A second requirement for a truly distributed object is that it must be accessible from anywhere — or at least from any computer which is part of the distributed object system.

Java RMI fulfils both of these requirements. However, it allows only one pattern for implementing truly distributed objects. In the case of RMI, each truly distributed object is functionally implemented by exactly one of the physical objects in the system; all of the rest of the objects are merely proxies. However, this need not necessarily be true.

The information need not all be held by just one object. Instead, a group of objects can all behave as interfaces to the truly distributed object, and yet have their information distributed between them. A simple example would be a truly distributed object that implemented a distributed file system. Files could be stored where they were created, and yet be accessible from anywhere. In this example, no single object contains the entire state of the system, unlike RMI. Other truly distributed objects include distributed computation engines, and distributed databases.

In fact, any computer program which operates across many computers can be seen as an example of a truly distributed object. What is interesting is that truly distributed objects make no restrictions upon the internal structure of the system — all client-server architectures are just a small subset of the set of possible truly distributed object systems. Whenever a group of objects, on different computers, work together to perform a single task, the result is a truly distributed object.

Truly distributed objects have certain extra requirements over ordinary objects. Given a reference to a truly distributed object:

1. There must be a mechanism for obtaining access to the local representative of that particular truly distributed object.
2. If there is no local representative, then access to a remote representative must be possible.
3. For this, there must also be a mechanism for communicating with both local and remote representatives of a truly distributed object.

The first two points above require a scheme for uniquely identifying a truly distributed object; this object naming is described in the following section. The third requirement, for communication, must be a feature of the distributed object system being used, and it is a feature of the architectural specification developed in chapter 9.

## 8.4 Naming of Truly Distributed Objects

For distributed objects to be able to cooperate, a mechanism is needed whereby objects can locate each other, for example by name. One mechanism frequently used for this is for each object to have associated with it a name and a location. This works well for objects which remain in the place where they were created. A refinement is to associate each object with its original location, and a unique name within that location. When the object moves to a new computer, the original location keeps a reference to the object's new location, so that the object can always be located by name. There are also mechanisms for anonymous communication between objects, using 'badges' [30, p. 9]

Most current systems, including CORBA and DCOM, presuppose that an object is in one place at any moment [30, p. 3] — if an object were to split into two objects, in order to perform a task, each of these objects would have a different name.

For distributed processing it is useful to be able to subdivide a single task across many computer systems, while still retaining one name for all of the subsections of the task. For example, if two objects in two different rooms (on two different computers) were both attempting to follow the movements of the same person, then those objects should have the same name, and should be addressed in the same way. It is then the responsibility of objects sharing a name, to coordinate their activities in order to present a consistent front to other objects.

## 8.5 Conclusion

Truly distributed objects enable a group of objects to act together as a single object, under a single name. This tool allows data hiding in distributed systems, just as ordinary objects enable data hiding in conventional object-oriented systems. In this way, a distributed system can be designed more easily than before, as a group of interacting truly distributed objects.

# Chapter 9

# Architectural Specification

Distributed objects require a computer environment which enables them to migrate between different computers. This chapter gives a precise specification of a computer architecture for distributed objects.

The architecture is specified entirely in terms of the services which it provides to distributed objects, which makes it independent of the programming language in which it is implemented. The specification also makes clear the separation of responsibility, between the distributed object system and the distributed objects within it. The wider implications of this specification are expanded upon in section 9.2.

## 9.1 Specification

The assumptions below specify an environment for distributed objects to inhabit. The basic assumptions are

1. All objects in the universe are Java objects.
2. There are two categories of object in the universe: ordinary objects and 'RemoteObjects'. RemoteObjects are actually ordinary objects with certain extra properties, described below in assumptions 4, 5, 6 and 7.[1]
3. The universe in which objects reside is divided into a number of 'Places' [28, p. 10]. If two objects are in the same Place, then they can communicate with each other

[1] The term 'object' often denotes both ordinary objects and RemoteObjects, when its use is unambiguous.

directly. Otherwise, they can communicate only indirectly. (Thus each Place has its own separate Java Virtual Machine).

4. There is a mechanism which enables any object to send a message to a particular RemoteObject in a given Place, and receive a reply [31].

5. When a RemoteObject is first created in a particular Place, it is assigned a unique 'Name' — unique across all Places.

6. If a RemoteObject is located in one Place, a 'Copy' of the object can be made to any other Place. This new RemoteObject has the same Name as the original, even though it is located in a different Place. The original object is responsible for actually producing the Copy, although any object may request that a copy be made.

7. A Copy of a RemoteObject is not necessarily identical to the original. Rather, its aim is to provide the same functionality as the original — the Copy is a proxy that provides access to the object's data. Thus the conglomeration of objects with the same Name form a 'SuperObject'.[2] These objects are discussed in more detail in chapter 8, as 'truly distributed objects'.

8. Each Place maintains a list of RemoteObjects located there, and this list is accessible to all objects in the universe. (When a Copy of a RemoteObject is made in a new Place, the Copy is added to the new Place's list.)

9. There is a 'Directory', which lists all available Places. This Directory is also accessible to all objects in the universe.

A conceptual class diagram for the relationships between these objects is given in figure 9.1.

## 9.2 Implications of the Specification

The assumptions above describe a universe within which many objects can co-exist and interact. Among these can be simple objects, which passively carry pieces of information from computer to computer. There can also be more advanced objects, which combine these elementary items of information, to draw more advanced conclusions.

[2] A SuperObject is essentially a distributed equivalent of an ordinary object. The difference between a SuperObject and a conventional distributed agent [28, p. 7] is that a SuperObject can be in many Places at once, while an agent may move between Places, but is limited to one location at a time. In this context, a SuperObject can offer the same services as a family of agents.

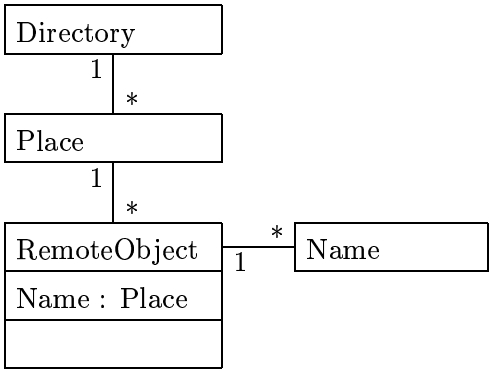



Figure 9.1: Conceptual class diagram

Objects can effectively move from one computer to another, since the original can cause a Copy of itself to be made at a new location. In addition, the Directory allows objects to discover which computers are available. As a result, an object can move itself to any computer in the universe. Thus, *the universe is essentially one large computer*, from the perspective of an object within it, since any object can access any other object from anywhere.

Places give RemoteObjects an explicit sense of distance. All objects which are located in the same Place are nearby and easily (speedily, cheaply) accessible; all objects not in the same Place are more expensive to access.

Places also allow a certain degree of security for objects, since RemoteObjects are responsible for making Copies of themselves. For example, an object containing sensitive information could refuse to allow a Copy of itself to be made, or it could send a dummy as a Copy, and allow sensitive information to be released only to pre-specified Places.

The universe allows objects to practise advanced migration strategies. The simplest technique is to allow no migration at all. Another technique is for an object to simply duplicate its information when a Copy is required. In that case, the original and the Copy will be entirely independent; if one is changed, the other will remain unaltered. A more advanced implementation might allow a single item of data to be shared between a number of Places, with the actual data item being moved to wherever it was needed, on demand. In that case, a group of RemoteObjects (all with the same name, but located in different Places) could provide distributed access to a single shared object.

The mechanisms required for these migration techniques are discussed in detail in section 11.5, where class `ProtoMigrator` is developed to demonstrate this behaviour.

The fact that RemoteObjects are responsible for Copying themselves provides great

flexibility in the interpretation and implementation of RemoteObjects. For example, RemoteObjects need not represent only data; they could have executing processes associated with them too. A distributed computation engine could be established so that, wherever the RemoteObjects were registered, a new thread of execution would be created. These threads (one per Place) could cooperate to solve the problem at hand. An example of a program which uses this strategy is presented in section 11.9.

## 9.3 Conclusion: A Metaphor for the Architecture

This section presents a metaphor for describing the architecture of the distributed object system as a number of cities.

Each Place in a distributed object system can be seen as a city. Within that city, there are many people, representing RemoteObjects. Each person also has an address at which they can be contacted, like Names in the distributed object world. People can communicate with each other, either within a city, or across cities, but communication with other people in the same city is considerably quicker and easier than communicating with another city. This is because a letter to another city could be delayed or lost in the mail. There are also similar risks for a person travelling between cities.

In the distributed world, a SuperObject is a group of RemoteObjects that do the same job in different places, and share a common Name. Similarly, in the metaphor, a group of people who do the same job can be considered to belong to the same company. If a person needs to communicate with the company, he or she approaches the local representative first, and contacts the company's head office only if essential.

Finally, each city has a directory, listing all of the People there, and the central post office has a directory of postal codes, listing every city. This is analogous to the listing service and Directory in the world of RemoteObjects.

This metaphor helps to illustrate the crucial differences between local objects and RemoteObjects — there can be delays and risks in moving between Places, which do not occur for objects which remain in the same Place. It also shows why a family of objects can work together more efficiently than one peripatetic, roaming object. A roaming object cannot be in two places at once, and therefore it incurs large costs when it must communicate with other objects in different Places simultaneously [32].

# Chapter 10

# Implementation

This chapter describes an architecture which was developed to satisfy the specifications of chapter 9.

The architecture allows distributed objects to share a network of computers for information processing, and it is upon this architecture that a 'Smart Building' will be demonstrated.

## 10.1 Correspondence between Classes and Specification

The Java classes contained in package `db.smartroom.server` implement a distributed object architecture. This section details how the classes used in that architecture correspond to the architectural specification of section 9.1. Each point in that specification has a correspondingly numbered point here, describing how it is fulfilled.

1. All objects in the system are Java objects. `Typewriter` text is used to denote Java class names, and also for Java source code extracts.
2. 'RemoteObjects' are those objects which implement the `RemoteObject` interface.
3. Each 'Place' corresponds to a `FileServer` — usually an instance of `FileServerImpl`.
4. Each `RemoteObject` has a `communicate` method, so that messages can be sent to it. `FileServers` also have `communicate` methods, to enable communication with RemoteObjects from other Java Virtual Machines. The return value of a call to `communicate` enables the RemoteObject to reply to each message.

5. 'Names' correspond to `RemoteReferences`; a `RemoteReference` to an object can be obtained with `RemoteObject.getHome()`; it is assigned using `RemoteObject.register()`. Two RemoteObjects `a` and `b` have the same Name if their `RemoteReferences` are equal — i.e. if `a.getHome().equals(b.getHome())` For convenience, the `equals` method of class `RemoteObject` is overridden, so that the expression above may be simplified to `a.equals(b)`.

6. RemoteObjects can be Copied by calling `FileServer.open(ref)` on a remote FileServer. Java's serialization and deserialization process causes the Copy to be made.

7. The local copy of a RemoteObject is accessed (from `RemoteReference ref`) with a call to `LocalFileServer.get().open(ref)`; if the object is found locally, a local reference to it is returned. If not, `null` is returned, and the object can be accessed using `ref.open()`

8. To obtain the list of objects on a FileServer, the `FileServer.listFiles()` method is used.

9. The 'Directory' is provided by implementations of `FileServerLister`, such as `FileServerListerImpl`.

From the list above, it can be seen that all of the requirements of the architectural specification are fulfilled by this implementation. The list also makes clear how Java classes correspond to the elements of the conceptual model.

## 10.2 Class Overview

The following Java classes and interfaces form the core of the architecture which was developed. They are all part of package `db.smartroom.server`.

`FileServer` A `FileServer` is created on each Java Virtual Machine; all of the `FileServers` together produce the space in which distributed objects can cooperate. This interface is RMI-enabled; in other words, remote method calls can be made to those classes which implement this interface.

`FileServerLister` `FileServerListers` enable objects to find `FileServers` (other than the local one). This interface is also RMI-enabled, allowing remote access from other JVMs.

`RemoteObject` Each distributed object (or 'RemoteObject') must implement the methods of interface `RemoteObject`. It is this interface which describes how objects interact with `FileServers` when they are created or duplicated.

`RemoteReference` Each `RemoteObject` is assigned a `RemoteReference` when it is first created on a `FileServer`. This `RemoteReference` uniquely identifies the object throughout the computer system.

`LocalFileServer` This class facilitates access to a `FileServer` and a `FileServerLister` on each Java Virtual Machine. If a `FileServer` is called for, and none exists yet, a new `FileServerImpl` implementation is created; similarly, if a new `FileServerListerImpl` will be provided if needed.

`FileServerImpl` This is a simple but thread-safe implementation of the `FileServer` interface. It maintains a list of all `RemoteObjects` stored on a particular virtual machine. (This class is RMI enabled.)

`FileServerListerImpl` This class implements `FileServerLister`, and is RMI enabled. This implementation assumes that the is only one `FileServerLister` in the system.

Figure 10.1 is a class diagram, which shows how these classes and interfaces interrelate.

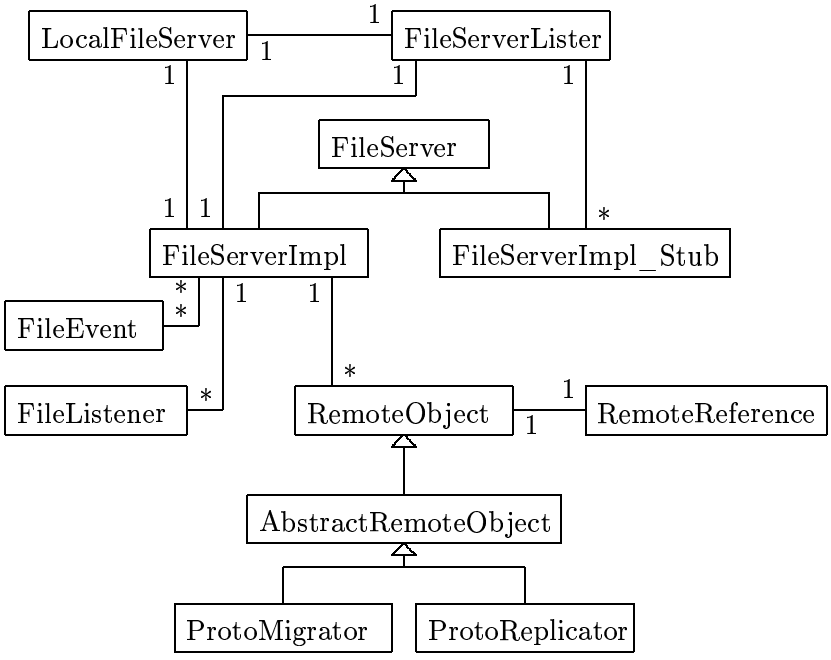



Figure 10.1: Class diagram of package `db.smartroom.server`

## 10.3 Details of the Implementation

### Class `FileServer`

FileServers form the structure of the space in which distributed objects reside. Each RemoteObject is associated with a specific FileServer, which acts as its home base. Although copies of the object may be made to other FileServers, a copy of the object always remains at the home base. As a result, a RemoteObject can always be accessed, just by knowing its RemoteReference. In other words, a RemoteReference not only uniquely identifies a RemoteObject; it also makes it possible to locate that object.

FileServers allow objects to be created, accessed, listed and deleted, both remotely and locally. They act as storage points for RemoteObjects, as name servers for giving RemoteObjects unique identifiers, and as interface points to allow other tasks to access the objects. It is also possible to add `FileListeners` to a FileServer; these listeners are notified whenever a change is made to the objects on the FileServer. Notifiable changes include creating a new file, deleting a file, and attempting to recreate a duplicate file on a FileServer. Each change is sent as a `FileEvent` object, which identifies the `RemoteObject` affected, and also the type of change which occurred.

### Class `RemoteObject`

`RemoteObjects` are the primary inhabitants of the universe created by the FileServers. Each RemoteObject is identified by the `RemoteReference` returned by its `getHome()` method.

The life-cycle of a typical RemoteObject is as follows:

1. Before it is first registered with a FileServer, a RemoteObject `obj` behaves just as any other Java object; because it implements the `Serializable` interface, it can be converted into a stream of bytes and transmitted from one computer to another, or temporarily stored on disk.

2. At some point, `obj` will be associated with a certain FileServer `filesrv`, as the result of a call such as `filesrv.create(obj)`.

3. `obj`'s `register` method will be called, with a unique `RemoteReference ref` which identifies `obj`.

4. The object which results from the call to `register` is stored in a list on `filesrv`. At this point, `filesrv.create` returns the `RemoteReference` to its caller.

5. When `obj` needs to be accessed, a call is made to `filesrv.open(ref)`. If the call is made from the same Java Virtual Machine on which `obj` is stored, then a local reference to `obj` is returned. Otherwise if a remote request has been made, `obj` is serialized, transmitted across the network, and deserialized on the caller's JVM. The deserialized copy of `obj` will then typically register itself with the new FileServer located there. (This re-registration will be made with the same home location as `obj`, and with `filesrv` as the home FileServer.)

   If the RemoteObject is not listed on the FileServer which is being called, `null` is returned. This enables the caller to decide whether to retrieve the RemoteObject from its home location, and to check whether a local copy of the object exists.

More specific information about using the classes of package `db.smartroom.server` is contained in the javadoc documentation comments embedded in the class files; appendix A includes a summary of this information. The source code for the example programs of chapter 11 demonstrates how the classes are typically used.

## 10.4 Conclusion

The classes developed in this chapter fulfil all of the specifications of chapter 9, providing a suitable environment for distributed objects to inhabit. The examples of the following chapter also help to prove that this implementation performs according to its specification, and that the specification is indeed useful in typical distributed object scenarios.

# Chapter 11

# Testing and Verification

The distributed object system implemented in chapter 10 is shown to be effective by the example programs of this chapter.

A set of programs was developed to test and exercise the system, in order to demonstrate that the architecture developed in this part of the dissertation performs according to its specification. They also serve as prototypes for typical applications of distributed objects, while proving that common problems can indeed be solved effectively using distributed objects. This chapter describes the purpose and results of each of these programs.

