Genetic Programming
and
Protocol Configuration

Paul Grace

Supervisor: Professor Gordon Blair

Dissertation for the degree of Master of Science in Distributed Interactive Systems

Computing Department
Lancaster University

September 2000

Supporting Documentation: http://members.tripod.co.uk/pjg1000/GP/

Abstract

Dynamic protocol stacks have been identified as a method of improving the performance of process communication by constructing the protocol that meets it requirements with a minimum overhead. However, the design of these stacks is a complex process for human designers who must identify the correct elements and ordering. Genetic programming is a machine learning technique method for generating computer programs automatically. It has been applied to many areas including electronic circuit design, image classification and machine code creation. However, it has never been utilised in the area of communication protocols. The main aim of the project is to identify if genetic programming techniques can be applied to protocol construction using the JavaGroups toolkit and generate protocols automatically.

This report describes the theory and application of genetic programming principles and also the technology behind and creation of dynamic protocols. There is also a description of the design and implementation of a genetic program that will create JavaGroups protocols to meet a set of requirements.

The results from the experimentation of the implemented system identify that GP can automatically generate the correct protocol stack for a required communication type (e.g. reliable point-to-point). The uses of this method to generate any stack depending on the user's requirement or within a changing real-time network situation are identified as future areas of work.

Table of Contents

Chapter 1 Introduction ………………………………………………………. 5

1.1 Scenarios and areas of interest …………………………………………………… 6
1.2 Aims of the project ……………………………………………………………….. 7
1.3 Structure of the report …………………………………………………………….. 8

Chapter 2 Genetic Programming ………………………………………………. 9

2.1 Motivation ………………………………………………………………………… 9
2.2 Genetic Programming Theory ……………………………………………….. 9
2.2.1 Representing the program ………………………………………………... 10
2.2.2 Initialising a Genetic Program …………………………………………… 11
2.2.3 Genetic operators ………………………………………………………… 12
2.2.4 Fitness and Selection …………………………………………………….. 13
2.2.5 The Basic Genetic Programming algorithm ……………………………….. 14
2.3 Applications of Genetic Programming ………………………………………….. 15
2.3.1 Electronic Circuit Design ………………………………………………… 15
2.3.2 Image classification ……………………………………………………… 15
2.3.3 Machine language ………………………………………………………… 16
2.4 Genetic Programming Toolkits ……………………………………………………. 16
2.4.1 GPsys …………………………………………………………………….. 16
2.4.2 Lil-GP ……………………………………………………………………... 17
2.4.3 GPC++ ……………………………………………………………………. 17
2.4.4 Choice of Toolkit …………………………………………………………. 17
2.5 Example GP runs ………………………………………………………………….. 17
2.5.1 Genetic Program for $y=x^2/2$ …………………………………………………. 17
2.5.2 Genetic Program for $y=x+x^2+x^3+x^4$ ……………………………………… 19
2.6 Conclusions ……………………………………………………………………….. 19

Chapter 3 Dynamically Configurable Protocols …………………………………… 21

3.1 Motivation …………………………………………………………………………. 21
3.2 Frameworks for protocol configurations ……………………………………………. 21
3.2.1 Early Systems …………………………………………………………….. 21
3.2.2 Group communication systems at Cornell University ……………………… 23
3.2.3 Quality of Service frameworks ……………………………………………. 25
3.2.4 An alternative approach ………………………………………………….. 27
3.3 Choice of Toolkit ………………………………………………………………….. 27
3.4 JavaGroups Architecture and Implementation details ………………………………... 28
3.5 Conclusions ……………………………………………………………………….. 30

Chapter 4 Application of Genetic Programming to Protocol Configuration ……. 32

4.1 Introduction ……………………………………………………………………….. 32
4.2 Key Design choices ……………………………………………………………….. 32
4.2.1 Representing a stack within the Genetic Program …………………………. 32
4.2.2 Genetic Operators acting on the Property stack ……………………………. 34
4.2.3 Fitness Functions …………………………………………………………. 34

4.3 Design & Implementation of Fitness Tests …………………………………………… 35
4.4 Conclusions ………………………………………………………………………… 37

Chapter 5 A First Experiment: Reliable Point-to-Point Communication ………. 38

5.1 Introduction ……………………………………………………………………... 38
5.2 Design of the Genetic Program …………………………………………………. 38
5.2.1 Introduction ……………………………………………………………… 38
5.2.2 Generation of Stacks ……………………………………………………….. 38
5.2.3 Fitness Function and Tests ……………………………………………… 39
5.3 Results of the Experiment ……………………………………………………….. 41
5.4 Conclusions ………………………………………………………………………… 42

Chapter 6 An advanced experiment: Reliable, Ordered Multicast communication .. 43

6.1 Introduction ………………………………………………………………………… 43
6.2 Design of the Genetic Program …………………………………………………. 43
6.3 System Architecture ……………………………………………………………... 45
6.4 Protocol Tests ……………………………………………………………………… 45
6.4.1 Communication Test ………………………………………………………45
6.4.2 Reliability Test …………………………………………………………… 47
6.4.3 Group Communication …………………………………………………… 48
6.4.4 Ordering Test …………………………………………………………….. 48
6.4.5 Quality of Service Test …………………………………………………... 50
6.5 Design and Implementation of a micro protocol ……………………………... 51
6.6 Testing and Results ……………………………………………………………… 52
6.7 Analysis of Results ……………………………………………………………… 52
6.8 Conclusions ………………………………………………………………………… 53

Chapter 7 Conclusions ………………………………………………………………. 54

7.1 Overview of Work ……………………………………………………………… 54
7.2 Major contributions ……………………………………………………………… 54
7.3 Other notable contributions ……………………………………………………… 55
7.4 Further Work ……………………………………………………………………….. 56
7.4.1 Future testing of current work ………………………………………… 56
7.4.2 Performance improvement ……………………………………………. 56
7.4.3 User specification of the Genetic Program's output ……………………….. 56
7.4.4 Dynamic adaptation …………………………………………………… 57
7.5 Concluding remarks ………………………………………………………………57