The list below gives an overview of the experiments described in this chapter and their objectives, by section number.

**11.1** RemoteObjects can be created and accessed using the local FileServer.

**11.2** FileServerListers enable FileServers on other Java Virtual Machines or other computers to be located.

**11.3** Objects can be created on a FileServer remotely.

**11.4** Data can be replicated to wherever it is needed, automatically.

**11.5** A single item of data can be shared across computers.

**11.6** This sharing is robust, even when different sources try to access the data simultaneously.

**11.7** RemoteObjects may be graphical objects.

**11.8** RemoteObjects may act as proxies for other RemoteObjects (and thus may also act as graphical interfaces to those objects).

**11.9** RemoteObjects can be used to implement distributed computation engines.

Full instructions for using these demonstrations are included in the `README.TXT` file, on the CD-ROM which is included with this dissertation. Abbreviated instructions follow.

To run any of these demonstrations, first ensure that Java 2 (the Java Development Kit, version 1.2) is installed on a computer. The demonstrations will operate on any computer platform that supports Java 2, but convenient batch files are written for Microsoft Windows. Next, include the `/Classes` directory of the CD-ROM in the `classpath` environmental variable. Then either type in the commands included with an example, or change to the appropriate directory to run the batch files. For example, to run the `MakeServer.bat` batch file of example 11.2, change to the `/Source/db/smartroom/server/debug` directory of the CD-ROM, and run `MakeServer.bat`

## 11.1 Demonstration of Local Object Creation and Listing

### Aim

To show how objects can be created on a FileServer, and then how a list of the resulting objects can be obtained locally (on the same Java Virtual Machine).

### Method

The main method of Java class `db.smartroom.server.debug.LocalCreate` accesses the local FileServer, and creates three instances of class LocalCreate there. Each of these objects displays the list of available objects, waits 10 seconds, then retrieves and displays the list again.

To run the program, execute the following command on any computer:

```
oldjava db.smartroom.server.debug.LocalCreate
```

### Results

As the objects are created, they show a progressively growing list of RemoteReferences. These objects also demonstrate that the methods of the FileServer class are thread-safe, since they attempt to access the file list from another thread during the object registration process.

From the output of this program, it can be seen that the FileServer's object list is locked while a new object is being registered there. (In other words, no other process can access the list, while one object is being registered.) The reason for this is to ensure that the newly registered object is guaranteed to be on the list for any new threads created during the registration process. A corollary of this is that register methods should not perform time-consuming operations before returning (since these would then block all access to the file list), but should instead delegate them to a separate thread.

## 11.2 Listing of File Servers

### Aim

To demonstrate the `FileServerLister` class, and how it enables objects to locate File-Servers on different Java Virtual Machines.

### Method

The main method of class `db.smartroom.server.debug.TestLister` lists all accessible FileServers listed with the local FileServerLister. (The preferred location for referencing the FileServerLister may be given as a command line parameter.) Class `db.smartroom.server.debug.MakeServer` has a main method which creates a File-Server with no files on it. This is used to set up a FileServer on another computer, without initially creating any objects on it.

To run the program, first make the `db.smartroom` classes available on a web server, at some location 'URL'. Next create a file '`rmipolicy`' containing the following:

```
grant {
permission java.net.SocketPermission "*", "accept,
connect, listen, resolve";
```

```
};
```

Execute '`rmiregistry`' on a particular computer XYZ, where XYZ is the computer's DNS name or IP address. Then, on any number of computers, execute

```
java -Djava.security.policy=rmipolicy -Djava.rmi.server
.codebase=URL db.smartroom.server.debug.MakeServer XYZ
```

Finally run the following command on the first computer:

```
java -Djava.security.policy=rmipolicy -Djava.rmi.server
.codebase=URL db.smartroom.server.debug.TestLister XYZ
```

The batch files '`MakeServer.bat XYZ`' and '`TestLister.bat XYZ`' in the `db/smartroom/server/debug` directory execute these commands, and a copy of '`rmipolicy`' is also in that directory.

### Results

**Remote Method Invocation** These programs provide a working demonstration that Java Remote Method Invocation (RMI) can practically connect objects on different JVM's. In this case, a number of FileServers (on different JVM's, and possibly on different computers) all connect to a single FileServerLister, and store their details there, using RMI. Then, again using RMI, the `TestLister` program connects to the FileServerLister to retrieve the list of FileServers.

This remote calling of the methods of an object is crucial to the distributed architecture; it is this that enables RemoteObjects to be accessed remotely through FileServers.

**Java Security Model** The new Java 2 security model is also used in this example. By default, all applets and applications are restricted in which system resources they can access [33]. For this example, a security policy file is created, in order to extend the rights of the demonstration programs; in this case, extra permission is granted to allow access to any network location. '`modifyThreadGroup`' permission is also given, to allow threads to sleep.

## 11.3 Remote Object Creation

### Aim

To show how an object can be created on a FileServer remotely (i.e. from a different Java Virtual Machine).

### Method

Class `db.smartroom.server.debug.RemoteCreate`'s main method creates a simple RemoteObject on every FileServer it can locate.

To run this example, run '`rmiregistry`' on computer XYZ, as in experiment 11.2. Then execute '`MakeServer.bat XYZ`' on any number of computers, and finally '`RemoteCreate.bat XYZ`' on one computer.

### Results

On each computer running a FileServer, `RemoteCreate` creates a `db.smartroom.server.debug.LocalCreate` object, with a title such as 'LC#1' or 'LC#2'. Since `LocalCreate` does not override the default serialization mechanism, and it follows the conventions of `db.smartroom.server.AbstractRemoteObject`, the object is first constructed on one JVM, and is then duplicated exactly to its new location.

`LocalCreate` has no serializable fields except for the object title, so that is the only additional information that is included when it is created remotely. When this example is run, it can be seen that this information is indeed transmitted, and stored on each FileServer. This is shown in the sample output below.

On the computer where `RemoteCreate.bat` was run, the sample output was:

```
Obtaining list of FileServers
Created LocalCreate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref:[endpoint:[137.158.135.197:2276](remote),
objID:[5876c40b:d8fc431958:-8000, 0]]]]
Created LocalCreate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref:[endpoint:[137.158.135.197:2284](remote),
```

```
objID:[5ad4c407:d8fc43418f:-8000, 0]]]]
Created LocalCreate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref:[endpoint:[137.158.135.197:2288](remote),
objID:[5ad9c407:d8fc4343f2:-8000, 0]]]]
Total FileServers found: 3
```

On the first FileServer, the output was

```
Registering LC#1@null
List of files while registering: []
New Thread started
Have created new file: LC#1@RemoteReference(0:1)
Files before delay: [RemoteReference(0:1)]
Files after delay: [RemoteReference(0:1)]
```

On the second FileServer, the output was

```
Registering LC#2@null
List of files while registering: []
New Thread started
Have created new file: LC#2@RemoteReference(0:1)
Files before delay: [RemoteReference(0:1)]
Files after delay: [RemoteReference(0:1)]
```

On the third FileServer, the output was

```
Registering LC#3@null
List of files while registering: []
New Thread started
Have created new file: LC#3@RemoteReference(0:1)
Files before delay: [RemoteReference(0:1)]
Files after delay: [RemoteReference(0:1)]
```

## 11.4 Replicating Objects

### Aim

To prove that an unchanging piece of information can be shared between FileServers, through a collection of RemoteObjects. (Alternatively, the information may be allowed to change, but the different versions of it will remain independent — so that changes to one copy of the information affect none of the other copies.)

### Method

The item of data which must migrate is attached to a RemoteObject of class `db.smartroom.server.ProtoReplicator`. Whenever the `ProtoMigrator` is copied to another FileServer, the data item moves with it. The migrating must be serializable, but there are no other restrictions on it. This allows the replication behaviour to be separated entirely from the object which is being replicated. ProtoReplicator is implemented as a simple extension of AbstractRemoteObject, which describes typical behaviour for a RemoteObject. It adds a non-transient data field to hold the data item, and a method for accessing it.

Class `db.smartroom.server.debug.Replicator` extends ProtoReplicator by adding methods to duplicate the Replicator to all available FileServers, and display a message wherever it is created. The copies are shown to be independent by changing the shared text message with each iteration.

This example can be run in the same way as experiment 11.3, using `Replicate.bat` instead of `RemoteCreate.bat`.

### Results

From the sample output shown below, it is clear that the text messages associated with the Replicator objects are independent of each other.

On the computer where `Replicate.bat` was run, sample output was:

```
Created Replicate on
db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:1076](remote),
```

```
objID:[5ad517ee:d9016bf440:-8000, 0]]]]
```

On the first FileServer, the output was

```
Have registered MyReplicator
Duplicating object
Delay for 5 seconds
Created duplicate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:1076](remote),
objID:[0]]]]
Created duplicate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:1083](remote),
objID:[5ad517fb:d9016c1f47:-8000, 0]]]]
Created duplicate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:1087](remote),
objID:[5ad517fb:d9016c21be:-8000, 0]]]]
Object name is now MyReplicator, Original
```

On the second FileServer, the output was

```
Have registered MyReplicator, Copy 2
Delay for 5 seconds
Object name is now MyReplicator, Copy 2
```

On the third FileServer, the output was

```
Have registered MyReplicator, Copy 3
Delay for 5 seconds
Object name is now MyReplicator, Copy 3
```

## 11.5 Migrating Objects

### Aim

To demonstrate how a single item of data can be shared across a number of FileServers. Whenever the data is needed on one FileServer, it must automatically migrate there

from wherever it is currently stored. However, if the data is currently in use, it must migrate as soon as possible after it is freed. Using this mechanism, many computers could share a single counter, or any other changeable data item.

### Method

Class `db.smartroom.server.ProtoMigrator` acts as a proxy for the migrating data item (described henceforth as the 'data object'). Thus, any object which needs access to the data object must communicate with the corresponding `ProtoMigrator` instead. (This `ProtoMigrator` is actually a `RemoteObject`, which can list itself on a FileServer.) When the data object is called for, the `ProtoMigrator` checks whether it is available locally. If not, it then checks the object's 'home' location (where it was first created). The `ProtoMigrator` at the home location either provides the data object immediately, or retrieves it from its current location instead. Thus all object migration (for a particular data object) is directed through the object's home location. Furthermore, the home `ProtoMigrator` always knows the current location of the object.

To implement this class, however, synchronization issues must also be taken into account [34]. For example, the implementation must ensure that it is impossible for a deadlock to result, even if many threads are current active within each `ProtoMigrator`, even when the `ProtoMigrators` (which share a common home) are located on different FileServers.

To run this example, run '`rmiregistry`' on computer XYZ, as in experiment 11.2. Then execute '`MakeServer.bat XYZ`' on any number of computers, and finally '`Migrate.bat XYZ`' on one computer.

### Implementation

#### The requirements of a migrating object

Class `db.smartroom.server.ProtoMigrator` allows a single object to be shared across a number of FileServers. In order for this to work, different copies of the `ProtoMigrator` object (on the different FileServers) must coordinate their activities to keep track of the floating object. To do this, each `ProtoMigrator` is allocated one of five essential states:

1. Before registration — state $B+$

2. Home with object — state *H+*
3. Home without object — state *H-*
4. Elsewhere with object — state *E+*
5. Elsewhere without object — state *E-*

In each of these states, a `ProtoMigrator` can receive any of the following requests:

1. Obtain the object (`getMigrator`)
2. Relinquish the object (`communicate`)
3. Serialize yourself[1]
4. Register with local FileServer (`register`)

Furthermore, each `ProtoMigrator` must ensure that its state remains consistent, even in the face of many, simultaneous requests. It must also ensure that it never enters a deadlock situation, in which two copies of a particular object are each engaged in waiting for the other to complete mutually exclusive tasks.

### Details of object states

Each `ProtoMigrator` object retains its state in its data variables. There are five essential variables.

`home` is inherited from `AbstractRemoteObject`, and stores the FileServer and the name which together uniquely identify this object. If this variable is null, then the state is *B+*. Otherwise, the FileServer referenced by `home` will always hold an instance of this particular `ProtoMigrator`, and thus `home` is used as a base for accessing the migrating data object. `home` is serialized normally; its state is preserved when the `ProtoMigrator` is serialized.

`transport` is used whenever the data object needs to be moved from one computer to another. It is serialized normally, and is used only in state *B+*. At all other times, its value is null.

[1] In the case of a `ProtoMigrator`, the serialization is automatically performed — however, some data fields are allowed to be serialized, while others are not, and in this way, serialization enables the state of the transmitted object to change. Serialization occurs whenever the object moves from one JVM to another. Serializing does not remove the original object, but instead creates another copy of it.

`gotObject` is a boolean variable, which states whether or not the data object is present in the `myObject` field. It is not serialized, and will thus revert to false whenever the `ProtoMigrator` is serialized.

`myObject` stores the data object, whenever it is accessible at a particular computer. In states *H-* and *E-*, its value is null. In states *H+* and *E+*, it holds the object and is non-null (unless the data object is null). In state *B+*, its value is unimportant. This is a transient field, and is not serialized; instead `myObject` is automatically made null on deserialization.

`currentHome` represents the last known location of the data object. This field is non-null only in state *H-*, since only the home `ProtoMigrator` needs to keep track of the data object's location when it is not at home. This field is not serialized.

From this, a table of states versus data item values can be produced, as shown in figure 11.1. (The column entitled 'at home?' shows whether the locally accessible FileServer is the same as the FileServer contained in the `home` field[2].)

| State | `home` | `transport` | `gotObject` | `myObject` | `currentHome` | at home? |
|---|---|---|---|---|---|---|
| *B+* | null | data object | true/false | anything | null | no |
| *H+* | non-null | null | true | data object | null | yes |
| *H-* | non-null | null | false | null | non-null | yes |
| *E+* | non-null | null | true | data object | null | no |
| *E-* | non-null | null | false | null | null | no |

Table 11.1: `ProtoMigrator` state and corresponding data item values.

Thus each state may be characterised by the values of its data items, as in figure 11.2.

| State | Characteristic expression |
|---|---|
| *B+* | `home == null` |
| *H+* | `(home.getFileServer() == local FileServer) && (gotObject)` |
| *H-* | `(home.getFileServer() == local FileServer) && (!gotObject)` |
| *E+* | `(home.getFileServer() != local FileServer) && (gotObject)` |
| *E-* | `(home.getFileServer() != local FileServer) && (!gotObject)` |

Table 11.2: Characteristic expression for identifying migrator states.

[2]The FileServer locally accessible to the `ProtoMigrator` is not a data item, but is accessible to all objects, and thus represents part of the state accessible to the `ProtoMigrator`.

### Object request methods

`getMigrator` is a request that the data object be moved to this computer. This request will usually be issued internally by a subclass of `ProtoMigrator`, to obtain a lock on the data object. It is also typical to synchronize on the `ProtoMigrator` before calling `getMigrator` (itself a synchronized method[3]).

`communicate` is used internally by `ProtoMigrators` to communicate with each other. (`communicate` is part of the `RemoteObject` interface.) One `ProtoMigrator` would call the `communicate` method of another, in order to retrieve the data object. The parameter passed to the receiver would be the FileServer on which the sender was located, and the return value would be the data object.

`readResolve` Serialization is handled automatically by Java and the FileServer. When a `ProtoMigrator` is deserialized on a FileServer, it immediately registers itself in the FileServer's local file list. The `home` and `transport` fields are serialized normally, while the other fields revert to their default values (false or null) when serialized.

`register` is called by the local FileServer when a particular object is created on that FileServer for the first time. The object's new `home` is passed as a parameter, and the returned local object reference will be stored in the FileServer's file list.

### Migration protocol

**State *B+*** When a `ProtoMigrator` is created, it begins in state *B+*, and stores its data item in the `transport` field.

`getMigrator` will result in the data object being copied to the `myObject` variable, and `gotObject` being set to true.

`communicate` will never be called.

Serialization will not change to object state.

[3]Synchronizing on an object is a means of ensuring that two processes cannot access the same object simultaneously. If one thread synchronizes on the object, and then a second thread tries to do the same, the second thread will be blocked until the first thread releases its synchronization. A synchronized method of an object automatically synchronizes on the object when it begins, and releases the synchronization when it has finished.

When `register` is called, `home` is assigned the new home location (which is guaranteed not to be null), `myObject` is overwritten by `transport`, `gotObject` is set to true, `transport` is set to null, and the state becomes *H+*.

**State *H+*** If `getMigrator` is called for a `ProtoMigrator` in state *H+*, no action need be taken, since the data object is already immediately available.

If `communicate` is called, with parameter P indicating the new home, then `myObject` should be returned, while `currentHome` is set to P, `myObject` is set to null, `gotObject` is set to false, and the state becomes *H-*. (The state of the caller will change from *E-* to *E+*.)

Serialization will result in a new object with state *E-*. The original *H+* object will remain and will not be finalized by the Java garbage collection system, since it is referenced in its home FileServer.

`register` will never be called for an *H+* `ProtoMigrator`.

**State *H-*** `getMigrator` will cause the home `ProtoMigrator` to call the `communicate` method of the `ProtoMigrator` at `currentHome`, with a null parameter. (This callee is guaranteed to have state *E+* currently.) The callee will then return the data object (and revert to state *E-*). The data object can then be stored in `myObject`, `gotObject` set to true, and the state becomes *H+*.

`communicate` will be called only by a `ProtoMigrator` in state *E-*, trying to obtain the data object by contacting the 'home base'. In that case, `getMigrator` should be called (to obtain the object), and then the `communicate` procedure of state *H+* should be followed.

Serialization will result in a new object with state *E-*.

`register` will never be called for an *H-* `ProtoMigrator`.

**State *E+*** `getMigrator` will cause no action to be taken — the data object is already available.

If `communicate` is called, `myObject` should be returned, while `gotObject` is set to false, `myObject` is set to null, and the state becomes *E-*. (This call will only ever be made by *H+* or *H-*, and the parameter passed to `communicate` will only ever be null.)

Serialization will result in a new object with state *E-*. Since the original *E+* object is registered with its local FileServer, it will not be deleted and the data will not be lost.

`register` will never be called.