Chapter 8 References ………………………………………………………………. 58

List of Figures

2.1 Even 2-parity function depicted as a rooted, point labelled tree ……………… 10
2.2 Linear representation of the function x= (x+12)-42 ………………………….. 11
2.3 Crossover operation applied to two parent trees ……………………………… 12
2.4 Linear crossover within two machine code programs ………………………… 13

3.1 The STREAM architecture …………………………………………………. 22
3.2 The Da Capo framework ……………………………………………………25
3.3 A simple sending a receiving group member using the JavaGroups toolkit ….. 30

4.1 Tree representation of UDP: NAKACK: UNICAST: FLUSH: GMS ………… 33
4.2 Core section of genetic programming system …………………………………36

5.1 Genetic Programming system for generating a reliable
point-to-point protocol stack…………………………………………………... 40

6.1 Architecture of the Genetic programming system to create reliable,
ordered multicast protocols …………………………………………………46
6.2 Graph representing the fitness measure for a given throughput value …………50

List of Tables

2.1 Fitness cases in the training set ……………………………………………….. 18
2.2 Results of GP for $x=x^2+x^3+x^4$ …………………………………………………19

3.1 Example protocol stack …………………………………………………….. 24
3.2 Description of the available protocol module in the JavaGroups toolkit ………29

5.1 Test results for reliable, unicast communication ……………………………… 41

6.1 Description of the methods used in the genetic programming system ……….. 47
6.2 Results of the GP runs for reliable multicast protocol generation …………… 52
6.3 Results of GP runs for reliable, ordered multicast protocol generation ………. 52
6.4 Explanation of fitness value returned for the five tests ……………………….. 53

Chapter 1 Introduction

1.1 Scenario and Areas of Interest

[Coulouris et al, 1994] define a *protocol* as a set of rules and formats to be used for communication between processes in order to perform a given task. A protocol is implemented by a pair of software modules located in the sending and receiving computers i.e. the protocol of the sending machine transforms the message into a format suitable for transmission over the network, while the receiving protocol module performs the inverse transformations on the message to regenerate it. Protocol modules are normally made up of a number of layers; each layer carries out some protocol functionality by applying transformations to encapsulate the data before passing the message to the layer above or below, depending on whether it is sending or receiving. A complete set of protocol layers is referred to as a *protocol stack*. The choice of stack for use in a communication system is an important design decision.

When a set of processes communicate with each other the functional requirements of this communication may change over time depending on the needs of the users; these could take the form of *reliability* (all messages sent must be received by every receiver), *ordering* (the messages must be seen by every receiver in the order they were sent) or *security* (unauthorised viewing of the communication must be prevented). There may also be *Quality of Service* requirements when dealing with distributed multimedia applications, such as video and audio conferencing. Quality of Service refers to the abstract specification of the non-functional requirements of a service. For example, timeliness can be specified by the end-to-end latency of frames or permitted jitter on frames in a stream interaction. Also, volume can be specified as the perceived throughput in bytes per second of a discrete interaction.

A possible solution is to maintain a single protocol stack to handle the requirements of every perceivable communication type. However, this would make it become complex and perform poorly. One way of preventing this is in the use of dynamic protocol stacks. Dynamic protocol stacks consist of individual building blocks that define a single protocol task, which are then layered together. These building blocks are fine-grained components called *micro-protocols*. The appropriate stack is constructed for each connection, which allows it to be minimal in size and provide the best performance. For example, the JavaGroups toolkit that is being used as part of the project is a group communication system that allows the implementation of group-based applications and is one of a number of systems that utilises configurable protocols.

A human designer who needs to create protocols for use in the JavaGroups toolkit [Cornell, 1999] or indeed any… must first identify the micro-protocol building blocks required, and then design the order of the layering and finally, must write the code to implement the protocol. This is a time consuming process, especially if a number of protocols are being implemented. The motivation behind this project is to investigate if this task can be automated (by applying genetic programming principles), so the JavaGroups protocol stacks can be generated without human intervention.

Genetic programming (GP) is a machine learning method that automatically generates the computer program for a specified problem. [Koza, 1992a] defines genetic programming to be a domain independent method that genetically breeds populations of computer programs to solve problems. Genetic programming starts with a set of randomly generated computer programs constructed of the available programmatic elements and then applies the principles of Darwinian evolution to breed a new generation of programs. The iterative process of applying evolution to each new generation is repeated until the optimum solution to the problem is obtained.

Therefore, the key to the technique lies in the evolution from one generation of programs to the evolved generation. Like the "survival of the fittest" theory, genetic programs select only the fittest or most suitable programs for evolution. This means that an evaluation metric called a fitness function (this measures how well the program solves the problem) is applied to each member and assigns it a fitness value. Genetic operators are then applied to the fittest programs to create the next generation.

[Koza, 1992a] states there are three genetic operators. Reproduction copies the selected program to the next generation. Crossover creates new offspring programs for the next generation by recombining randomly chosen parts from the two selected programs. Finally, mutation creates one new offspring program by randomly mutating a randomly chosen part of the selected program.

1.2 Aims of the Project

Genetic programming is a new technique in the domain of machine learning and has never been applied to the area of protocol construction. Therefore, this project examines the benefits genetic programming can bring to dynamic protocol configuration and identify if this new field of application is feasible.

Therefore, the main aims are:

- To identify what it means to apply GP to the domain of protocols. What is the fitness of a protocol stack? How can the fitness be measured? How do the genetic operators affect a protocol stack?
- To assess how reliable a method GP is in the automatic creation of protocols. Can it always produce the correct protocol? Is it better that human performance in the design of the protocol for a communication?
- To investigate how expensive the technique is. What is the time taken to produce a stack automatically? How much resources are needed to perform the task?