**State *E-*** `getMigrator` will result in a call to the `communicate` method of the `ProtoMigrator` at `home`, with the local FileServer as parameter.

`communicate` will never be called. This is because a `communicate` call is only ever made to an object in state *E+* or an object at its home location (*H+* or *H-*).

Serialization will result in a new object with state *E-*.

If `register` is called, the new home location must be the same as `home` (otherwise an exception may be thrown). No change needs to be made to the state; the `ProtoMigrator` can simply return itself, to be listed in the local FileServer.

### Deadlock prevention

Deadlock prevention is achieved by imposing an implicit order on the resources available in the system [19, p. 226]. Resources must be requested in order, from the top to the bottom of the following list:

1. Synchronize on an instance of the `ProtoMigrator` which does not hold the data object, and is not the home location (i.e. synchronizing on *E-*)
2. Synchronize on an instance of the `ProtoMigrator` which is at its home location (i.e. synchronizing on *H-* or *H+* or *B+*)
3. Synchronize on an instance of the `ProtoMigrator` which holds the data object, but is not at the home location (i.e. synchronizing on *E+*)

In the system, there will be at most one instance of *E+*, exactly one of *H-* or *H+* or *B+*, and any number of *E-*'s. However, since no instance of *E-* ever needs to synchronize on another *E-*, there is no possibility of a deadlock resulting from *E-* synchronizations.

This can also be seen, by observing that a `ProtoMigrator` in state *E-* will only ever initiate calls to other `ProtoMigrators` is states *H-* and *H+*. An *H+* or a *B+* or an *E+* will never initiate such calls. An *H-* will only call an *E+*. Furthermore, no calls by

FileServers to the `register` method rely on subcalls to other `ProtoMigrators`. Thus the resource allocation order will never be violated.

### Possible enhancements

The `ProtoMigrator` class assumes that the network connecting FileServers together is reliable. However, there may be physical failures in the network, preventing transmission while the data object is being moved between FileServers. In that case, a copy of the migrating object should be held back, until transmission is confirmed. If the transmission fails, then the local copy should be kept until the connection can be re-established. (However, if this happens, then the local copy cannot be accessed until the network is repaired. This is because the original data message may have been successfully transmitted, while the reply was lost. If the data object were modified on both the sender and receiver, then an inconsistent state would result.)

### Results

On each computer running a FileServer, a `db.smartroom.server.debug.Migrate` object is created. Each of these objects then displays the value of the shared string. From the sample output below, the messages between `Migrate` objects can be seen, as each in turn accesses the data item.

On the computer where `Migrate.bat` was run, sample output was:

```
Created Migrate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:4598](remote),
objID:[5ac8365e:d90ca5bbb7:-8000, 0]]]]
```

On the first FileServer, the output was

```
Registering MyMigrator@Not_Yet_Registered
Have registered MyMigrator@RemoteReference(0:1)
Have created new file: MyMigrator@RemoteReference(0:1)
Duplicating object
Created duplicate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:4598](remote),
objID:[0]]]]
```

```
Created duplicate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:4605](remote),
objID:[5ac8362b:d90ca5e741:-8000, 0]]]]
Created duplicate on db.smartroom.server.FileServerImpl_Stub
[RemoteStub [ref: [endpoint:[137.158.135.197:4609](remote),
objID:[5ac8362b:d90ca5e9b8:-8000, 0]]]]
End of duplications
Fetching migrator...Done; value is MyMigrator,
Copy 3@RemoteReference(0:1)
Communicate (state H-) returning MyMigrator, Copy 3
Communicate (state H-) returning MyMigrator, Copy 3
```

On the second FileServer, the output was

```
Registering Data_Not_Present@RemoteReference(0:1)
Have registered Data_Not_Present@RemoteReference(0:1)
Have created new file: Data_Not_Present@RemoteReference(0:1)
Creating a copy on local FileServer
Fetching migrator...Done; value is MyMigrator,
Copy 3@RemoteReference(0:1)
Communicate (state E-) returning MyMigrator, Copy 3
```

On the third FileServer, the output was

```
Registering Data_Not_Present@RemoteReference(0:1)
Have registered Data_Not_Present@RemoteReference(0:1)
Have created new file: Data_Not_Present@RemoteReference(0:1)
Creating a copy on local FileServer
Fetching migrator...Done; value is MyMigrator,
Copy 3@RemoteReference(0:1)
```

## 11.6 Migration with Simultaneous Access

### Aim

To rigorously test sharing a single counter between many FileServers. A carrier object is created on every FileServer; each carrier retrieves and increments the shared counter 1000 times (with random delays between increments). If these delays are set to zero, this test effectively exercises the `ProtoMigrator` class under the harshest conditions possible, as many objects simultaneously attempt to use the same counter.

### Method

Class `db.smartroom.server.debug.MigrateFast` simply extends class `db.smartroom.server.ProtoMigrator`. The extended class creates a new execution thread on each computer where its instance is registered with the local FileServer. This thread then repeatedly increments the shared counter, with an optional delay.

To run this example, run '`rmiregistry`' on computer XYZ, as in experiment 11.2. Then execute '`MakeServer.bat XYZ`' on any number of computers, and finally '`MigrateFast.bat XYZ DELAY`' on one computer, where DELAY is the maximum value of the delay in milliseconds between counter increments. If the DELAY parameter is omitted, it defaults to 0.

### Results

Even when the delays are set to zero, the programs still continue to operate, and correctly interlock in incrementing the counter.

## 11.7 Demonstration of Graphical RemoteObjects

### Aim

To demonstrate that RemoteObjects can at the same time be graphical objects.

### Method

Class `db.smartroom.viewer.debug.TestRemoteJPanel` is a subclass of class `db.smartroom.viewer.ProtoRemoteJPanel`, itself a subclass of `javax.swing.JPanel` — one of the basic graphical objects in the new Java graphics hierarchy called 'Swing'.

A `JPanel` is 'a generic lightweight container', according the Java API specification; it holds other graphical objects together, in order to display them. A `JPanel` is not automatically associated with any screen real estate. Instead, whenever an instance of the new class `db.smartroom.viewer.debug.TestRemoteJPanel` is registered on its local FileServer, it automatically creates a `JFrame` in which to display itself.

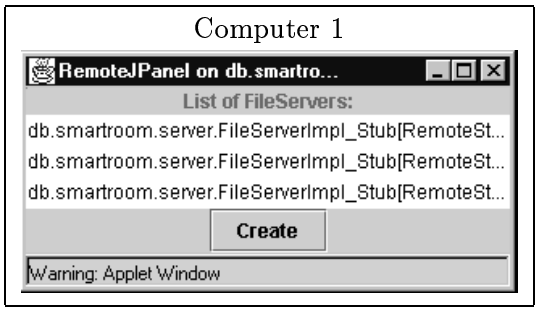
Each `TestRemoteJPanel` displays a list of FileServers, and has a 'create' button, which instructs the object to duplicate itself to the selected FileServer. `TestRemoteJPanel`'s main method creates a `TestRemoteJPanel` object on the first FileServer it can find. When that object is registered, it creates a new window to display itself. By selecting a FileServer from the list, and clicking the 'create' button, the user can instruct the object to create a duplicate of itself on that FileServer.

To run this example, run '`rmiregistry`' on computer XYZ, as in experiment 11.2. Then execute '`MakeServer.bat XYZ`' on any number of computers, and finally '`TestRemoteJPanel.bat XYZ`' on one computer.

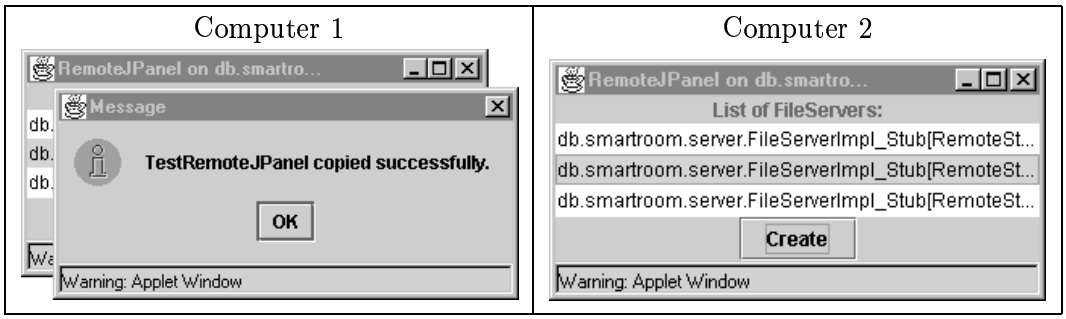
### Results

For this demonstration, `MakeServer` was run on three computers (Computer 1, Computer 2 and Computer 3), then `TestRemoteJPanel` was called on a fourth computer.

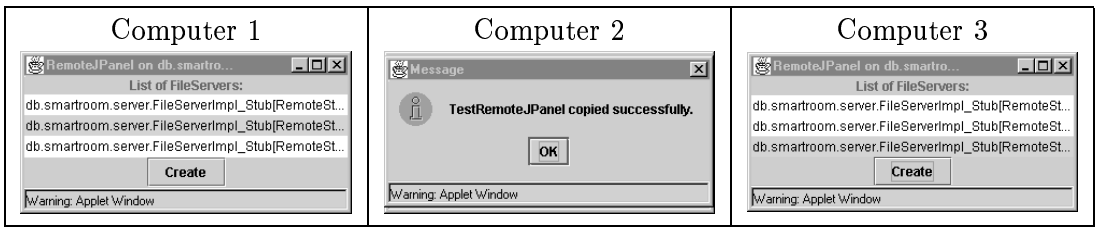
1. When the main program of `TestRemoteJPanel` was started, the following window appeared on the screen of one of the server computers, Computer 1. It shows that there are three FileServers available (including the one on Computer 1). The other computers showed no Java Applet Windows.



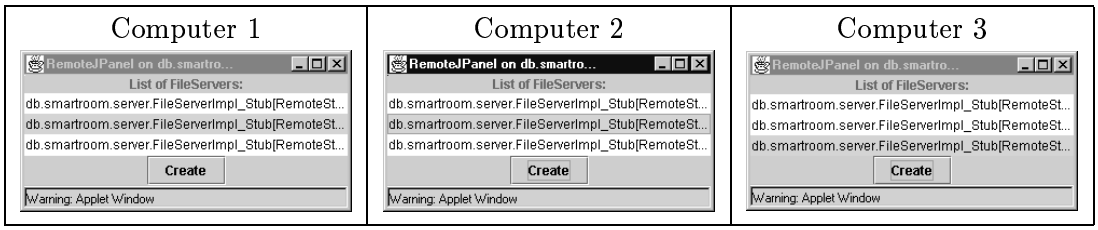

2. The second item in the list of FileServers was selected, and the create button was pressed. This caused a copy of this `RemoteJPanel` object to be created on Computer 2, and a message to be shown on Computer 1. The resulting screen displays were:




3. Next, the third FileServer was selected in the window on Computer 2, and the create button was pressed. This made a copy of the `RemoteJPanel` to Computer 3, giving:




4. Finally, the third item in Computer 2's window was selected again. However, this time it made no change to the windows. This proves that once a particular `RemoteJPanel` has been created on a computer, attempting to create another copy of the same instance will have no effect — the existing copy takes precedence.




This experiment shows that graphical objects can be included easily into the distributed object system. It demonstrates that the system can feature interactive objects, as well as objects with predefined behaviour. This is very useful, because it allows a user to directly modify information in the system, and view new information as it appears. The user could be an operator, monitoring the system state, or there could be a group of users working cooperatively, from different computers.

## 11.8 Graphical RemoteObjects Wrapping Other RemoteObjects

### Aim

To show how a graphical object can act as an intermediary for another RemoteObject.

### Method

Class `db.smartroom.viewer.debug.TestWindow` trivially extends `db.smartroom.viewer.demo.CounterWindow`. A `CounterWindow` displays and allows interactive control of any `db.smartroom.viewer.demo.Counter` object that implements the `RemoteObject` interface. (This includes migrating or replicating Counter implementations). The `CounterWindow` registers itself separately as a RemoteObject, so the `Counter` which it is viewing is moved independently from the viewing window. However, this class does itself implement the `Counter` interface, so if it is used in place of the `Counter` which it is mirroring, the original `Counter` and the viewer will migrate together.

Thus the `CounterWindow` acts as a stand-in for the `Counter` which it carries. Wherever the `CounterWindow` goes, the original `Counter` will follow. (This essential behaviour is contained in `db.smartroom.viewer.ProtoCarrierJPanel`; `CounterWindow` extends it

by adding a graphical interface for displaying and modifying the value of the counter, and also by adding the methods of the `Counter` interface.

#### Results

This example shows how a user interface can be added transparently to a data class, without modifying any of the class's code. This separation of data and interface simplifies programming, and corresponds to the Model-View-Controller (MVC) architecture. The Model-View-Controller (MVC) architecture was introduced by the Smalltalk language developers, for creating applications with user interfaces [35]. One aim of this design pattern is to separate the information which is being manipulated (the 'model') from the user's 'view' of the data (typically within a window), and from the 'controller' (through which the user manipulates the data). These three portions are essentially independent, with only limited interactions between them [22, p. 626].

Transparently adding a graphical display not only simplifies debugging (since the graphical class can stand in for the original in any situation); it also allows many different visual representations (views) of the same object, in different parts of the screen.

An alternative approach would be to write a subclass of a particular class which implements the `Counter` interface, to act as its stand-in (and provide a graphical representation). However, this subclass could then interface to only that particular class, and not to any other class that implements `Counter`. The advantage of the solution used in this example is that it can display any `Counter`.

## 11.9 Distributed Processing

#### Aim

This experiment shows that distributed objects need not be simple data receptacles. It demonstrates how a computation may be distributed across a group of FileServers.

#### Method

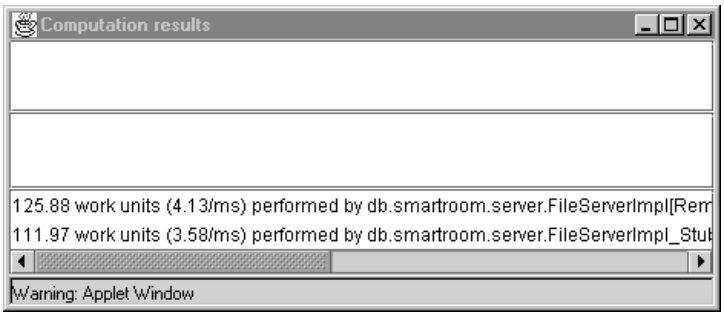
Wherever an instance of `db.smartroom.server.debug.PrimeFinder` is registered with a FileServer, it starts an execution thread (of very low priority, so as not to impinge on other processes) to calculate prime numbers.

If the RemoteObject is being registered for the first time, a window is created. This window displays the number of new prime numbers found, and the rate at which prime numbers are being found. If it is a subsequent registration, communication is entered into with the `PrimeFinder` at the home FileServer, which allocates blocks of prime numbers to be computed, and collates results in its window.

To run this experiment using two computers, run '`rmiregistry`' on computer XYZ. Then '`MakeServer.bat XYZ`' once on one computer and (optionally) once on another. Then run '`PrimeFinder.bat XYZ`' from anywhere.

### Results

1. When `MakeServer` was run on two computers (Computer 1 and Computer 2), before `PrimeFinder` was run, the window shown below appeared on Computer 1, counting up the number of work units performed in finding primes on both Computer 1 and Computer 2.




   The computers had processors of different speeds, which explains the different average computation rates (in brackets). On Computer 1, the text output was

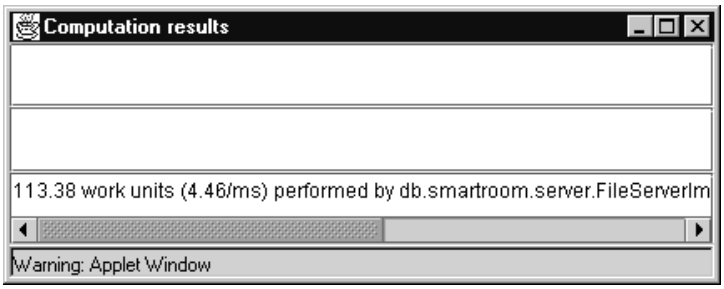
```
Have registered PrimeFinder@RemoteReference(0:1)
Duplicating object
FileServerLister: Ping received
Created duplicate on db.smartroom.server.FileServerImpl_Stub[RemoteStub
[ref: [endpoint:[137.158.135.197:2531](remote),objID:[0]]]]
Created duplicate on db.smartroom.server.FileServerImpl_Stub[RemoteStub
[ref: [endpoint:[137.158.135.191:1053](remote),objID:[6853172e:
da6cb782db:-8000, 0]]]]
End of duplications
About to do computation
Finding primes
```

On Computer 2, the output was

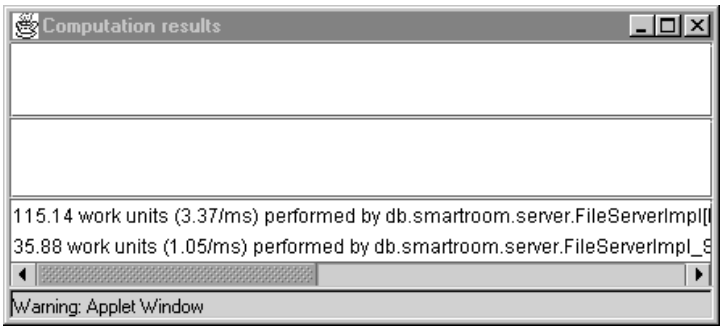
```
Have registered PrimeFinder@RemoteReference(0:1)
About to do computation
Finding primes
```

This shows that both computers were participating in the distributed computation.

2. When `MakeServer` was run on just one computer, the new window display was:




3. Finally, when `MakeServer` was run twice on the same computer before `PrimeFinder`, the output was:




In summary, when `MakeServer` is run on two computers, the rate at which prime numbers are computed is double the rate when just one computer has a FileServer. Conversely, when `MakeServer` is run twice on the same computer, the combined rate of computation is the same as for a single server, since the servers must share a single processor.

This clearly demonstrates one of the advantages of distributed computing, and also proves that the architecture developed can effectively use this mechanism.

## 11.10 Conclusion

These examples demonstrate that this distributed object system can effectively resolve many of the challenges of distributed systems, mentioned in part I. This chapter has shown the creation, migration, and intercommunication of objects, and tested the architecture developed in chapter 10.

# Chapter 12

# Conclusion

The clearly specified object-oriented architecture, developed in this part, gives distributed objects the ability to control their own migration, enabling them to implement sophisticated migration strategies.

Because a group of conventional objects can be treated as a single totally distributed object, programming of distributed systems is greatly simplified.

The experiments prove that the implementation of the architecture works according to its specification, and that the specification provides a useful environment for distributed objects to inhabit. Finally, the experiments also serve as prototypes for further programming in part IV, since they illustrate many of the essential strategies employed by distributed objects.

# Part III

# A Building Simulator

# Chapter 13

# Introduction

A Smart Building can demonstrate the usefulness of distributed objects in a real pattern analysis situation. A simulated building is needed to provide a rich and powerful source of data for this Smart Building.

The building simulator developed in this part enables complex behaviour to be simulated with ease. Virtual cameras can observe this, and injected information into the Smart Building, just as real cameras will do once a physical Smart Building has been built. Complex interactions between objects are achieved using a flexible two-phase message passing protocol, which allows potential actions within the building to be proposed and amended, before they are allowed to occur.

Chapter 14 explains the need for a simulated building, and outlines how it models a real building. Next, it identifies the primary participants in the simulated building, and their roles. It also justifies the choice of a complex model for the simulation, as necessary for realism and flexibility. Finally, chapter 15 shows how the building was implemented using Java, highlighting the design considerations of the implementation, and showing a sample run.

# Chapter 14

# Simulation of a Building

To demonstrate the feasibility of a 'Smart Building', a program was created in order to simulate a building and its inhabitants. This program produces sample data for testing how the Smart Building reconstructs what happens inside it. However, its use extends beyond this, too; since the objects in the simulated building directly parallel those contained in the real building, the simulated objects can be replaced directly with their real equivalents when the real building is complete.

## 14.1 Overview

The simulated building consists of a collection of rooms, doors between rooms, obstacles, cameras, and people with programmed behaviour. This can all be displayed graphically, so that the activities within the room can be seen clearly. When a sensor in the simulated building detects that something has happened, it communicates that message to the appropriate FileServer. The RemoteObjects there then attempt to reconstruct what happened. This is exactly the same as what a real sensor would do, when it detected a change in its environment.

Inhabitants of the simulated building interact by broadcasting their intended actions to all of the other inhabitants. Any inhabitant may decide to veto the action; if none does, then the action is allowed to occur. This protocol enables complex behaviour to occur in the simulated building. Since the inhabitants receive feedback from their actions, they can use this information to change their plans if their intended actions are disallowed. They can also observe the behaviour of the other objects, and use this to interact with them. For example, two simulated people could decide to pause briefly if they met in a

room.

## 14.2 Structure of the Simulated Building

The simulated building provides an environment in which simulated objects can interact, just as a distributed object system provides an environment for distributed objects. Objects in the building interact by broadcasting messages to each other describing their activities; therefore, the simulated building provides the mechanism for sending and receiving these messages.

Central to the simulated building is an event passing system, through which all activities in the building flow. This mechanism also allows activities to be proposed and amended or vetoed, before they take effect. The users of this are the simulated objects which send, receive and interpret these events, and it is they that represent the information generated by the building.

The simulated building consists of four essential groups of objects:

1. 'Events'
2. 'Simulated Objects'
3. 'Viewers'
4. 'Coordinating Objects'

Within the simulated building, Events are used for all communications between Simulated Objects. They might signify, for example, that movement is occurring or that time is passing — or they could carry any other piece of information that other objects should be made aware of. All Events are implemented as subclasses of class `ActivityEvent`.

Simulated Objects represent actual objects in the real world — both sensors, and objects that might interact with the sensors. Typical sensors could be cameras analysing motion, face detection systems (which identify a person from a video camera), and swipe card readers in doorways. Significant other simulated objects include virtual people (to trigger the cameras) as well as those objects which interact with the people (such as room boundaries and obstructions within a room). Simulated Objects are subclasses of `NamedObject`, and implement `ActivityListener`.

Viewers create graphical representations of the Simulated Objects. This facilitates comparison between simulated behaviour and behaviour reconstructed by the Smart Building. Viewers can also graphically display those events which activate sensors (such as movement), transparently overlaying a plan view of a room. Finally, Viewers can allow the user to interactively control Simulated Objects, with a mouse. All Viewers are subclasses of `java.awt.Component`, and they typically implement `java.util.Observer`.

Coordinating Objects are responsible for ensuring that Events are sent to all Simulated Objects, that Simulated Objects are allocated appropriate Viewers, and that Viewers are associated with screen windows correctly. They include the `Building` simulator object, and the controller of `Views`.

## 14.3 Justification of Complexity

It would certainly be simpler to test the Smart Building with a predefined sequence of motion and face detection events instead. However, that approach imposes severe limitations on the scope of the simulation:

**Verification** A predefined sequence of events must be generated and inspected manually. With a sophisticated model, complex event sequences can be generated automatically. Furthermore, these sequences can be shown graphically, to clarify the meanings of the events.

**Validation** If events are generated manually, then only a few pre-defined scenarios will be considered by the Smart Building system. This increases the risk that flaws in the system will go undetected, until real sensors are attached. With an automatic system, in contrast, many slightly varying scenarios can be used to test the system. In addition, behaviour within the system can be specified at a much higher level than with the manual system.

**Realism** A predefined model must be rewritten almost completely in order to refine the models of the sensors within the building. With a more advanced simulation, only the sensor modules must be rewritten; everything else can remain unchanged. (For example, if it were decided that a real camera could see only part of a room, it would be simple to implement a corresponding simulated camera.)

The sophisticated model can also prevent unrealistic scenarios from occurring — for example, it can guarantee that people do not walk through desks or partitions between offices — while the simple model offers no such sanity checks.

**Extensibility** The sophisticated building can be reconfigured easily, to contain new objects or more people. In contrast, the simple model requires an entirely new scenario to be devised.

It is essential for the simulation to be as realistic as possible, to simplify the transition to a real building. Furthermore, an accurate simulation can be more convenient than a real building, since sample data for a particular scenario can be produced automatically, whenever it is needed.

The building simulator used in this project is indeed complex. However, this complexity offers realism, extensibility and flexibility which could not otherwise be achieved.

## 14.4 Conclusion

In this chapter, we have developed a framework for a building simulation that will allow application of the distributed system to a concrete problem — that of a Smart Building. The complexity and realism of the simulation will verify the Smart Building as a meaningful distributed pattern recognition problem.

# Chapter 15

# Implementing a Simulated Building

This chapter shows how a building simulator was implemented using Java, as a source of data for a computerised Smart Building. The mechanisms which coordinate all of the simulated actions in the building are defined, and the design considerations of this particular implementation are explored.

## 15.1 Class Overview

The classes which make up the simulated building are spread across four packages.

`db.smartroom.simulator` contains the classes which describe the internal geometry of the simulated building, and also the classes for each of the actual simulated objects within the building.

`db.smartroom.simulator.event` provides classes and interfaces for dealing with the activities which occur in the simulated building.

`db.smartroom.simulator.viewer` enables the activities within a simulated building to be displayed and controlled graphically.

`db.smartroom.simulator.debug` contains classes for testing and demonstrating the operation of the building simulator.

In package `db.smartroom.simulator`:

- Classes `Point`, `Dimension` and `BoundingBox` provide geometrical abstractions of the three-dimensional space which constitutes the simulated building. These classes make it simple, for example, to find the midpoint of a region in which movement has taken place (a `BoundingBox`), or determine whether two movements overlap.

- Class `NamedObject` is a superclass for all inhabitants of the simulated building. This is a convenience class, which enables objects to be identified by a name rather than by a cryptic memory reference. Its subclasses include `Room`, `Obstruction`, `Person`, `Door` and `Camera` objects within the simulated building. Since each activity takes place within a `Room`, many of these objects reference their `Room` in the simulated building.

- Class `Building` operates a virtual building, and the chronological events within that building. It is responsible for passing proposed activities to the inhabitants of the simulated building, and collating the responses.

In package `db.smartroom.simulator.event`:

- Class `ActivityEvent` a superclass for all activities which can take place in the simulated building. Activity subclasses include `ClockTickEvent`, `MotionEvent` and `ImpossibleEvent`.

- Interface `ActivityListener` defines how inhabitants of the simulated building are informed of new `ActivityEvents`.

- `ActivityReplaceException` is the exception which `ActivityListeners` throw to prevent a proposed `ActivityEvent` from taking place.

In package `db.smartroom.simulator.viewer`:

- Class `View` projects three-dimensional objects (from package `db.smartroom.simulator`) into two-dimensional Java graphics objects. This class could implement any projection, to give plan or perspective views of a room.

- Class `ViewController` operates and manipulates a single `View` of the simulated building. It is an extensible abstract factory [29, p. 87,91] which defines which viewer object to associate with each object in the building.

- `VisibleBuilding` extends the `Building` class, to create a graphical display automatically for each new building inhabitant. It can also support multiple simultaneous views of the building, using many `ViewControllers`.

- Classes `ActivityViewer`, `ObstructionViewer`, `PersonViewer` and `RoomViewer` provide graphical representations of `ActivityEvent`, `Obstruction`, `Person` and `Room` objects respectively. The `PersonViewer` class also allows the motion of a person to be controlled by the user, by clicking the mouse on the image of the person.

In package `db.smartroom.simulator.debug`:

- `GeometryTest` is a test suite for demonstrating that the geometry classes of the simulated building perform correctly. This class uses the JUnit [36] testing framework to automate these tests, so that they can be performed each time the code is modified and recompiled.

- Class `TestBuildingViewer` creates a simple two-roomed simulated building, with two inhabitants, and runs it. Section 15.5 shows how to run this demonstration.

## 15.2 Operation of the Building

The operation of the building simulator is controlled by a single thread in class `Building`. This thread is responsible for maintaining a central building clock, and for notifying objects within the building whenever anything occurs.

Events are used to signal to objects in the building (called ActivityListeners) that some activity has occurred, of which they might need to take note. Most events are issued in two steps. When an event is to occur, the `sendActivityEvent` method of the building class is called.

First, the building calls the `vetoableActivity` method of each `ActivityListener`. This enables other listeners to be fore-warned about a potential activity, and to veto it if necessary. Each other listener may also suggest a compromise replacement activity, if it wants to. Then, if the activity is not vetoed or if consensus is reached about a compromise, the activity is deemed to have occurred, and the `activityOccurred` method of each `ActivityListener` is called by the building. If the activity is vetoed, and no compromise could be reached, then no event occurs. In either case, the process

which attempted to produce the event is notified of the result: either the event which occurred is returned, or a special event of type `ImpossibleEvent` signals a non-event.[1]

As a concrete example, assume that a Person object represents a virtual person walking within the virtual building. Now assume that the person intends to take a virtual step forward. If there is no obstruction ahead, the step should be allowed to proceed. However, if there is an obstruction, then the action should be replaced with a suitable substitute. In terms of the building simulator mechanism, the `vetoableActivity` method of each `ActivityListener` within the building will be called by the Person object. The obstruction (itself a listener within the building) learns of the proposed event. If the event would result in the person overlapping the obstruction, then it should veto the activity; otherwise, it should allow it to proceed. In this way, the system can be prevented from entering into an impossible state.

This mechanism also facilitates more complex interactions between the denizens of the virtual building. For example, when a virtual person walked to a door, this event would be transmuted to teleport the person into the next room. (This is needed because the rooms in the building are assumed to be independent spaces.) A more sophisticated model could ensure that motions in one room would be seen by cameras in the next room, only if the door connecting the rooms were open.

The suggested replacement actions are also tested for feasibility, before being allowed to occur. For example, if a door were to suggest that a person's movement should be replaced by a teleportation to the next room, and there was already a person standing in the doorway of the next room, then that person's object would block the teleportation, and both the replacement activity and the original activity would fail (unless another substitute activity were suggested).

The exact protocol for choosing a suitable replacement event is given below.

1. Set `bestEvt` to the proposed `ActivityEvent`.
2. For each listener:
   (a) Call `vetoableActivity(bestEvt)`
   (b) If the listener throws a replacement event (other than `ImpossibleEvent`), replace `bestEvt` with this new event.

[1]The two-phase notification mechanism of this class extends the technique of the `java.beans.PropertyChangeEvent` class of the Java 2 platform. This class has two listener interfaces `java.beans.VetoableChangeListener` and `java.beans.PropertyChangeListener` and exception `java.beans.PropertyVetoException`. These listeners allow the properties associated with various Java beans to be constrained, since they can veto unacceptable property values.

3. After vetoableActivity has been called for all listeners, there are three possible outcomes:

   (a) If everyone agrees (throws no exception), then the action is approved — return `bestEvt` to the caller.

   (b) If everyone either agrees or throws `ImpossibleEvent`, then the action is disallowed — return an `ImpossibleEvent`.

   (c) If any listener has offered a replacement event, go back to step 2 (using the replacement event). This repetition may occur at most `n` times, where `n` is the number of listeners. Otherwise, return an `ImpossibleEvent`.

This protocol guarantees that each listener is given a chance to suggest a replacement event.

## 15.3 Building Design Considerations

This building simulation makes assumptions that all activity within the building takes place within a single thread, that each room has its own coordinate system, and that events are essentially independent of each other. This section justifies these assumptions, and explores their implications and the limitations which they impose on the simulation.

### 15.3.1 Single Threaded Operation

The operation of the building simulator is single threaded. This is desirable for several reasons.

**Display and Interaction** It is useful for complete information to be available about the building. It is needed by viewers which display the simulated building, for comparison with the distributed reconstruction of the building. It also simplifies interactive control of the virtual building.

**Speed** If the simulator were multi-threaded, or even itself distributed across many computers, the interdependencies of the occupants would result in numerous locks and inter-process synchronisations. This is especially true because each action in the building potentially affects all of the other object within the building, especially those within the same room. (If each simulated room were confined to a single

place, this restriction would not be as severe. Because the building simulator is event driven, it would be possible for it to run in a distributed fashion, but it would not be convenient.)

**Simplicity** Because there is only a single thread of execution, programming and debugging of the simulator is simplified. This also makes it simpler to avoid inconsistent states which might result if two actions were performed simultaneously, and eliminates deadlock issues.

**Integrity** The simulated building should remain independent from the distributed engine attempting to reconstruct the activities within the building. Furthermore, the simulated building should be simpler than the reconstructor; particularly, it should not rely on the mechanisms which the reconstructor is testing — otherwise, it may be impossible to distinguish flaws in the reconstructor from flaws in the simulator.

### 15.3.2 Independent Coordinate Systems

A further assumption which is made in the implementation of the building simulator is that each room has an independent coordinate system. Although this is not true in physical terms, it eliminates the need for a global Cartesian coordinate frame. Furthermore, it allows a building to be designed as a combination of rooms, positioned relative to each other by the placement of doors. This more accurately models the real building which would eventually replace the simulator; in the real building too, it would be more convenient to specify the rooms relative to each other, instead of as coordinates relative to a global origin.[2]

This assumption also simplifies the design of objects within rooms; instead of scrutinising all activities that occurred within the building, they could ignore all activities that occurred in other rooms, and concentrate only on those within their room. Having self-contained rooms does make creating doors between rooms more complicated — a person must jump instantaneously from one room to the next. However, a door object can cause a person to seem to be in two places at once, by creating phantom motion events on the other side of the door from the person, and by preventing any other person from walking through the door (or very near to it), while a person is standing in the doorway. Section 15.4.3 contains more detail about how doors are implemented in the simulated

[2] In a real building, it is convenient to allow each camera to use its own coordinate scheme, and separate the image analysis performed by the camera from the correlation of points from different coordinate schemes.

building.

Finally, the assumption of independent coordinates has the further advantage that it enables non-standard topologies to be programmed into the building simulator easily. For example, a lift could be programmed as a special type of room, which sometimes adjoins one room, and sometimes another.

### 15.3.3 First Order Event Model

Events in the system are assumed to be atomic and self-contained. In other words, the simulated building does not allow for conditional event combinations. The only exception to this rule is the fact that each event may be vetoed, or a replacement event may be suggested, before the event is allowed to occur.

Secondly, the current design does not allow a new `ActivityEvent` to be sent from within a call to `vetoableActivity`. Otherwise, it might be that the activity currently being vetted had already been given the stamp of approval by some of the `ActivityListeners`. If the new `ActivityEvent` caused the system state to change, then the vetted activity might no longer be legal, yet it could nevertheless occur, since the affected `ActivityListeners` would not necessarily be consulted again.

In essence, such a call would contradict the premise of a single thread of operation in the building. It is essential that the system state does not change within a call to `vetoableActivity`. However, sending a new `ActivityEvent` within a call to `activityOccurred` does not cause the same inconsistencies, and is therefore allowed. In addition, calling `firevetoableActivity` from within `vetoableActivity` is also allowed, since the new call will not cause the system state to change. (This corollary is used by door objects, to allow a person to appear to be in many places at once — the details of this are in section 15.4.3.)

Another issue with the current consensus model is that it is non-deterministic, in that the order in which `ActivityListeners` are polled for their views on a particular event is not pre-specified. In certain cases, this may result in a sub-optimal replacement activity, or even in an action being vetoed by all listeners, when a compromise could possibly have been reached [37].

There are two reasons why this is not a serious limitation. Firstly, there are few interactions which affect many objects simultaneously within a building, so the order of polling is seldom an issue. The second consideration is this: even if two different possible

outcomes were discovered, making a choice between them would require a mechanism for comparing their relative worth, which is as difficult a problem again.

The only way to overcome these shortcomings would be to use a far more sophisticated event model. This could allow generalised event roll-back — to enable nested events and contingent event sequences. In this way, compound, complex actions could be produced, such as conjunctions or disjunctions of other actions.