In order to complete these aims the following sub goals needed to be achieved:

- Evaluate the available genetic programming toolkits and choose the most suitable for the implementation of the solution.
- Design the fitness function and selection method, which will be used to measure how well the generated protocols meet their functional and non-functional requirements and decides if the genetic program will evolve it further.

- Design the GP algorithm i.e. how natural selection is performed. This is achieved by designing how the GP represents the protocol structure and how the evolution operators are applied to these structures.
- Implement and test a genetic program that automatically generates the protocol stack for reliable point-to-point communication. This will be evolved from a smaller subset of available layers that are needed to provide the required functionality.
- Extend the previous goal's genetic program to automatically generate protocol stacks for a larger domain of the protocol layer, to produce a protocol stack that provides reliable, ordered multicast communication.

1.3 Structure of the report

Chapter 2 of this report describes the area of genetic programming in detail. The theory and areas of its application are discussed. Currently available toolkits for implementing genetic programming solutions are evaluated and a choice is made of an appropriate toolkit. Finally, two simple genetic programs are stepped through to show the process in action.

Chapter 3 examines the area of dynamically configurable protocols. It inspects the properties and design of eight systems that use lightweight protocol modules or allow the protocol to dynamically adapt to its environment. This section also describes the reasons behind the choice of the JavaGroups toolkit as part of the implementation.

Chapter 4 details the basic design of the genetic programming system that is used to generate the protocol stacks for the two types of communication. It illustrates how genetic programming is applied to the domain of configurable protocol in terms of representing the stack and testing each member of the population and designating it a fitness value.

Chapter 5 describes the design of the experiment to automatically generate a protocol that provides reliable point-to-point communication. It includes the testing of the experiment and analyses the results that were obtained.

Chapter 6 describes the design of a set of experiments to automatically generate a family of protocols that provides reliable, ordered multicast communication. The testing and results obtained from these experiments are demonstrated.

Chapter 7 draws conclusions from the work done, evaluating how successful the application of genetic programming to the domain of protocol configuration has been. Possible future work in this area is identified.

Chapter 2 Genetic Programming

2.1 Motivation

[Banzhaf et al. 1997] state that the automatic programming of computers without the need for a human programmer will be one of the most important areas of computer research over the next twenty years. This is due to the exponential leap forward of hardware capability and speed, while the software lags behind. Advances in software development, such as object-oriented programming, object libraries and rapid prototyping still leave the development of code in the hands of the programmer. The time taken to code complex systems by a human means that the software is often obsolete before it is released i.e. it does not allow for the advances in hardware that have since taken place.

The main goal of genetic programming is to be able to tell the computer what task we wish it to perform and to have it learn how to perform that task. [Koza et al, 1996b] summarise this as*, an automatic programming technique in which what you want is what you get (WYWIWYG).* Therefore, genetic programming aspires to allow the computer to program itself or other computers. However, it is not capable of this feat just yet. This remainder of this chapter describes the current operation of genetic programs, how they are implemented and the problem areas they have been applied to.

2.2 Genetic Programming Theory

Like other artificial intelligence techniques, the theory of genetic programming is based on models of the natural world. In nature, the evolutionary process occurs if an entity can: reproduce, has the ability to survive in its current environment and there is variety in the set of reproducing entities. Individuals that are better able to perform tasks in their environment (i.e. the fittest individuals) survive and reproduce at a higher rate; less fit entities might not survive and if they do, reproduce at a lower rate. This is the concept of *"survival of the fittest"* and *"natural selection"* [Darwin, 1859]. It is this model that the theory of genetic programming is based upon. A population of computer programs are evolved using techniques similar to their natural equivalent to produce the fittest computer programs for solving a problem.

[Banzhaf et al, 1997] recognise three fundamental features of all genetic programming systems that are described in further detail in the remainder of this section:

- **Program structures**. Every genetic program assembles variable length program structures from basic units called *functions* and *terminals*.

- **Genetic operators**. The initial programs in the population are transformed using genetic operators.

- **Simulated evolution of a population by means of fitness based selection**. The simulated evolution is driven in some form of fitness-based selection. Fitness based selection determines which programs are selected for further improvements.

2.2.1 Representing the program

The first step of any genetic program is to generate an initial population of computer programs from which the fittest can be selected and evolved. The functions and terminals are the primitives with which a program in genetic programming is built. *Terminals* provide a value to the system, while *functions* process a value already in the system.

Every genetic program is made up of a *terminal set* and a *function set*. "The terminal set is comprised of the inputs to the genetic program, the constants supplied to the genetic program and the zero argument functions. The function set is composed of the statements, operators and functions available to the GP system" [Banzhaf et al, 1997]. For example, given the mathematical problem in equation 2.1:

$$F(x) = Sin(x) + \sqrt{2x + Cos(x)} \qquad \textbf{(Equation 2.1)}$$

A possible function set may be {sin, cos, +, √} and the terminal set could be {x, -10 to 10}. The genetic program then uses these two sets to create an initial population of programs by randomly generating a set of structures made up of elements from these two sets.

[Banzhaf et al, 1997] identify that the choice of elements for the terminal and function set is an important design decision. The sets should at least hold the appropriate elements to solve a problem. For example, a GP system for solving mathematical problems with only an add operator in the function set, would only solve trivial problems, while a larger set of arithmetic functions could solve more diverse problems. However, the function set must not be too large, because the search space is increased and the search for a solution is therefore harder.

[Koza, 1992a] initially represented each program as a tree structure corresponding to the parse tree of the program. A node in the tree represents each function call and its descendant nodes give the arguments to the function. An example program tree is shown in figure 2.1.

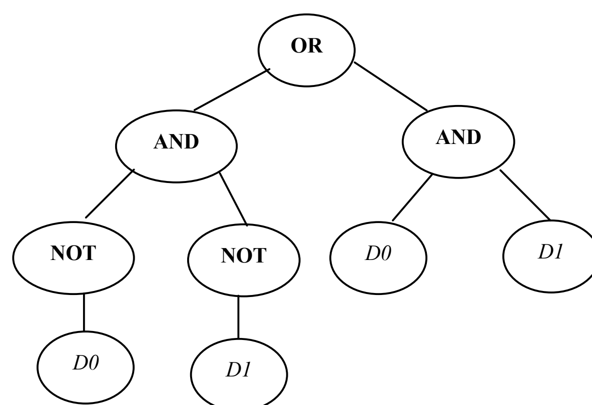

Figure 2.1 Even 2-parity function depicted as a rooted, point labelled tree. [Koza, 1992a]

However, [Banzhaf et al, 1997] state *it is already clear from the GP literature that programs or programming language structures may be represented in ways other than trees. Therefore,*

we do not limit our definition of GP to include only systems that use expression trees to represent programs. Other possible representations are linear and graph structures.

A linear structure is simply a set of instructions that execute from top to bottom. The function set and terminal set are combined to create these individual instructions rather than be placed as separate elements within a tree. Figure 2.2 shows the linear representation of a simple machine code program.

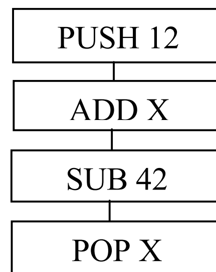

Figure 2.2 Linear representation of the function x= (x+12)-42

PADO [Teller and Veloso, 1995] is the name of a graph based GP system. Graphs are capable of representing very complex program structures. A graph structure is simply a set of nodes connected by edges, where an edge is a pointer between two connected nodes indicating the direction of flow of program control.

2.2.2 Initialising a Genetic Program

The first stage of a GP run is to create a set of program structures that can be evolved into future generations of programs. This first set is called the *initial population*. The process of initialisation is different for the tree and linear representations.

[Koza, 1992a] defines two methods for initialising tree structures called *full* and *grow*. The grow method produces trees of an irregular shape, because nodes are randomly selected from both the function set and the terminal set throughout the tree (except the root, which is obviously a function). Once a branch contains a terminal node, that branch has ended. However, instead of selecting nodes randomly from the function and terminal set the full method chooses only functions until the node is at the designated maximum tree depth. Then it chooses only terminals. The result is that every branch goes to full depth.

[Banzhaf et al, 1997] note that diversity is valuable in GP populations and that these methods could result in uniform sets of structures, because the routine is the same for all individuals. Therefore, the *ramped half-and-half* method [Koza, 1992b] was devised. It simply uses a set of maximum possible depths e.g. if maximum 6 is set then trees with depths 2, 3, 4, 5, 6 are initialised and for each of these groups half of the trees are created with the full method and the remainder with the grow method.

Initialisation of linear structures [Banzhaf et al, 1997] takes the form of randomly generating a set of instructions taking values from the function set and the terminal set and creating one individual command. The maximum length for an instruction sequence is given and a random number between this and 2 is generated. An individual structure, therefore, holds this number of the randomly generated instructions. It is identified that equivalent methods to full and grow may be applied in this area.

2.2.3 Genetic operators

This section describes the three primary operators used to modify the structures undergoing adaptation in genetic programming; they will be described in their operation on both tree and linear structures. They have also been adapted to cover graph structures [Teller, 1996] but this is not discussed here. The three most commonly used genetic operators are:

- Crossover
- Reproduction
- Mutation

[Koza, 1992a] states that tree crossover creates variation in the population by producing new offspring that consist of parts taken from each parent. Two parents are chosen because of their fitness. In each parent a random sub-tree is chosen. The sub-trees are then swapped. The resulting two trees are the individuals that are passed to the next generation. This is shown in figure 2.3.

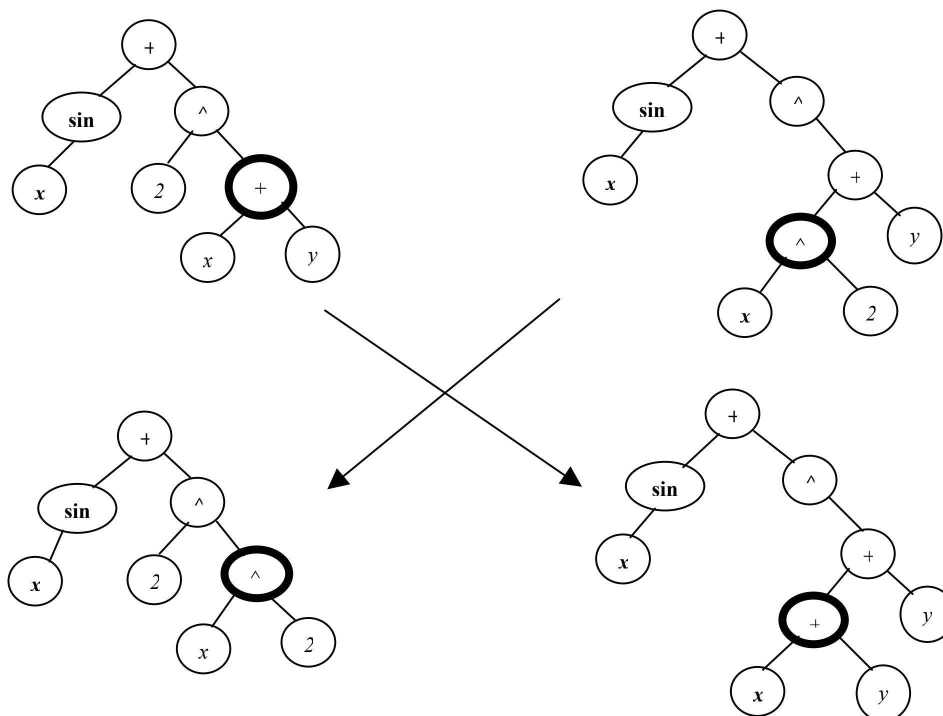

Figure 2. 3 Crossover operation applied to two parent trees (top). Crossover nodes (shown in bold) are chosen at random. The sub-trees rooted at these crossover points are then exchanged to create children trees. [Mitchell, 1996]

Linear crossover [Banzhaf et al, 1997] swaps linear segments of code between two parents instead of sub-trees. A random sequence of instructions is taken from each parent and swapped leaving two new individual children. This is shown in figure 2.4.