The building simulator design is chosen from the continuum of designs between a simple, predefined sequence of events and a complex and completely general model of the building. The compromise used in this simulation maintains as simple a model as possible for the building simulator, but still allows interactions between objects in the building, and guarantees that impossible states do not occur.

## 15.4 Simulated Objects

This section describes and justifies how the Simulated Objects in the virtual building were implemented. Each of these objects implements the `ActivityListener` interface; thus each object must implement `vetoableActivity` and `activityOccurred` methods, to enable it to interact with the other objects in the building.

### 15.4.1 Obstructions

Obstructions represent those regions of a room where no movement may occur. Objects such as desks and office partitions can be modelled effectively in this way. The behaviour is implemented by Java class `Obstruction`; if any `MotionEvent` which would intersect the obstruction is proposed, the `vetoableActivity` method will veto it. (The `activityOccurred` method does nothing, since the only way to reverse inappropriate activities is through the `vetoableActivity` method.)

### 15.4.2 People

Within the simulated building, people are just the same as moving obstructions. As a result, class `Person` extends class `Obstruction`. When other objects are moving, the `Person` object behaves in the same way as its superclass. However, whenever time passes and a `ClockTickEvent` activity occurs, the `Person` object generates its own activity, signalling its intention to move to a new location. While this is happening, the

`vetoableActivity` method changes its behaviour to accept all movements — otherwise the activity would be vetoed by the very `Person` object which proposed it in the first place.

If the motion is allowed to occur, then the position of the `Person` object is updated, and the object reverts to its earlier behaviour.

### 15.4.3 Doors

Doors have very special behaviour, which effectively allows them to connect the different spaces of two rooms together. The doorway behaves as a tiny interstitial space between two rooms, so that a virtual person standing in the doorway seems to be in two places at once.

If a person walks close to a doorway, they are automatically transported into the doorway's space. This prevents them from being stopped from moving when they reach the edge of the room overlapped by the doorway. All movements and potential movements that occur in the doorway's space are mirrored into the two rooms on either side of the doorway. This ensures that a person moving in the doorway does not bump into objects on either side of the door.

When a person in the doorway reaches the edge of the door's space, they are again transported into the appropriate room space instead, and the doorway plays no further role.

### 15.4.4 Virtual Sensors

Virtual sensors correspond directly to sensors in the real world. As a result, the Smart Building can be tested with the output of virtual sensors; later, the virtual sensors can be replaced with real sensors. Ideally, these virtual sensors should behave as realistically as possible, to smooth this transition, and also to ensure that the simulated results are reliable.

However, this transition need not be made instantaneously; it is quite possible to intersperse both real and simulated objects. Thus, a single real 'Smart Room' could be treated as part of a virtual Smart Building, or extra virtual people could be included in a real room, in order to test the system's behaviour beyond the limits of existing hardware.

In this simulation, the primary virtual sensor is a `Camera` object. This object detects all `MotionEvents` within its `Room`. A certain, configurable percentage of these are randomly selected and faithfully recorded, and injected into the Smart Building, while the rest are assumed to be failed detections. Similarly, spurious motion artefacts are occasionally produced, at a configurable average rate, and also injected into the Smart Building. Finally, the virtual cameras add random jitter to the boundaries of the detected movement.

This model is intended to approximate the behaviour of a motion detector connected to a real camera within a room. A more advanced simulation of a camera could take into account that two people moving might cause the camera to register only a single, large motion, instead of two separate motions.

## 15.5 Sample Run

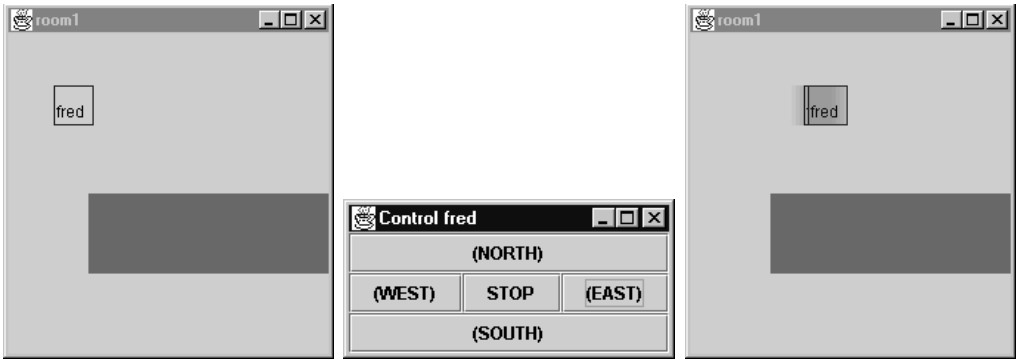
To demonstrate the simulated building, run the batch file '`TestSmallVisibleBuilding.bat`' in the `db/smartroom/simulator/debug` directory. This creates a simple simulated building, consisting of two room connected by a door. The building also contains two virtual people, 'jill' and 'fred', with pre-programmed paths.

### Single Room Demonstration

This section shows the movements of one person in a single room, in figure 15.1.

1. A simple room is created, shown as a window. The room contains one person — the rectangle labelled 'fred' — and also an obstruction in the lower right corner.
2. If the user clicks on the 'fred' rectangle, a new window appears for controlling Fred's movement.
3. If the east control button is pressed, Fred begins to move east in the simulated room. The shaded trail (coloured blue) is drawn by the `ActivityViewer` class, to show the sequence of `MotionEvent` objects generated over the last few clock intervals. The head of the trail is light red, showing Fred's most recently proposed movement.
4. If the south control is pressed, Fred begins to move south.

5. Fred moves south until he eventually reaches the obstruction, which prevents him from moving any further.




(a) A simple room (b) Controlling Fred (c) Fred moves east

(d) Fred moves south (e) The obstruction stops Fred

Figure 15.1: The movements of one person in a simulated room

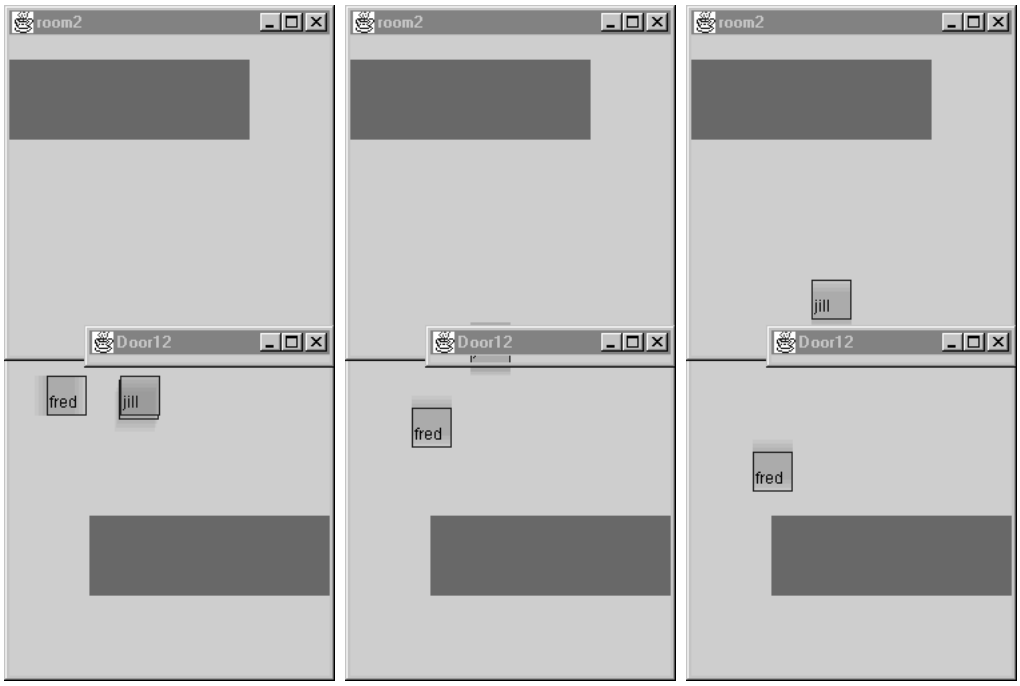
### Double Room Demonstration

Figure 15.2 shows the simulated building created by `TestSmallVisibleBuilding`, and illustrates how doors operate.

1. There are two rooms, and a door which connects them. In the screen snapshot, they are shown as three overlapping windows, with the door window placed in front of the two room windows. In the first sub-figure, Jill is moving in a north-easterly direction, while Fred is moving eastwards.

2. In the second sub-figure, Jill is standing in the doorway, but `MotionEvent` shadows can be seen in both rooms, demonstrating how doors replicate motions into other spaces.

3. Finally, Jill is in the second room, showing doors' ability to transport virtual people between different co-ordinate spaces.




(a) Two simulated rooms (b) Jill walks through the doorway (c) Jill moves into room 2

Figure 15.2: A simulated building

## 15.6 Conclusion

An accurate building simulation can be achieved only by faithfully modelling all of the relevant objects in a building, rather than simulating only the people. By simulating objects such as furniture and cameras too, the simulation can be made far more realistic and more flexible than a simple simulation.

This realism is essential for developing a Smart Building that can operate with real cameras, as well as simulated sensors.

# Chapter 16

# Conclusion

Careful design of the simulated building provides a model for the Smart Building that is easily applicable to the real world. The complex building simulator parallels an actual office environment, including simulated objects such as people and doors. The simulation is simple to run on a single computer, the rooms have been made independent of each other, and objects within the rooms can interact to produce complex behaviour.

In the following part, we use the simulation as a source of information for the Smart Building.

# Part IV

# The Smart Building

# Chapter 17

# Introduction

A Smart Building is a stringent test of the power of distributed pattern analysis. Part IV of this dissertation demonstrates the strength of the architecture which was developed in part II in a real application — a computerised Smart Building.

Chapter 18 introduces the concept of a Smart Building, and explains why a Smart Building is a good test for a distributed object system. Chapter 19 shows how a Smart Building can be implemented using the distributed object architecture of part II. Chapter 20 proposes applications and enhancements for future work on this architecture, for commercial adaptation.

Chapter 21 concludes this dissertation, with a summary of the costs and benefits of using a distributed object architecture for distributed pattern analysis.

# Chapter 18

# Definition of a Smart Building

A 'Smart Building' is a computerised building which is aware of what is happening inside it. It can follow the movements of the people in the building as they walk from room to room. It can also use this information to assist the inhabitants, for example:

1. On the factory floor, it could sound a warning if someone was in danger from moving machinery.
2. In the office, it could automatically turn on the lights as someone entered a room, and it could forward telephone calls to the phone nearest to a person.
3. In a hospital, it could monitor a ward full of babies, and alert the hospital staff to distressed behaviour. In addition, it could follow the movements of the hospital staff, and locate them immediately whenever they were needed.
4. The house of a disabled or infirm person could act as a 'virtual butler' [38] to enable the person to live independently, without full-time nursing support.

A Smart Building constructs its model of reality using the sensors at its disposal, such as cameras and microphones. This model may include physical information, such as the locations and movements of the people in each room. It might also include other information, such as the expected intentions of a person. Many objects in this model would correspond directly to objects in the real world; this would enable the Smart Building to make predictions or deductions about what is happening inside it.

Finally, a Smart Building provides feedback, either directly or indirectly, to its inhabitants. Direct feedback could include controlling lights, or warning a factory worker of

danger. An example of indirect feedback would be informing a nurse in the hospital that a particular baby was crying, or automatically identifying which babies were crying excessively.

The rooms of a Smart Building are largely independent of each other, since movement in one room seldom has any effect on the objects in other rooms. This makes it ideal to implement a Smart Building using a distributed object system, with one computer for each room. The information that must traverse rooms can then be encapsulated into totally distributed objects, while the bulk of the pattern analysis can be performed locally within each room.

The objects on each room's computer would usually correspond to the objects present in the real room. Thus the computer model can almost exactly parallel the real world. Since objects on the same computer can interact more easily than objects on widely separated computers, this model accurately duplicates the relationships between the real objects. Similarly, when a real person walks from one room to another in the building, the object representing that person will also migrate from the one computer to the other.

# Chapter 19

# Implementation

A Smart Building illustrates the usefulness of distributed pattern recognition. Since most activity within a particular room of the building does not affect the other rooms, each room can have its own computer performing most of its data analysis. The rooms then need to communicate with each other only when a person moves between rooms. This can all be implemented easily using a distributed object architecture.

## 19.1 Overview

As people move within a building, they cause cameras to detect motion. This chapter describes Smart Building software which uses only these movements to reconstruct the behaviour of people within the building. As people move from room to room, the software uses distributed computing techniques to follow their movements, and share this information between the computers affected.

The Smart Building performs its analysis by stages, as follows:

1. When a camera detects movement, it creates a `Movement` RemoteObject on the FileServer associated with the camera's room.
2. A `PathController` object consumes these `Movements`, and attempts to string them together into `Path` objects, each consisting of a sequence of movements.
   (a) A collection of independent `Path` objects is known as a `Scenario`. Within the paths of a particular `Scenario`, each `Movement` object occurs exactly once.

(b) The `PathController` maintains a list of the most likely `Scenarios`, and when a new movement occurs, it adds it to each `Scenario` in turn, to determine a new set of most likely `Scenarios`. The most likely `Scenario` is the one which best represents the detected `Movements`, and satisfies the constraints of a `Path`.

(c) When a `Path` in the best reconstruction has been unchanged for a certain length of time, it is declared complete, and will not be changed again. At this point, the `PathController` registers it with the local FileServer, which makes the contents of the `Path` public information.

3. A `PathMatcher` consumes `Path` objects from each local FileServer, and pairs together those `Paths` which begin and end near to each other. It can pair `Paths` from different rooms, using extra information about the relative locations of rooms.

   (a) There is one `PathMatcher` object on each FileServer associated with a room. However, all of these objects are actually representatives of a single truly distributed object, spread across the entire building. These `PathMatchers` use an internal communication system to coordinate their information [31].

   (b) When a `PathMatcher` decides that two paths match, it creates a `SimpleMatch` RemoteObject on the local FileServer of the one path, and a duplicate of it on the FileServer associated with the other path.

4. A viewer class displays the activities of the `PathController` and `PathMatcher` in a window.

## 19.2 Applications of Distributed Objects

Distributed objects are used at various stages of the Smart Building reconstruction process. First, `Movement` objects are simple, immutable distributed objects. They are assembled on one computer — the computer which runs the simulated building — then they are created as distributed objects on the computer associated with their room of origin. Since `Movements` are RemoteObjects, they can be accessed remotely, from another computer, if their details are needed.

Next, the `PathController` uses FileServer services to consume these `Movements`, and produce `Paths` from them.

`PathMatchers` are far more sophisticated distributed objects. In this implementation, each `PathMatcher` has information about the adjacent rooms and their relative positions.

When a `Path` is discovered which begins near a room boundary, the `PathMatchers` of the two rooms communicate, to discover whether a person has moved from one room to the next. In the same way, a `PathMatcher` communicates with itself to discover correspondences between movements within a room. Thus the `PathMatcher` provides a conceptually consistent interface for finding and pairing adjacent paths, whether they are both in the same room, or in separate rooms. Whenever a path association is discovered, the `PathMatcher` creates a new `SimpleMatch` RemoteObject, to provide other objects with the new information.

This modular design makes its intermediate results public, using FileServers. This has a number of advantages:

1. The phases of this implementation are essentially independent of each other; they communicate their information indirectly, by creating objects on the local FileServer. As a result, one portion can be modified or rewritten entirely, without changing the other parts of the system.

2. Intermediate results are available for other processes to use. For example, the reconstruction viewer listens to the creation of new files, and draws its reconstruction based on the objects created on the local FileServer.

3. Extra data consumers can be added dynamically. Furthermore, different reconstruction methods can be used simultaneously with the same data, for comparison purposes.

4. Separation into modules simplifies the overall design, by using a number of small independent modules, rather than a single huge processing scheme.

Without distributed objects, this simple and elegant communication between computers would be impossible to achieve. Similarly, file creation events make it possible to completely separate producers and consumers of information. In a single-computer system, this is not an issue, since the ultimate consumer can be responsible for setting up all of the information producers. However, in a distributed system, it is often impractical to reboot computers and start processes in a predefined sequence — it is far more convenient to be able to start processes independently, in any order.

## 19.3 Flow of Information

This section outlines the flow of information within the Smart Building, from its source at the sensors, all the way to the objects representing people in the building and their movements.

### 19.3.1 Movement Objects

When a camera detects movement, it creates a `Movement` object on one of the FileServers which constitute the Smart Building. This is how information enters the building, and it is at this point that it becomes available for reconstruction purposes.

However, the simulated cameras are designed with imperfections; they occasionally fail to register a movement which did occur, and they sometimes register spurious movements when nothing actually happened. This makes the task of reconstructing behaviour from movements more difficult, and more realistic.

Each `Movement` has accessor methods to identify its location and its position. The location of a movement identifies the coordinate space in which the movement occurred, while the position indicates the bounds of the motion, within the coordinate space. A measure of the 'distance' between two movements can also be obtained. This enables movements from different coordinate spaces to be compared to each other, in later phases of the analysis. In fact, this is the only essential feature of the `Movement` interface: an ability to measure the distance between any two movements. Thus movements form a metric space [39].[1]

The assumption of independent coordinate spaces is similar to that of section 15.3.2, in the simulated building. It enables cameras to perform their computations independently of their positions within the building. There is also an implicit assumption that the preprocessing for extracting regions of movement is completed before the movement is injected into the building, and that this preprocessing does not rely on any other objects in the Smart Building. (In other words, the preprocessing can take place independently of extra information, but extra information may be used if it is available.)

In summary, `Movement` objects represent brief motions within the Smart Building, and also the distances between them.

[1] This distance measure should satisfy the requirements of a metric space — classes which implement `Movement` agree to satisfy this condition.

### 19.3.2 Paths and Scenarios

There are two interfaces which are widely used at various levels in the simulated building: the `Path` and `Scenario` interfaces. `Paths` are sequences of movements, while `Scenarios` are collections of `Paths`.

#### The `Path` Interface

A `Path` represents a sequence of events within the Smart Building. It also provides a measure of the 'value' of the `Path` — this measures the relative likelihood that this sequence of events results from a single source (such as a person walking), and not from random coincidence of events.

For example, a sequence of `Movement` events, all very near to each other, occurring within a limited time frame, would have a high value, since this could easily represent a person working in a room. In contrast, `Movements` jumping rapidly back and forth around the room should have a low value; they might be noise, or they might be more valuable as two separate `Paths`.

The Smart Building attempts to assemble events into `Paths` with the highest total value possible.

#### The `Scenario` Interface

A `Scenario` is a collection of independent `Paths`. No event is repeated between the `Paths` of a `Scenario`, thus a `Scenario` represents a group of `Paths` that might have occurred in parallel with each other. Like `Paths`, `Scenarios` also have a value function associated with them; `Scenarios` with higher values should represent more ordered scenarios than those of low value.

The Smart Building uses `Scenarios` in order to decide how best to allocate events to `Paths`. `Scenarios` guarantee that the Smart Building will not make impossible reconstructions, since there is no overlap of events within a `Scenario`. By maintaining a number of `Scenarios`, the Smart Building can consider many possible reconstructions simultaneously, before it decides to commit to a particular view of what happened in the building.

New events can be added to any `Path` within the `Scenario`, and the value of the `Scenario` which would be produced can be computed for comparison purposes, in order to decide

with which `Path` each new event should be associated.

Events and `Paths` can also be removed from a `Scenario`; for example, when it has been decided that a certain `Path` definitely occurred, that Path should be removed from all `Scenarios` (since it is no longer a basis for uncertainty). This may significantly change the values of other `Paths` in other `Scenarios`, since they may have been using the events of the `Path` to be removed, as part of smaller, independent Paths. This ensures that all of the `Scenarios` being considered consist of the same set of events.

### 19.3.3 Controller Objects

The Smart Building tries to combine elementary pieces of information, such as `Movements`, into more complex information, such as `Paths`. In order to do this, it must be aware of the creation of new motion events, and decide how they would best be combined into `Paths`. `PathController` and `PathMatcher` objects do just this.

#### `PathController`

There is a `PathController` object located on each FileServer (or 'Place') where there is a camera. This controller listens for the creation of new `Movement` objects in the system. In a simple case, the controller collects only those events which are completely contained within a certain space.

The `PathController` typically maintains a number of `Scenarios`, reflecting likely combinations of `Movement` into `Paths`. Whenever a new movement occurs, each `Path` in each `Scenario` is tested with the movement, to see where it should be allocated. (The new movement may otherwise be best as the start of a new `Path`.)

When the controller decides that a certain `Path` is complete, it removes the `Path` from all of its `Scenarios`, and registers that `Path` on the local FileServer. This makes the `Path` available for other objects to view and use.

#### `PathMatcher`

A `PathMatcher` maintains a distributed network of `Paths`, and attempts to coordinate and combine them to form longer Paths. These links between paths are represented using `SimpleMatch` objects. The `PathMatcher` is a truly distributed RemoteObject,

with a representative on every FileServer where there is a camera. The `PathMatcher` becomes aware of new Paths by listening to each FileServer, for new `Path` registrations.

Just as `Movements` are the source objects for the `PathController`, `Paths` are the sources objects for the `PathMatcher`.

In a simple system, these two controllers are independent stages in the flow of information through the Smart Building. This implies a tradeoff between the quality of the reconstructed view of the building, and the lag in producing it — the `PathMatcher` cannot begin its work until it receives information from the `PathController`.

A more complicated design would require these controllers to be interconnected, granting the `PathMatcher` access to the incomplete analyses of the `PathController`. The advantage of this is that reconstructions are available immediately an event occurs, making the reconstruction process very responsive. However, the costs are that this approach requires more computational resources than the simple system, it requires more binding between `PathController` and `PathMatcher`, and the intermediate results are subject to change.

### 19.3.4 Summary

Information enters the Smart Building from cameras as `Movement` objects. The `PathController` combines `Movements` within each space into `Paths`, using `Scenarios` to decide which paths are best. The `PathMatcher` then produces `SimpleMatches` connecting `Paths`, which can span different rooms.

Figure 19.1 shows a conceptual UML diagram of the correlations between these classes.

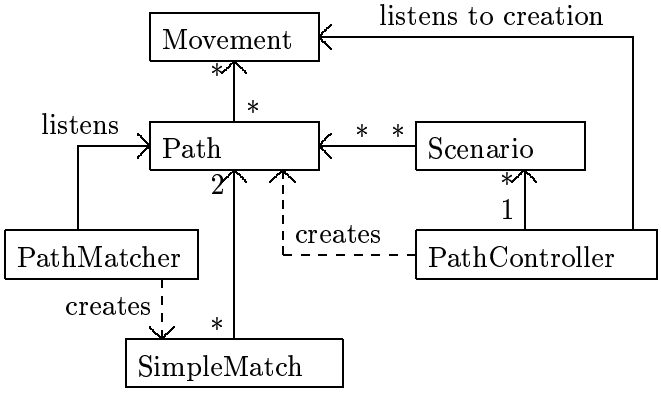



Figure 19.1: Conceptual class diagram for the Smart Building

## 19.4 Segmenting Movements into Paths

The task of the Smart Building described in this section is to reconstruct the movements of people from a sequence of motion detections. With complete information, each movement can simply be associated with the nearest movement from the previous time instant. However, this will not work if the information is incomplete (the camera is not perfect), or if people are entering or leaving the room, or stop moving for a short while.

This section describes how the suitability of a reconstruction can be quantified, to produce sensible reconstructions, even with inaccurate information. Ideally, the value of the reconstruction would be a true Bayesian probability measure. It would be the absolute probability that this is the correct reconstruction, given the measured movement data. This could be computed if the probabilistic mapping from movement paths to motion detections were known, and the absolute probabilities of movement detections and path configurations were also available.

Simpler measures of path values are sufficient to produce good path reconstructions in the Smart Building implemented here. Nevertheless, there are certain issues which must be considered in producing these reconstructions.

### 19.4.1 The Problem of Singletons

Singletons are paths containing just one motion detection. Although certain actions in the Smart Building will produce only a single motion detection, this occurs only occasionally. More often, a singleton will result from a spurious motion detection (false positive), or from a movement being allocated to a new path, when it should have been appended to an existing path.

When new paths begin, however, they must also start as singletons. Furthermore, when a movement occurs, it is impossible to decide whether it would ultimately remain a singleton, if it were to be made into a new path, or whether other movements would be associated with the path later. The difficulty is the lack of future information.

A distinction must be made between complete and incomplete paths. Complete paths do not change, while data may still be added to incomplete paths. If a complete path and an incomplete path contain the same movements, they may still have different probabilities.

For example, a complete singleton path shows that just one movement occurred, and was not followed by any other movements. An incomplete singleton path shows that a single

movement has occurred, but that there is no information about subsequent movements. In other words, there is a significant difference between knowing that a movement has not occurred, and not knowing anything about that movement. This information must be used in determining the likelihood of various scenarios of allocating movements to paths.

Another way of seeing this is to observe that paths contain not only events, but also non-events: if events are not contained in a path, this is significant information.

### 19.4.2 Evaluating a Scenario

The `PathController` tries to determine the best possible reconstruction scenario, which describes the movements within the building. In order to do this, it needs a way of valuing a scenario, to compare it with other scenarios.

The information available to the `PathController` consists of the movements detected by the cameras in a room ($M$), and the current set of scenarios ($R_1$, $R_2$, ... $R_n$). The `PathController` can measure the suitability of a reconstructed scenario $R_i$ by calculating the probability that it is the best of the available scenarios, namely $P(R_j|M, \text{Reconstruction is } R_1 \text{ or } R_2 \text{ or } \ldots\ R_n)$. Now, using Bayes' formula,

$$\begin{aligned} P(R_i|M, R_1\ or \ldots R_n) &= \frac{P(M, R_1\ or \ldots R_n|R_i) \times P(R_i)}{\sum_{k=1}^{n} P(M, R_1\ or \ldots R_n|R_k) \times P(R_k)} \\ &= \frac{P(M|R_i) \times P(R_i)}{\sum_k P(M|R_k) \times P(R_k)} \end{aligned}$$

The probability $P(M|R_i)$ represents the probability that the movements $M$ would have resulted from the collections of paths $R_i$. $P(R_i)$ represents the absolute probability of scenario $R_i$ arising — in other words, it represents the prior probability of the particular paths contained in $R_i$ occurring. This is an infinitesimal number, in absolute terms, since there are so many scenarios that could possibly occur. However, the $P(R_i)$ terms can be replaced with relative probabilities, since the scaling factor between absolute and relative probability cancels out in the formula above.

For a given collection of scenarios $R_1$, $R_2$, ... $R_n$, the denominator term in the equation above does not change, so it can be ignored for the purposes of ranking scenarios according to their applicability. Furthermore, if the paths of actions that caused the movements within the building were independent of each other, then the probability of

the reconstruction $P(R_i)$ would be the product of the probabilities of the Paths. In addition, if each of the movements in $M$ is represented exactly once in each scenario $R_i$, then the term $P(M|R_i)$ can be disregarded for a first approximation, since all reconstructions will then fit the data reasonably well.

That is how the `SimpleScenario` class operates; the scenario computes the product of the values of its individual paths, and presents that as its total value.

### 19.4.3 A Simple Example of Probability Calculation

The following simple example illustrates how probabilities can be calculated in a real situation, to decide on possible reconstructions. In this example, movement events are received as the result of movements within a building. The following assumptions are made about the detected movements, and the paths which produced them:

1. It is assumed that the movement events are time-stamped when they are received,
2. Reconstructed paths must consist of at least two movements,
3. The movements in a path are consecutive in time — one per time unit,
4. The movements contained in a path are detected perfectly, and
5. There are no spurious movements.

#### Determining the Validity of Paths

It is convenient to be able to perform reconstructions iteratively, successively improving the reconstructions over time as more data is received. However, this requires us to be able to consider partially reconstructed paths, as well as complete ones, in our analysis. To this end, the following requirements determine when a path is valid and when it is invalid.

1. For a valid complete path (one to which no further movements will be added)
   (a) All movements must be consecutive, i.e. every time slot must be occupied
   (b) The path length must be at least 2
2. For a valid incomplete path (to which more movements may be added)

(a) All movements must be consecutive

(b) The latest movement in the path must be time stamped with the current time

(c) There is no length limitation, since movements could still be added to an incomplete path, to make a complete one

These descriptions can be combined into a single compound description; any path is valid if and only if:

1. All movements are consecutive
2. Either (path length $> 1$) or (path time stamp = last movement's time stamp)

For notational convenience, a path may be described by a list of $\mu$ time-stamped movements $\mathrm{m}_1\mathrm{m}_2 \ldots \mathrm{m}_\mu$, and a time-stamp $\tau$ for the path, and written as Path($\mathrm{m}_1\mathrm{m}_2 \ldots \mathrm{m}_\mu$, t=$\tau$). The time-stamp of a movement $\mathrm{m}_k$ is written as $\mathrm{t}(\mathrm{m}_k)$. In symbols, Path($\mathrm{m}_1\mathrm{m}_2 \ldots \mathrm{m}_\mu$, t=$\tau$) is valid iff :

1. $\mathrm{t}(\mathrm{m}_{k+1}) = \mathrm{t}(\mathrm{m}_k) + 1$ for all $k$
2. $(\mu > 1) \vee (\mathrm{t}(\mathrm{m}_\mu) = \tau)$

The probability that a path $p$ is valid can then be written as $\mathrm{P}(p)$, with $\mathrm{P}(p) = 1$ if the path is valid and $\mathrm{P}(p) = 0$ if the path is invalid.

A scenario is a list of paths. A scenario is valid iff all of its constituent paths are valid. If scenario $s$ consists of paths $p_1, p_2, \ldots p_n$ then

$$\mathrm{P}(s) = \mathrm{P}(p_1, p_2, \ldots p_n) = \prod_{k=1}^{n} p_k$$

This agrees with the probability computation at the end of section 19.4.2.

### A Sample Reconstruction

Using the assumptions above, possible scenarios can be reconstructed from the following sample data, in which four movements are received over five time periods: movement a is

received when time t=1, b when t=2, c at t=3, and d at t=4. When t=5, no movement is detected.

At each time interval, the available information is the current time, the set of motions detected thus far, and the possible scenarios produced during the previous period. By combining the new events with the old scenarios in every way possible, new scenarios are produced, and tested for validity.

- When t=0:
  1. No movements have yet been detected
  2. The set of potential scenarios is empty.
- When t=1:
  1. Detected movements = {a}
  2. Potential scenarios:
     - $p_1$=Path(a, t=1) — which is possible, by the assumptions above
- When t=2:
  1. Detected movements = {a,b}
  2. Potential scenarios:
     - $p_{2a1}$=Path(a, t=2) and $p_{2b1}$=Path(b, t=2) — impossible, since P($p_{2a1}$) = 0 so P($p_{2a1}, p_{2b1}$) = 0. Since this scenario is impossible, it is ignored in the next time interval.
     - $p_{2b}$=Path(ab, t=2) — possible, since P($p_{2b}$) = 1
- When t=3:
  1. Detected movements = {a,b,c}
  2. Potential scenarios:
     - Path(ab, t=3) + Path(c, t=3) — possible
     - Path(abc, t=3) — possible
- When t=4:
  1. Detected movements = {a,b,c,d}
  2. Potential scenarios:

  - Path(ab, t=4) + Path(c, t=4) + Path(d, t=4) — impossible
  - Path(abd, t=4) + Path(c, t=4) — impossible
  - Path(ab, t=4) + Path(cd, t=4) — possible
  - Path(abc, t=4) + Path(d, t=4) — possible
  - Path(abcd, t=4) — possible

- When t=5:

  1. Detected movements = {a,b,c,d}
  2. Potential scenarios:
     - Path(ab, t=5) + Path(cd, t=5) — possible
     - Path(abc, t=5) + Path(d, t=5) — impossible
     - Path(abcd, t=5) — possible

In conclusion, there are therefore two possible scenarios which are consistent with the detected events. Either there was one long path containing all four movements (abcd), or there were two short paths consisting of two movements each (ab and cd).

The simple model above cannot determine which of the possible scenarios is more likely to be correct. However, it does demonstrate how a probabilistic approach can be used to decide how to combine events into sensible reconstructions.

## 19.5 Implementation Details

The concepts developed above were used to implement a Smart Building, which received its input data from the simulated building of part III. The `Movement` and `Path` interfaces were implemented by classes `SimpleMovement` and `SimplePath` in package `db.smartroom.building.simple`. This package also includes a `SimplePathController` and a `SimplePathMatcher`, which together correlate movements into paths, and match paths with each other.

### 19.5.1 Values of `SimplePaths`

This section describes the calculation which is used to determine the value of `SimplePath` objects in the Smart Building. A `SimplePath` represents a sequence of movements, made by one person, within a single space, such as a room.

The `PathController` uses path values to decide which reconstructions are best. High values represent good reconstructions, while low values represent bad ones. For `SimplePaths`, the following assumptions are made:

1. There is a maximum displacement between two consecutive movements. This is because there is a real maximum speed at which people can move through a building. For this illustration, that maximum speed is assumed to be 2m/s, a brisk walking speed.

2. There is a maximum time delay between consecutive movements on a path. After this time delay has elapsed, the path is considered complete, and adding new movements to the path will give a new path of probability 0. Here, the maximum delay is 4 detection periods. This is designed to avoid losing track of a path because of false negative camera detections. When a person stops moving and then starts again, that should produce two separate paths.

3. Completed singleton paths are assigned a value of 0.001. They usually result from false positive camera detections.

4. Other paths are assigned a value equal to the reciprocal of the average step-wise speed of movement, or 100, whichever is less. 2m/s average net speed gives a value of 0.5s/m; half that speed gives a value of 1.

### 19.5.2 Implementing a `PathMatcher`

Class `SimplePathMatcher` keeps track of the correlations between Paths, whether they are on the same computer, or on separate computers.

#### Detection

When the `SimplePathMatcher` is first created and registered with a FileServer, it duplicates itself to all other FileServers in the Smart Building. Then, whenever a new `Path` object is created anywhere in the building, the local `SimplePathMatcher` representative decides which rooms are near to the start of the path, and informs their matchers of the path.

#### Correlation

When a `SimplePathMatcher` is informed that a path started near its space (using its `communicate` method), it compares the endpoints of the paths which originated in its space to the starting point of the new path. If the points are close enough to each other, in space and time, then the matcher creates a `SimpleMatch` object on both its local FileServer and the new path's FileServer, to inform other objects that the two paths are linked.

The `SimplePathMatcher` also uses the method above to match two paths which are in the same space. This shows how distributed objects can be used for communication across computers, and within the same computer, easily and flexibly.

## 19.6 A Sample Run

The Smart Building's operation is demonstrated in this section, using a screen capture from a sample run. In this demonstration, three computers cooperate to simulate and reconstruct actions in a Smart Building.

The demonstration creates a simulated building consisting of two rooms, like the simulation of section 15.5. The two inhabitants, Jill and Fred, have preprogrammed paths of action, although their actions can also be controlled directly by a person, by clicking on them using a mouse. Fred begins in room 1, moves east for 10 time units, south for 20, then stops. Jill also begins in room 1, but moves north-north-east for 30 units, through the doorway into the second room. There, she walks east for 10 units and pauses for 10 clock ticks. Finally, she moves south and south-west for 30 units, back into room 1, before stopping.

Figure 19.2, on page 130, shows a screen snapshot of the simulated building and snapshots of the reconstructions of the two rooms. The snapshots were taken just before the end of the activities described above. More screen shots of the Smart Building are included in Appendix B.

#### Simulated Building

The simulation snapshot shows both Fred and Jill in room 1 (the lower room); section 15.5 contains a detailed explanation of the simulation screen display.

### Motion Reconstruction

The reconstruction windows show complete, reconstructed paths as light lines, and incomplete, tentative paths as dark lines. Furthermore, when two paths have been matched with each other, the end of the first path is shown as a light circle, and the start of the second path as a dark circle.