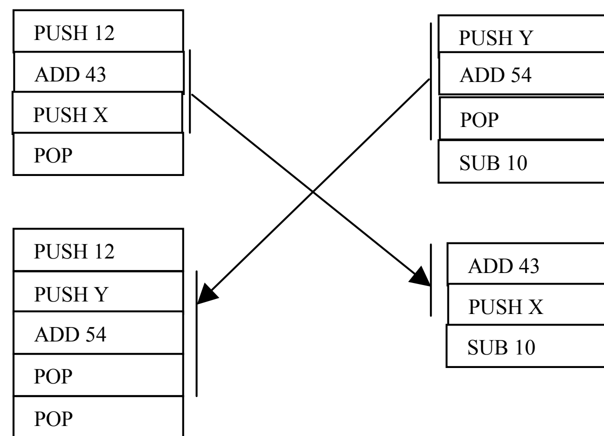

Figure 2.4 Linear crossover within two machine code programs

[Koza 1992a] describes the tree reproduction operator as the basic engine of natural selection. The operator is asexual in that it operates on only one tree and produces only one tree when it is performed. The operation consists of two steps; first a single tree is selected from the population on the basis of its fitness and then it is copied without alteration into the new population (i.e. new generation). The operation is identical for linear based structures.

Mutation operates on a single individual in the population. In the case of tree mutation described by [Koza 1992a] this introduces random changes to tree structures within the population. The operation begins by selecting a point at random within the tree and then removing whatever is currently at and below that point and replacing it with a randomly generated sub tree.

Linear GP mutation differs from the method for tree structures [Banzhaf et al, 1997]. When an individual is chosen for mutation, the mutation operator first selects one instruction from that individual for mutation. It then, makes one or more changes within that instruction. The changes may take the form of different operators, variables or constants being used, or a completely new instruction being inserted.

2.2.4 Fitness and Selection

Genetic programming must choose which members of a population will be subject to the genetic operators described previously. The evaluation metric used is called the *fitness function*. This is defined by [Banzhaf et al, 1997] as the measure used during simulated evolution of how well a program has learned to predict the outputs from the inputs. The aim of a fitness function is to give feedback to the evolutionary algorithm concerning which members should be given a higher probability of reproducing and which are more likely to be removed. The function should, therefore, be designed to give graded (rather than Boolean)

feedback about how a program performs. In *standardized fitness* [Koza, 1992a] the fittest individual is assigned a fitness value of zero.

Having given each member of the population a fitness rating the next step is to select those to be used for future generations. There are a number of selection algorithms to perform such a task. *Fitness proportional selection* [Holland, 1975] specifies probabilities for individuals to reproduce offspring based on its fitness value, using the function in equation 2.2:

$$p_i = f_i \Big/ \sum_i f_i \qquad \textbf{(Equation 2.2)}$$

Ranking selection [Grefenstette and Baker, 1989] is based on the fitness order into which the individuals can be sorted. The ranking of an individual in terms of its peers determines whether it is then selected. *Tournament selection* is based on the competition between subsets of the population. The winners of these mini competitions are then selected for future evaluation. This has become a popular method because it does not require a centralised fitness comparison of all individuals. Therefore, saving computation time and allowing for the selection to be run in parallel.

2.2.5 The Basic Genetic Programming algorithm

Having described all the individual elements that make up a genetic programming system. The complete algorithm for a GP run can now be described. The preliminary steps that must be carried out to create a GP are as follows [Banzhaf et al, 1997]:

- Define the terminal set for the problem domain.
- Define the function set for the problem domain.
- Define the fitness function.
- Define parameters such as the initial population size, the probability of crossover, the maximum size of an individual, a selection method and the termination criterion (maximum number of generations or a fitness value).

Once these steps have been completed the GP algorithm defined by [Koza, 1992a] can commence. This algorithm is based on generating new evolutions of programs, which are well defined and distinct. The newer population is created from and then replaces the older population. This cycle follows these steps:

1. Initialise the population
2. Evaluate the individual programs in the existing population. Assign a numerical rating or fitness to each individual.
3. Until the new population is fully populated, repeat the following steps
    a. Select an individual in the population using the selection algorithm
    b. Perform genetic operations on the selected individual.
    c. Insert the result of the genetic operation into the new population
4. If the termination criterion is fulfilled, then continue. Otherwise replace existing with a new population using steps 2-3.

5. Present the best individual in the population as the output from the algorithm

2.3 Applications of Genetic Programming

Genetic programming has been applied to many problem areas covering a range of science, computer science and engineering domains. This section looks in detail at some of the applications that are relevant to the project domain. That is, they attempt to generate linear structures or they utilise building blocks as the primitives given to the genetic program. These applications are Electronic Circuit Design, machine language programming and Image classification.

2.3.1 Electronic Circuit Design

[Koza et al, 1996a] summarize the use of a genetic program in the significantly complex problem of designing electronic filter circuits. The primitive functions used by the GP to construct its programs are functions that edit the circuit by inserting or deleting circuit components and wiring connections. The fitness of each program is calculated by simulating the circuit's outputs (using the SPICE circuit simulator) to determine how closely this circuit meets the design specifications for the desired filter. The fitness score is the sum of the magnitudes of errors between the desired and actual circuit output at 101 different frequencies. The results of running the GP showed that a high percentage of the initial simulatable population were unable to be simulated, but at each generation this percentage dropped. After 137 generations the best circuit produced behaviour similar to that desired.

The use of genetic programming in a domain as complex as electronic circuit devices shows that GP can be applied to other problem areas not just to computer program and mathematical function generation. It also illustrates that it is possible to incorporate abstract building blocks, as long as the input and outputs from them can be simulated in an appropriate fitness test.

2.3.2 Image classification

[Teller and Veloso, 1996] implemented the PADO system for classification of images. As described previously it is a graph based genetic program. Classifying images means dealing with large sets of data and fuzzy relationships. The problem domain of creating image-processing algorithms is a complex one and time consuming for a human creator; therefore, it is suitable for machine learning techniques.

The generated program for classifying an image is tested on one image at a time, extending the solution to recognise more objects on each GP run. Results from testing the system using 100 training images showed that classification was between 70% and 90% correct.

Like electronic circuit design this is a complex problem to solve, and involves the use of abstract building blocks within the solution to create a program. The terminal set contains

domain specific functions: PIXEL, VARIANCE, and DIFFERENCE. These provide the methods that help classify images that are passed as inputs to the generated program.

2.3.3 Machine language

Machine code is considered hard to learn, program and master when compared to higher level programming languages. Therefore, it is desirable for it to be generated automatically using genetic programming. The GEMS system [Crepeau, 1995] provides an interpreter for the Z80 microprocessor. GEMS implements 660 of the 691 possible instructions and has been used to evolve a program for generating the string "hello world" made up of 58 instructions. Each instruction is viewed as atomic and individual and therefore, the population members are made up of linear strings of these instructions.

Another linear genetic programming system used to generate machine code is the AIMGP (Automatic Induction of Machine Code with Genetic Programming) system [Banzhaf et al, 1997], which includes the notion of memory registers for supplying the input values and output stores for the machine code instructions.

The application of genetic programming to this domain using the technique of linear structures, rather than the original tree concepts show that GP is a flexible method. That is, it can be modified to suit the problem area, rather than shaping the problem to work with the original tree-structured concepts.

2.4 Genetic Programming Toolkits

Given the genetic programming algorithm in section 2.2.5 it is possible to implement your own code for generating an evolutionary program. However, there are a number of toolkits available to overcome this complex and time-consuming task. This section describes some of the available systems and then reasons why one was selected to perform the project.

2.4.1 GPsys

GPsys [Quereshi, 1998] is a Java based genetic programming system developed by Adil Quereshi. Programs must be represented as tree structures developed by simply defining the function and terminal set. Java classes are available to set genetic programming parameters such as population size, number of generations, grow method and crossover probability. The selection algorithm used is the tournament method; therefore, the system is memory efficient.

However, the main features of this toolkit are that it supports generic functions and terminals. The user must simply write a Java class exhibiting the behaviour of the function or primitive and it is then easily fitted into the system. The fitness definition also allows the user to write an algorithm for generating the value that can then be passed to the toolkits evolutionary algorithm.

2.4.2 Lil-GP

Lil-GP [Punch & Zongker, 1998] is a C programming language, tree-based genetic programming toolkit. It provides a graphical user interface for creating the GP and viewing the generational results. Its features are that it is difficult to change fitness and program elements, because creating new DLLs for each does this. It is also difficult to incorporate information from outside programs.

2.4.3 GPC++

GPC++ [Fraser, 1994] provides a library of C++ functions for creating a Genetic program in a similar vein to the GPsys system. Standard functions are available for initialising tree structures and the GP run. Methods can be defined for the fitness function and the function and terminal sets.

2.4.4 Choice of Toolkit

The GPsys toolkit was chosen as the implementation tool for the project because it is Java based and, therefore, will allow easier co-ordination between itself and the JavaGroups communication toolkit. The use of the Java programming language was also important because the project implementer was more experienced using this language than the other two languages. The other important features of the system were its flexibility in defining the elements of the structure and the fitness function, without any constraints on what can be achieved.

None of the available systems provide diversity in the type of program representation i.e. linear or graph based as opposed to the standard tree. Therefore, the flexibility in defining the tree structure must be an important factor.

2.5 Example GP runs

This section demonstrates how the process of creating a genetic program to solve a problem is carried out using the chosen toolkit for its implementation. Two simple mathematical functions ($y=x^2/2$ and $y=x+x^2+x^3+x^4$) are solved.

2.5.1 Genetic Program for $y=x^2/2$

The following describes the steps in creating a program for automating the function:

$$y = f(x) = \frac{x^2}{2} \qquad \textbf{(Equation 2.3)}$$

Ten fitness cases were used for the testing of the functions, taken in the interval [0,1] and shown in table 2.1.

	Input	Output
Fitness case 1	0.0	0.000
Fitness case 2	0.1	0.005
Fitness case 3	0.2	0.020
Fitness case 4	0.3	0.045
Fitness case 5	0.4	0.080
Fitness case 6	0.5	0.125
Fitness case 7	0.6	0.180
Fitness case 8	0.7	0.245
Fitness case 9	0.8	0.320
Fitness case 10	0.9	0.405

Table 2.1 Fitness cases (input and output values) in the training set

By following the steps described in section 2.2 a genetic program can be generated. For this problem these are:

1. Select a terminal set: Variable x, integer constants between –5 and +5. {x, -5, …,5}
2. Select a function set: Arithmetic functions +, -, *, /. {+, -, *, /}
3. Fitness function: If y_i is the output of the ith training set member and x_i is the input value of the ith training set member. Eval($prog_i$, x_i) returns the output of the current individual given input x. Then the fitness test can be described by:

$$Fitness(x) = \sum_{i=1}^{10} y_i - eval(prog_i, x_i)$$

4. Initial population size = 600, crossover probability= 90%, mutation probability = 5%, Number of generations = 5, tournament size = 7.