The reconstruction of room 1 shows that Fred made one continuous movement, walking east first and then south. The reconstructed path is jagged because the simulated camera is designed to introduce noise into its detected movements, and also occasionally to fail to detect movements.

The reconstruction also shows that Jill walked north-east until she left room 1, and that she re-entered the room walking south-south-east.

Furthermore, there is a spot near the middle of the reconstruction, representing a false positive detection introduced by the cameras. Correctly, this was not associated with either Fred or Jill's paths.

Room 2's reconstruction shows that Jill walked north and east. She then paused (the path terminates) before moving south.

### Path Matching

There is a light circle in room 1's reconstruction where Jill left the room, and a dark circle where she entered room 2. This shows that the two separate paths have been associated with each other correctly, by the `SimplePathMatcher`. There are also two circle where Jill paused in room 2, where two paths were again joined.

There is no connection between the final path of room 2 and the tentative path of room 1. This is because the tentative path has not yet been registered with the local FileServer, and it may still be removed if it is superseded by better reconstruction of the activity in the building.

### Running the Demonstration

This section gives instructions for running the demonstration on three different computers:

1. Set up two room computers, for reconstructions: on computer XYZ, run `rmiregistry`. Then run the batch file '`MakeRoomServer.bat XYZ`' in the `db/smartroom/building/debug` directory. On a second computer, again run '`MakeRoomServer.bat XYZ`'.

2. Run the building simulator: on a third computer, run '`TestSmallSimpleBuilding.bat XYZ`'. This will create a simulated building on this computer, and use the other two computers to perform reconstructions.

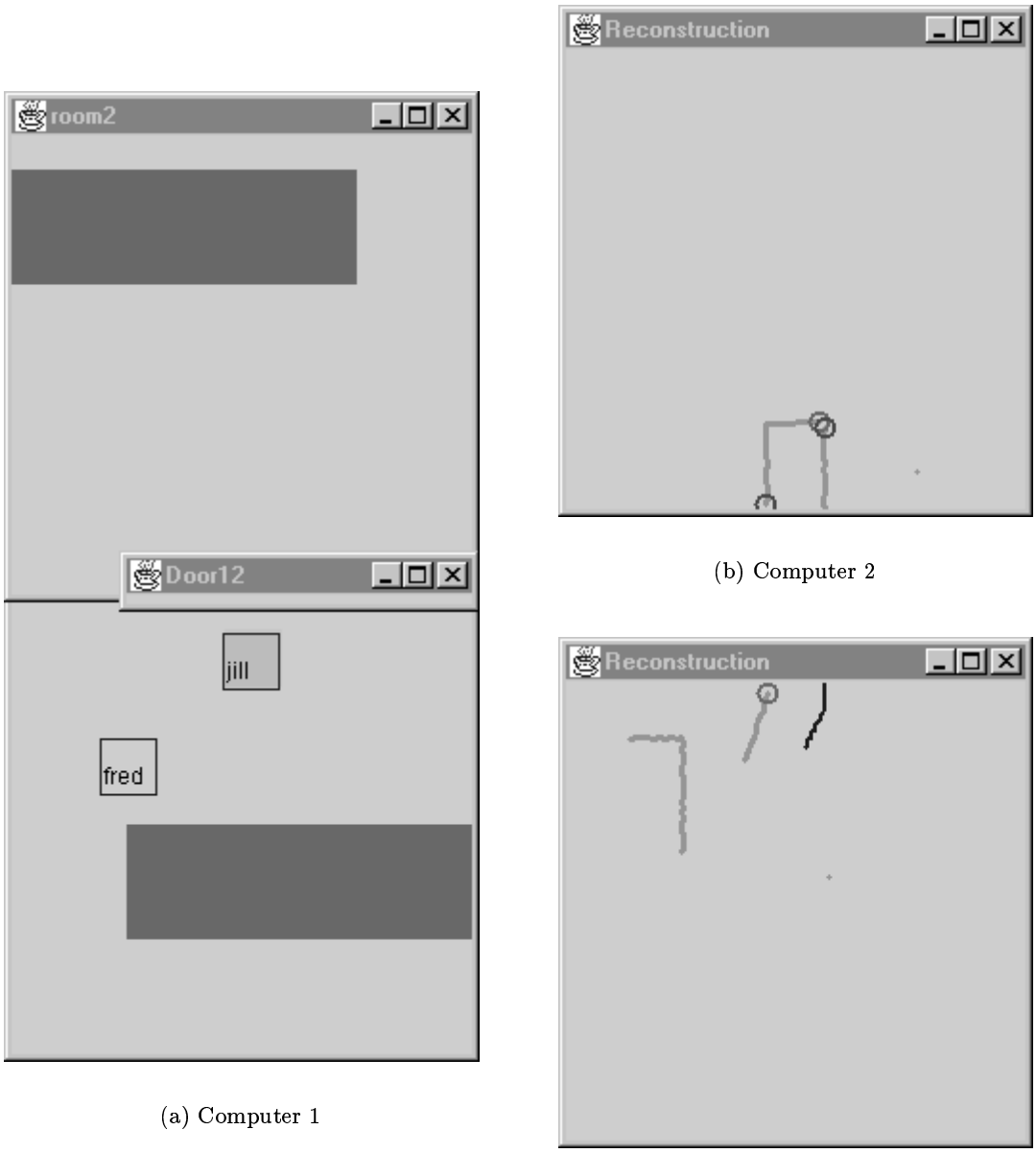
It is also possible to run the demonstration on one computer, using three Java Virtual Machines. Since the JVM's are separate, they are independent, and their only communication is through standard networking protocols — from their perspective, they might as well all be on different computers. This can be done in the same way as above, or by running the batch file '`TestSmall.bat`'

## 19.7 Conclusion

A Smart Building is an environment which inevitably spans many computers. This chapter has shown that a distributed object system can actually exploit the separation of computers in a Smart Building, to reconstruct movements in the building.

The distributed object system explicitly uses the spatial localisation of movements within each room, to perform local reconstructions first. Communication between computers is then needed only when paths are found to end close to doors to other rooms. Truly distributed objects simplify the design of distributed systems, by moving communication and distribution considerations into the objects responsible for the communication.

The Smart Building demonstrates how reconstruction tasks can be shared between computers, eliminating the need for a central, controlling computer.




(a) Computer 1

(b) Computer 2

(c) Computer 3

Figure 19.2: The Smart Building simulation and reconstructions

# Chapter 20

# Possible Enhancements and Further Applications

This chapter describes additional enhancements which could be made to the distributed object system and the Smart Building, to make it suitable for commercial use.

## 20.1 The Distributed Object System

The distributed object system could be extended to enable inactive distributed objects to be stored on disk. This would enable the distributed object system to be shut down and restarted without losing its inventory of objects. It would also allow the system to operate for months at a time, without exhausting the computers' memory resources, as long as there is enough disk space available.

To do this, a mechanism would be needed to flag inactive distributed objects. In Java, the automatic garbage collection system could be used to do this, by ensuring that FileServers contain only weak references to their RemoteObjects, and that other distributed objects hold only `RemoteReferences`, not direct references to local RemoteObjects.

Furthermore, widely distributed RemoteObjects could be designed with a protocol for deleting inactive representatives, after centralising their data. This would reduce the storage requirements for inactive objects, and also facilitate reactivation of these objects, since each inactive object would be stored only at its home location.

Distributed objects could also be upgraded to take into account network failures, by

delaying the communication of information until network connections are re-established.

Finally, the cost of communication between distributed objects could be modelled explicitly [18], to optimise the use of the available communication bandwidth. Such a system would possibly allow RemoteObjects to specify Quality of Service requirements, such as those used in Asynchronous Transfer Mode (ATM) networks [40].

## 20.2 The Building Simulation

The building simulation currently does not allow simulated people to walk into each other — they are modelled as rectangular, moving obstructions which cannot overlap. This model could be relaxed to allow two people to move closer together, by marking only a limited central area of each person's region of movement as actually occupied. The behaviour of simulated people could also be improved, to allow goal-directed behaviour instead of pre-specified or user controlled movements.

The camera model would then need to be redesigned, to produce a single movement event when two virtual people overlap. In addition, three-dimensional objects could also be extended and abstracted, from simple rectangular bounding boxes to arbitrary solids, to better model people and other objects.

## 20.3 The Smart Building

Path reconstructions could be made more realistic, by using Bayesian probabilities to accurately model the behaviour of real cameras, and using this information to determine path probabilities rigorously. If people can overlap in front of a camera, then provision must also be made to reconstruct overlapping movement paths — now a single movement can be allocated to two paths.

Paths in the Smart Building could also be associated with real people, by integrating a face detection system and a distributed database of known faces. This could also be achieved by observing other characteristics of the people in the building, such as height.

Finally, the Smart Building software could be used to model a real building, by replacing the building simulation with real cameras.

# Chapter 21

# Conclusions

Distributed systems enable a group of computers to work together towards a common goal. This dissertation has shown the development of a distributed object architecture, which enables a network of smaller computers to perform the task of a central computer at a far lower cost.

This architecture is most useful in situations where data can be processed near to its source. The system can then take advantage of the geographical proximity of interrelated data sources, allowing them to inform each other of their results. Many real-time control systems belong to this category, and they can also benefit from the other advantages of distributed systems: greatly enhanced fault tolerance, and a reduction in lag between the production and processing of data, especially in very large networks.

The object-oriented design of the architecture allows both active agents and passive data to interact as peers in the system. Since each distributed object is able to define its own migration strategy, novel distribution techniques are made possible. Furthermore, the technique of 'truly distributed objects' enables a group of distributed objects on different computers to act together, as if they were a single object. This allows a single task to be shared by many computers elegantly.

A 'Smart Building' is a rigorous test of the benefits of distributed pattern recognition. This computerised building has one computer in each room, and it uses the distributed object architecture to enable them to cooperate in interpreting the behaviour of people in the building.

This shows that distributed pattern analysis is both powerful and convenient, when implemented using a distributed object system.

# Bibliography


[1] J. P. Morrill, "Distributed recognition of patterns in time series data," *Communications of the ACM*, vol. 41, pp. 45–51, May 1998.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] T. Cornell, "Coursenotes: Introduction to prolog (prolog-einführung)." Postscript file, email `cornell@sfs.nphil.uni-tuebingen.de`, Mar. 9, 1998.

[4] J. Wielemaker, "Swi-prolog 2.9.9." `ftp://swi.psy.uva.nl/pub/SWI-Prolog/`, Mar. 1998.

[5] K.-M. Lam and H. Yan, "An analytic-to-holistic approach for face recognition based on a single frontal view," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 673–686, July 1998.

[6] M. Bichsel, "Analyzing a scene's picture set under varying lighting," *Computer Vision and Image Understanding*, vol. 71, pp. 271–280, Sept. 1998.

[7] S. M. Weiss and C. A. Kulikowski, *Computer Systems That Learn*. San Francisco, California: Morgan Kaufmann Publishers, Inc., 1991.

[8] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*. New York: Wiley-Interscience, 1973.

[9] C. M. Bishop, *Neural networks for pattern recognition*. Oxford: Clarendon, 1995.

[10] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 23+, Jan. 1998.

[11] S. Haykin, *Neural Networks: a comprehensive foundation*. Prentice Hall, 1999.

[12] J. Koza, *Genetic Programming — On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[13] D. Dubois and H. Prade, *Possibility Theory: An Approach to Computerized Processing of Uncertainty.* Plenum Press, 1988.

[14] A. G. Skarmeta and H. M. Barberá, "Fuzzy logic based intelligent agents for reactive navigation in autonomous systems," in *Fifth International Conference on Fuzzy Theory and Technology (FT&T'97)*, (Durham, USA), pp. 168–171, Mar. 1997.

[15] R. E. Neapolitan, *Probabilistic Reasoning in Expert Systems.* John Wiley & Sons Inc., New York, 1990.

[16] G. Wagner, *Foundations of knowledge systems : with applications to databases and agents.* Kluwer Academic Publishers, 1998.

[17] J. Earman, *Bayes or bust : a critical examination of Bayesian confirmation theory.* Cambridge, Mass: MIT Press, 1992.

[18] J. Kim and D. J. Lilja, "Performance-based path determination for interprocessor communication in distributed computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 316–327, Mar. 1999.

[19] A. Silberschatz and P. B. Galvin, *Operating System Concepts.* Addison-Wesley Publishing Company, 1994.

[20] F. S. Wong and M. R. Ito, "Design and evaluation of the event-driven computer," *IEE Proceedings, Part E*, vol. 131, pp. 209–222, Nov. 1984.

[21] J. Yang and A. K. Mok, "Symbolic model checking for event-driven real-time systems," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 386–412, Mar. 1997.

[22] B. Eckel, *Thinking in Java.* Electronically distributed at `http://www.EckelObjects.com`, 1997.

[23] A. Dogac, C. Dengi, and M. T. Öszu, "Distributed object computing platforms," *Communications of the ACM*, vol. 41, pp. 95–103, Sept. 1998.

[24] Microsoft, "DCOM architecture: White paper." `http://www.microsoft.com/`, 1998.

[25] S. Purao, H. Jain, and D. Nazareth, "Effective distribution of object-oriented applications," *Communications of the ACM*, vol. 41, pp. 100–108, Aug. 1998.

[26] J. M. Hyde and M. R. Cutkosky, "A phase management framework for event-driven dextrous manipulation," *IEEE Transactions on Robotics and Automation*, vol. 14, pp. 978–985, Dec. 1998.

[27] A. Geist, A. Beguelin, *et al.*, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. Electronically distributed at `http://www.epm.ornl.gov/pvm/`.

[28] OMG (Object Management Group), "Mobile agent specification." Electronically distributed at `http://www.omg.org`, July 31, 1998.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1994.

[30] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer, "Mole - concepts of a mobile agent system." Universität Stuttgart Fakultät Informatik, Aug. 1997.

[31] E. A. Kendall, P. V. M. Krishna, C. V. Pathak, and C. B. Suresh, "Patterns of intelligent and mobile agents," in *AGENTS '98. Proceedings of the second international conference on Autonomous agents*, (Minneapolis, 1998), pp. 92–99, ACM Press, May 1998.

[32] T.-Y. Yen and W. Wolf, "Performance estimation for real-time distributed embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 1125–1136, Nov. 1998.

[33] M. Campione, K. Walrath, *et al.*, "The Java Tutorial: Security in JDK 1.2." `http://java.sun.com/docs/books/tutorial/security1.2/index.html`, July 1999.

[34] S. Srinivasan and N. K. Jha, "Safety and reliability driven task allocation in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 238–250, Mar. 1999.

[35] G. E. Krasner and S. T. Pope, "A cookbook for using the model view controller user interface paradigm in Smalltalk–80," *Journal of Object-Oriented Programming*, vol. 1, pp. 26–49, Aug. 1988.

[36] E. Gamma and K. Beck, "JUnit 2.1: Test infected: Programmers love writing tests." `ftp://www.armaties.com/TestingFramework/JUnit/`, 1998.

[37] A. D. Stoyen, T. J. Marlowe, M. F. Younis, and P. V. Petrov, "A development environment for complex distributed real-time applications," *IEEE Transactions on Software Engineering*, vol. 25, pp. 50–74, Jan. 1999.

[38] A. P. Pentland, "Smart rooms and Pfinder," *Scientific American*, vol. 274, pp. 68–72, Apr. 1996.

[39] J. R. Giles, *Introduction to the analysis of metric spaces.* Cambridge: Cambridge University Press, 1987.

[40] J.-F. Frigon, "Dynamic reservation tdma medium access control protocol for wireless atm networks." Masters thesis, University of British Columbia, 1998.

# Appendix A

# Programmer's Reference for the Distributed Object System

This appendix contains a programmer's reference for essential Java classes and interfaces, for the distributed object system developed in part II. Interface `FileServer`, interface `RemoteObject` and class `RemoteReference` are all contained in package `db.smartroom.server`.

## Interface `FileServer`

```
public interface FileServer
extends java.rmi.Remote, java.io.Serializable
```

A FileServer stores a list of RemoteObjects, which are local to the current Java Virtual Machine. FileServers allow objects to be created, accessed, listed and deleted, both remotely and locally. FileServers provide the 'places' where RemoteObjects can exist and interact.

See Also: `RemoteObject, FileServerLister`

### Method Summary

- `void addFileListener(FileListener l)`
  Adds a FileListener to the FileServer.

- `java.io.Serializable communicate(RemoteReference obj, java.io.Serializable data)`
  Send a message to a RemoteObject on this FileServer.

- `RemoteReference create(RemoteObject obj)`
  Create (and keep) a copy of an object on this FileServer.

- `java.util.Set listFiles()`
  Return a list of (RemoteReferences to) all of the files stored on this FileServer.

- `RemoteObject open(RemoteObject obj)`
  Return the FileServer's copy of the object obj, namely x satisfying x.equals(obj).

- `RemoteObject open(RemoteReference obj)`
  Return the FileServer's copy of an object with the same home as obj.

- `boolean remove(RemoteReference o)`
  Removes a file from the given FileServer (optional operation).

- `void removeFileListener(FileListener l)`
  Removes a FileListener from the FileServer.

## Interface `RemoteObject`

```
public interface RemoteObject
extends java.io.Serializable
```

The RemoteObject interface describes objects which can be can be created on FileServers, and specifies the legal interactions for those objects.

RemoteObjects should allow themselves to be completely serialized, until they are first Registered on a file server. Thereafter, they should never again allow themselves to be registered, except under the same name as before, and they should serialize alternative representations of themselves.

### Method Summary

- `java.io.Serializable communicate(java.io.Serializable info)`
  Enables messages to be sent to a particular object.

- `boolean equals(java.lang.Object obj)`
  RemoteObjects should correctly reimplement the equals() and hashCode() methods of the Object class, to correctly equate any two objects with a common 'home', and also equate a RemoteObject with its corresponding RemoteReference

- `RemoteReference getHome()`
  Discover object's home location int hashCode()

- `RemoteObject register(RemoteReference home)`
  Register is called whenever a RemoteObject is about to be stored on a FileServer.

## Class `RemoteReference`

```
public class RemoteReference
extends java.lang.Object implements java.io.Serializable
```

RemoteReferences are passive links to RemoteObjects. They enable an object to be located uniquely by name, from anywhere. They should seldom be used explicitly, except within the implementations of RemoteObjects.