These steps are implemented in the GPSYS toolkit by creating the following java programs:

- Example1Parameters.java defines the terminal set and function set
- Example1GPParameters.java defines the parameters of the GP run
- Example1Fitness.java defines the fitness function as described above
- Example1.java controls the overall definition and execution of the GP

When fully implemented the GP was run and the following results were obtained. On three separate runs the following individuals were returned:

- (mul (div *xfloat 2*) *xfloat*)
- (div (mul *xfloat xfloat*) 2)
- (mul (add *xfloat xfloat*) (div *xfloat* 4))

All three are equivalent to $x^2/2$ and therefore the genetic program has returned a number of functions that will provide the correct code for solving the problem. The differences between the results lie in the complexity of the functions. The last result is the most complex; it uses the most functions and therefore will perform the worst. The next experiment examines how the number of generations and population size affects the result.

2.5.2 Genetic Program for $y=x+x^2+x^3+x^4$

A second more complex task is to implement a genetic program to automatically generate the solution for:

$$y = f(x) = x + x^2 + x^3 + x^4 \quad \textbf{(Equation 2.4)}$$

The same process used in 2.5.1 was performed using 10 fitness cases, the same fitness test, and terminal set. However, the function set was extended to {+, -, *, /, sin, cos, log}.

The results from running the GP 5 times with a different population size and number of generations each time are shown in table 2.2.

Population	Number of Generations	Result
500	10	(+ (+ (+ (* x x) x) (* x (* x x))) (* (* x x) (* x x)))
750	10	(+ (* (* x x) (* x x)) (+ (+ x (* x x)) (* (* x x) x)))
200	20	No correct result
1000	10	(+ (* (* x x) (* x x)) (+ x (* (+ (* x x) x) x)))
450	30	(+ (+ (+ (* x x) x) (* x (* x x))) (* (* x x) (* x x)))

Table 2.2 Results of GP for $x=x+x^2+x^3+x^4$

The results show that a range of functions that are equivalent to the required one are produced. The tests also identify that the number of generations and population size affect the result; if either of the two is too small a result may not be obtained.

2.6 Conclusions

[Koza et al, 1996b] detail four situations in which genetic programming has out performed human performance. That is, the genetic program has come up with a solution that is better than a human designer. This is because genetic programming can interpret and generate diverse solutions for the problem that a human would not consider.

[Banzhaf et al, 1997] identifies GP as an effective method for performing a solution search. For suitably complex domains, exhaustive search strategies of every possible solution cannot be completed within a reasonable time frame. Therefore, GP is a better alternative. Also, it states that genetic programming competes favourably with beam searches, hill climbing and simulated annealing; [O' Reilly and Oppacher, 1994] provide a comparison with these methods.

The ultimate goal of automatic programming is to tell the computer what we want and then get the solution generated automatically. Genetic programming does not provide this yet,

because GPs are designed to solve individual problems with a specific fitness test and run. For example, a GP that only generates an electronic filter circuit rather than a system that could build any circuit depending on the user's requirements.

[Banzhaf et al, 1997] states that GP may not be able to be used in many real-world situations because the fitness measure cannot be defined; this is simply because the circumstances are too difficult to describe in terms of genetic programming for a human programmer.

One of the main problems of genetic programming is that it is computationally expensive. It takes time to test and produce fitness values for each population member and then breed them over a number of generations. The GP also consumes a large amount of memory to store all of the complex structures of the population individuals.

Another problem of Genetic programming is that it is unreliable and not guaranteed to find the solution to a problem, even if one exists. However, it provides the closest result possible in terms of fitness. If there is no exact solution possible then this is invaluable.

The positive and negative issues of genetic programming identified here are returned to at the end of the dissertation.

Chapter 3 Dynamically Configurable Protocols

3.1 Motivation

Over a period of time a set of communicating processes may have a range of both functional and non-functional communication requirements. For example, initially packets must be sent reliably and in order; however, later the processes must send packets securely because private information is being sent. The non-functional quality of service requirements may also change. For example, high throughput is needed initially and later there is a demand for low jitter.

The creation of an end-to-end protocol to deal with all these differing communication requirements would introduce redundancy and complexity. Therefore, the performance of the protocol is liable to be poor, because there is the overhead of protocol processing that is not needed. [Plagemann, 1996] describes the approach of configurable protocols as an enhanced solution. The main principles of which are *decomposition* and *configuration*. Complex protocols are decomposed into fine granular building blocks each defining a single protocol task. The goal of configuration is to then combine these building blocks in such a manner that the resulting protocol configuration is as light as possible and meets the requirements of the application.

Dynamic adaptability in distributed systems has become an important issue to consider, due to the emergence of new types of applications and end systems. For example, a teleconferencing system may be able to run at very high video-frame rates and little compression over a high-speed local area network. However, when the load on the network increases it may need to reduce the video-frame rate or apply other methods to deal with the change in the network environment. Furthermore, one of the hosts in such an application may also be mobile and may move between connections providing varying levels of service. Therefore, lightweight protocols should be able to *re-configure* themselves during runtime in order to adapt to these changes, providing a method to solve this problem.

A number of frameworks have been developed allowing the configuration and dynamic re-configuration of protocols using lightweight elements. These are described in the following section: studying the early systems that drove the area of research, the systems that provide protocols for group communication and the implementations that allow the creation of protocols that meet the Quality of Service requirements of distributed multimedia applications.

3.2 Frameworks for Protocol Configuration

3.2.1 Early Systems

There were two initial systems that concentrated on the use of fine-grained protocol components to ensure that end-system protocols contained no redundancy or unnecessary complexity, therefore, increasing protocol performance. These were the STREAMs and X-Kernel systems.

STREAMS [Ritchie, 1984] was designed to allow greater flexibility in the design of UNIX device drivers and to replace the traditional rigid connections between processes and terminals. A stream is a full duplex connection between a user process and a device. It consists of several linearly connected processing modules. Modules communicate by passing messages to their neighbours through a uniform interface. Each of these modules encapsulates some part of protocol processing. A diagrammatic representation of a stream is shown in figure 3.1.