A FileServer acts as the home base for the object, so that the object can always be located from its RemoteReference.

### Constructor Summary

- `RemoteReference(FileServer theServer, java.lang.String objName)`
  Constructs a new RemoteReference.

### Method Summary

- `java.io.Serializable communicate(java.io.Serializable data)`
  Sends a message to the original object, at its home location

- `boolean equals(java.lang.Object obj)`
  Two references are equal only if they refer to the same object on the same server.

- `FileServer getFileServer()`
  Returns the file server on which this object was originally created.

- `int hashCode()`
  The hash-table function is rewritten, too, so that two objects that agree with equals() will also agree with their hash table entries

- `RemoteObject open()`
  Opens the original object, from its home location

- `java.lang.String toString()`
  Returns a string representation of this RemoteReference.

# Appendix B

# Screen Captures from a Smart Building

This appendix illustrates typical screen displays produced by the Smart Building. These images were all generated with the program `TestSmallSimpleBuilding` in package `db.smartroom.building.debug`, used in section 19.6.

The left half of each diagram shows a plan view of the simulated building. The top right window shows a reconstruction of the upper room of the simulation, and the bottom right window shows the lower room's reconstruction.

Figures B.1 through B.5 show snapshots of the building at time t=0, t=15, t=35, t=55 and t=80 respectively.

In the simulation, Fred moves east from t=0 until t=10, south until t=30, and then stops moving.

Jill moves north-north-east from t=0 until t=30, east until t=40, pauses until t=50, then finally south and south-west until t=80, before stopping.

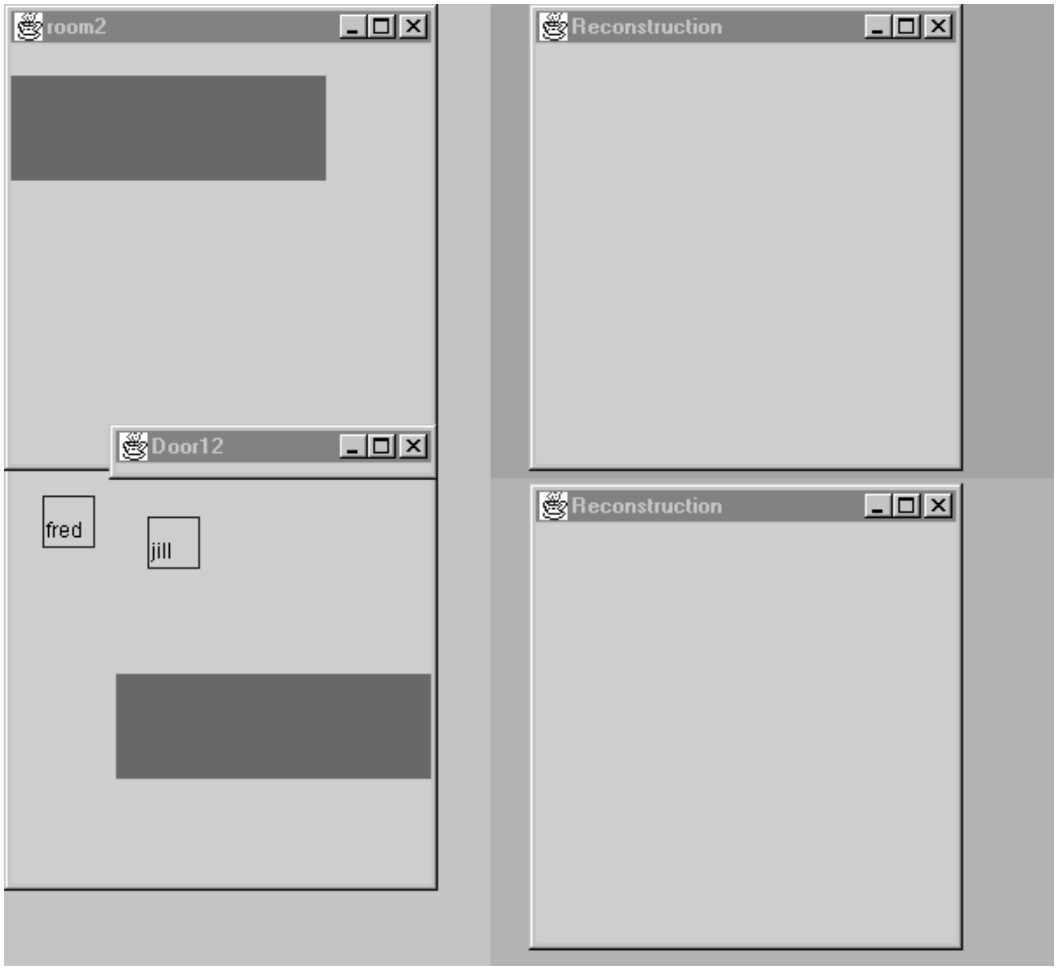



Figure B.1: The Smart Building at time t=0

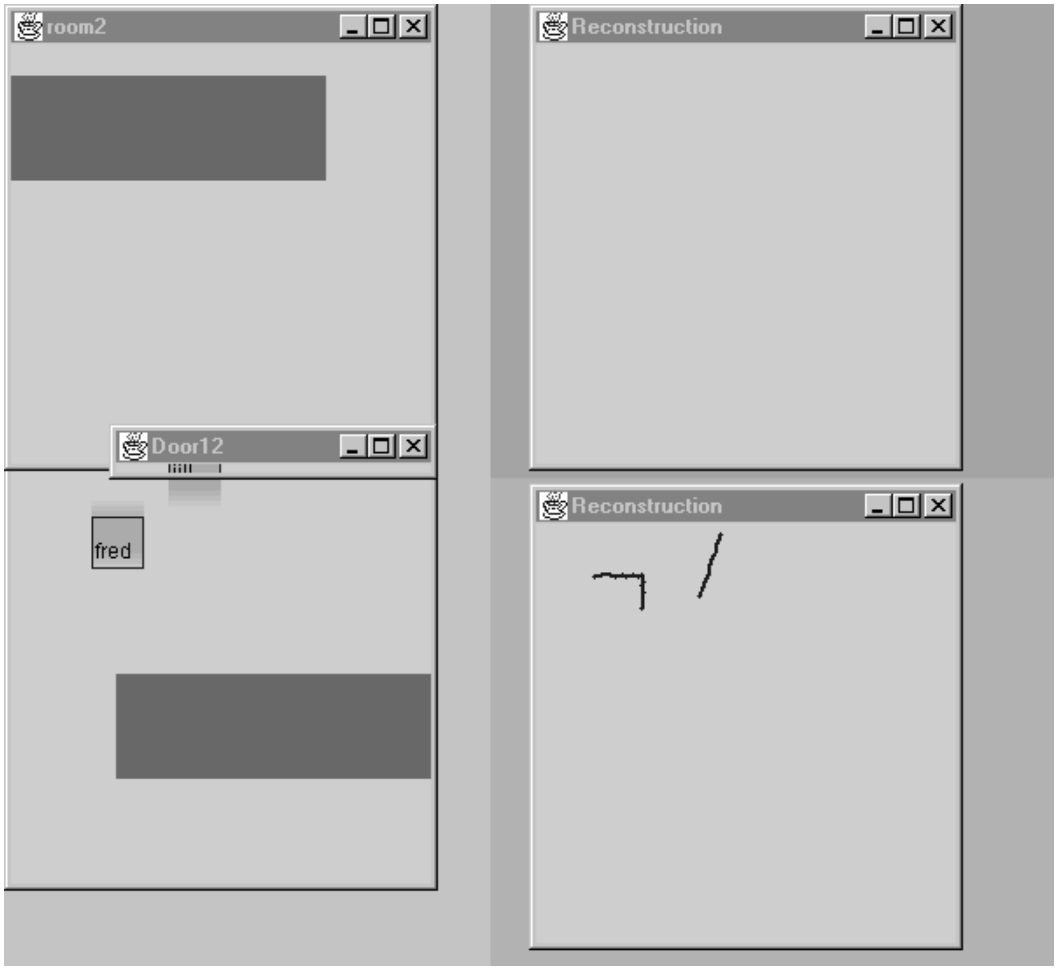



Figure B.2: The Smart Building at time t=15

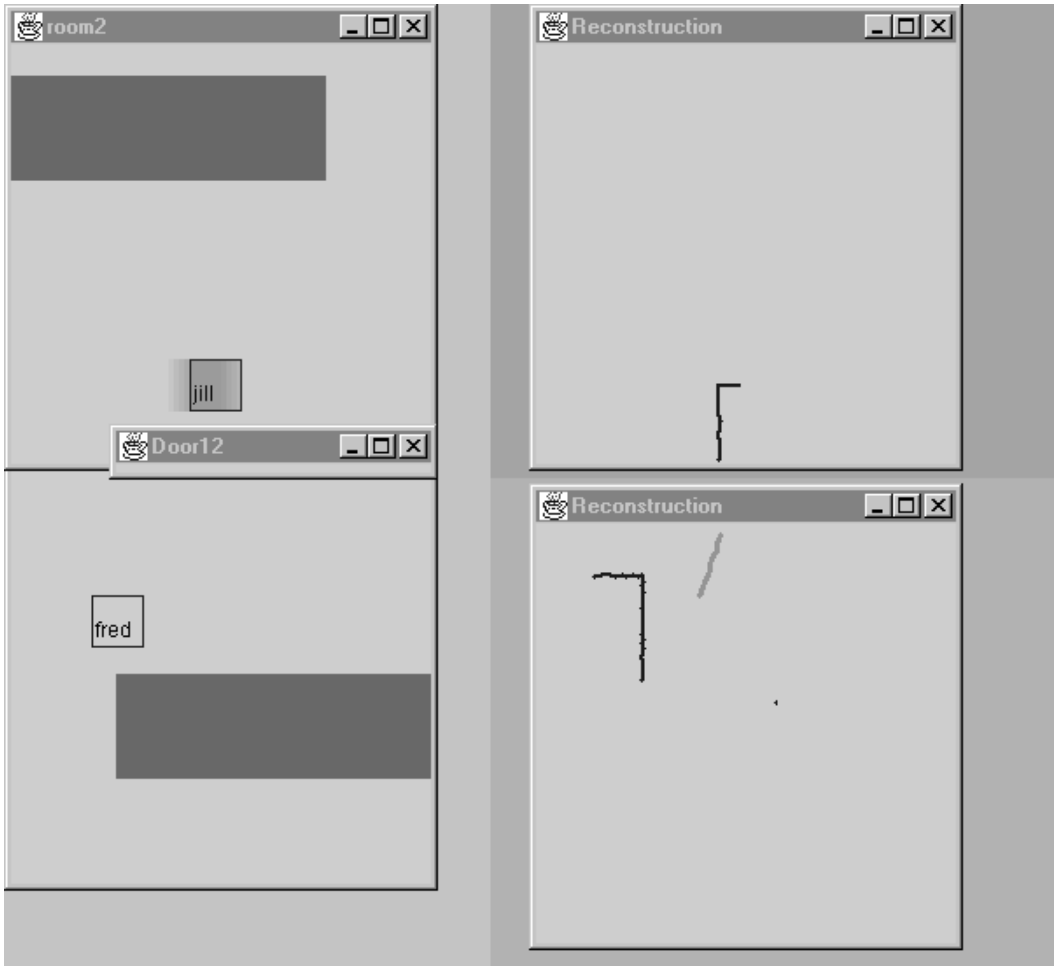



Figure B.3: The Smart Building at time t=35

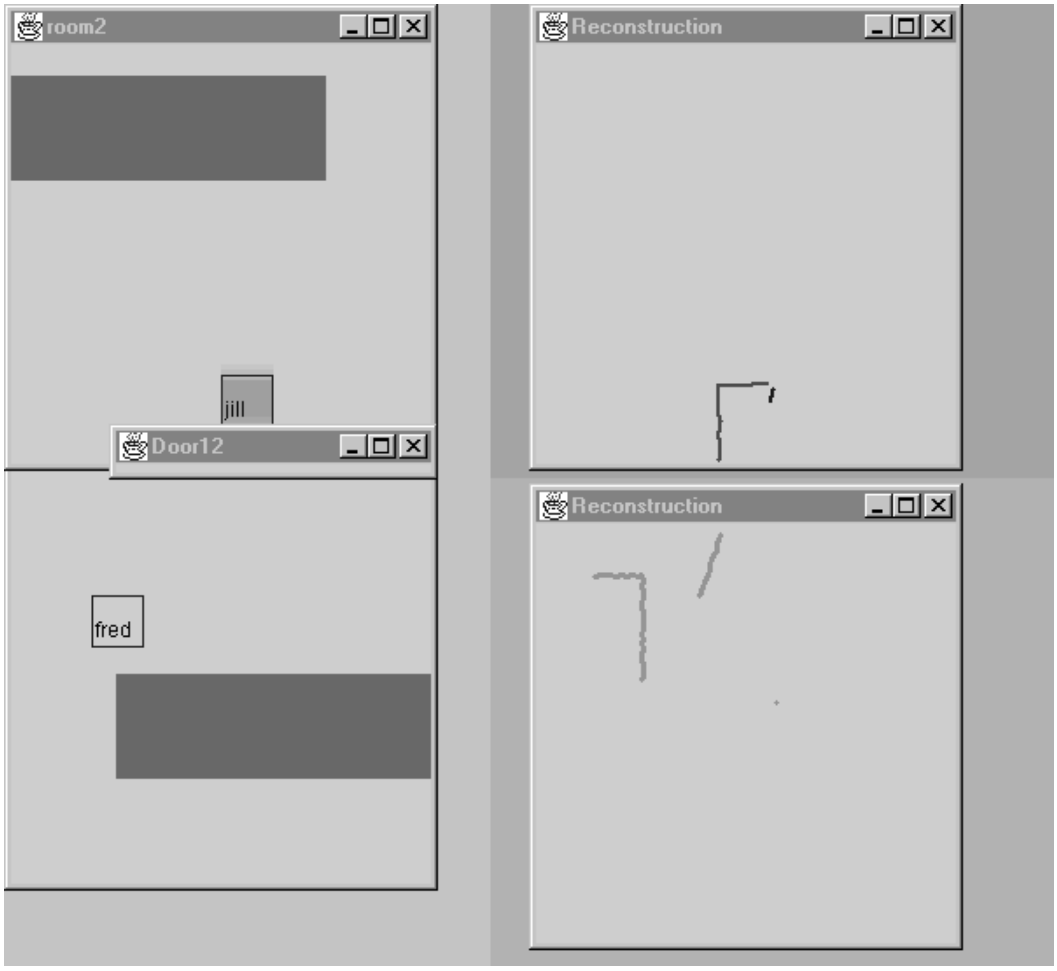



Figure B.4: The Smart Building at time t=55

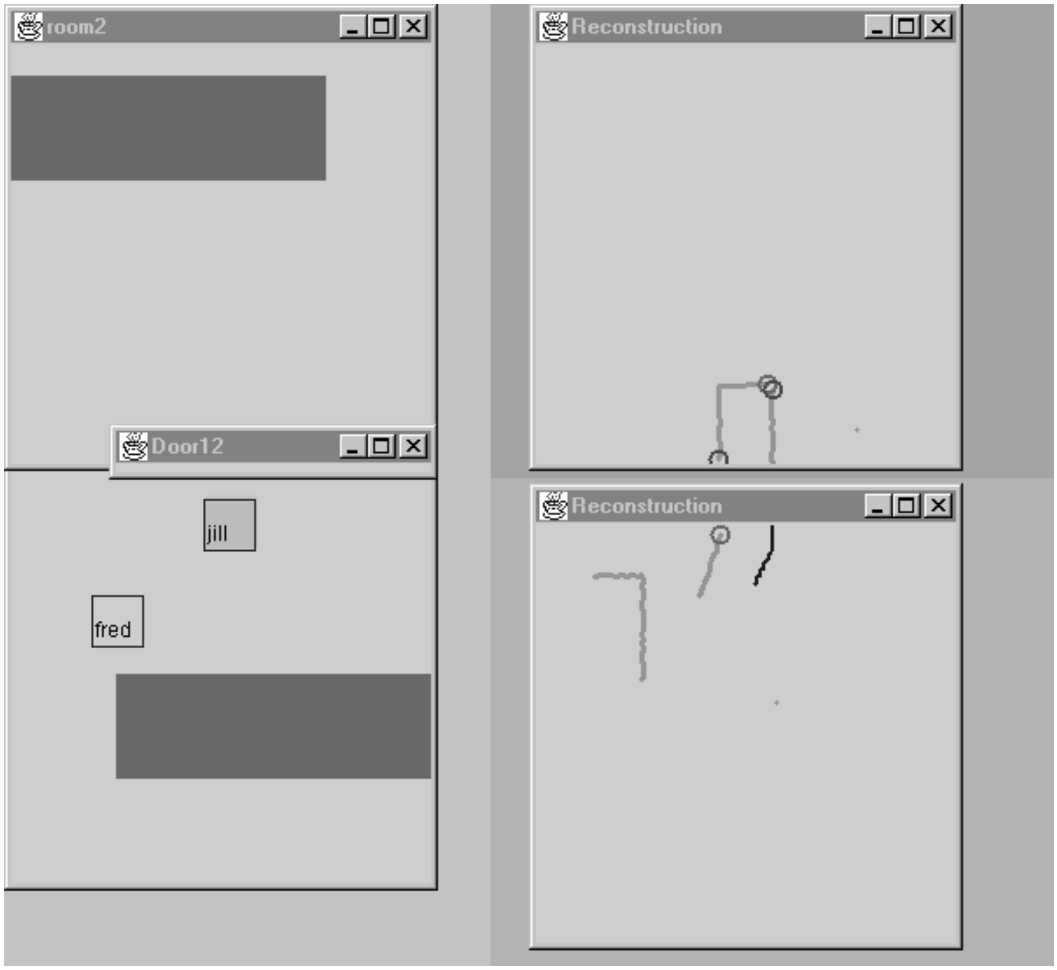



Figure B.5: The Smart Building at time t=80