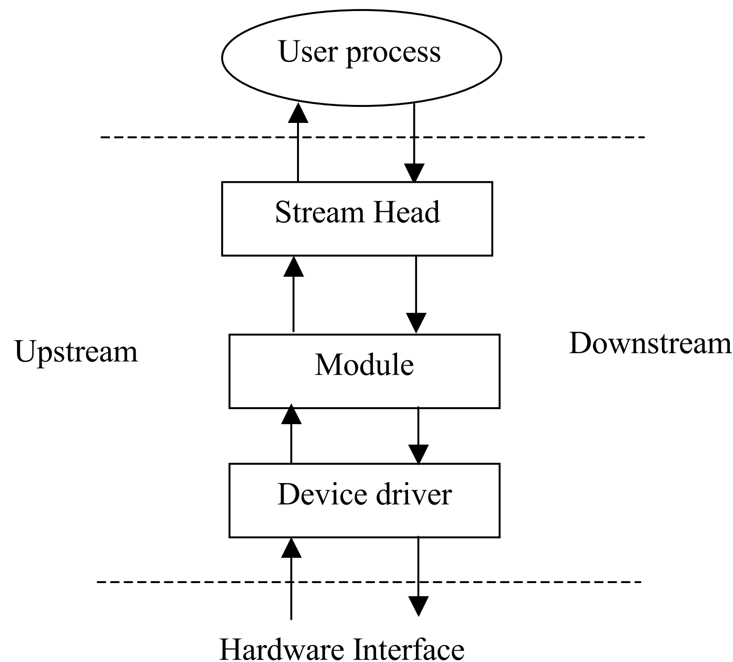

Figure 3.1 The STREAM architecture

The processing modules in a stream are thought of as a stack whose top is next to the user program. Thus to install a processing module after opening a device, an I/O push is executed on the relevant stream with the desired new processing module. By the same technique modules can be removed at run-time. It is this feature that allows the dynamic behaviour of protocols at run-time and the ability to adapt to the changing requirements of an application.

The STREAMS architecture is completely open-ended and was designed for much more than protocol implementations. Any programs that read from *stdin* and write to *stdout* can be inserted as a module into a stream, because the PIPE mechanism is implemented as a stream. The ideas brought about by this seminal paper paved the way for the dynamic protocols systems that can choose the appropriate processing parts that are needed and adapt at run-time.

The X-kernel system [O'Malley & Peterson, 1992][Hutchison, 1991] is an Operating System kernel that provides an explicit architecture for constructing network protocol stacks using a library of C functions. The protocols are implemented in terms of layers and sessions, where a layer provides some protocol functionality and a session holds the state needed for use of the layer. The layers and sessions provide a uniform interface through which messages are

delivered. This method allows the layers to be composed into directed graphs to represent the structure of the network software.

[O'Malley & Peterson, 1992] state that previous network software graphs had three important properties: the graph is simple, the nodes encapsulate complex protocol functionality and the graph is static. The X-kernel is defined as a new way of organising the protocol software as opposed to the previous method. It is designed to be a complex graph structure with nodes that encapsulate simple lightweight protocol functionality (usually an individual algorithm).

[Hiltu, 1998] added the concept of the "micro-protocol", which is a component of a layer that deals with some *individual* functionality of the layer. In the X-kernel a micro-protocol is not composed into an explicit structure, but is plugged into a layer and communicates by modifying the data structures within the layer. The properties of a layer, therefore, depend on the individual properties of each micro-protocol that has been inserted into it.

Protocol graphs are defined statically; therefore, they cannot be dynamically altered during run time. The reason for this is that the designers believed that the network architecture (not the protocol) should be dynamic to deal with any changing requirements.

3.2.2 Group communication systems at Cornell University

[Hayden, 1998] states that group communication is an approach to overcoming the problems encountered when distributed processes co-ordinate on tasks that require consistent actions to be taken by different components of the system. The protocols that are usually employed to achieve such goals are extremely complex and error prone. The task is complicated by the additional need to survive faults in different parts of the system, adapt to changes in the environment and meet the performance requirements. A central feature of group communication systems is the process of *group abstraction*, which provides operations for processes to join and leave groups and to communicate within a group.

Work at Cornell University has provided a number of group communication systems. The three described in this section use a framework of configurable protocol layers within their implementation. Horus, Ensemble and JavaGroups are the three systems.

Horus [Van Rennesse et al, 1996] provides a framework for the development of distributed applications based on group communications. The overall Horus framework provides a large collection of system and application protocols that have been developed to allow the application designer to construct a communication module that exactly meets the application requirements at minimal cost. These are composed as a stack of layers through which threads carry messages. The types of properties provided by these layers are reliability, stability and ordering.

The Horus project was originally launched as an effort to redesign the Isis group communication system [Birman et al, 1994], but evolved into a general purpose

communication architecture with advanced support for the development of robust distributed systems in settings for which Isis was unsuitable, such as applications that have special security or real-time requirements.

Ensemble [Hayden, 1998] is a group communication toolkit, based on the previous Horus development. It was created using the ML functional programming language to provide a more efficient solution that improved the performance of the protocol layers. The use of ML did this by reducing the size of the protocol layers and allowing for layers to be decomposed into finer elements.

Ensemble provides techniques to create protocols for both end-to-end communication and multi-party communication. In order to create the protocol, Ensemble provides a number of building blocks called micro-protocols (c.f. X-kernel). Each of these building blocks adds some individual protocol functionality. The micro-protocol modules can be stacked and re-stacked in a variety of ways to meet the communication demands of its applications. Table 3.1 shows an example protocol stack in Ensemble, with the functionality added by each layer described. This configuration would give the possibility of sending larger messages than is supported by the underlying transport, and will also give reliable FIFO transmission and window-based flow control.

Protocol	Description
Top	Top-most protocol layer
Frag	Fragmentation and reassembly
Pt2ptw	Point-to-point window flow control
Pt2pt	Reliable FIFO point-to-point
Bottom	Bottom-most protocol layer

Table 3.1: Example protocol stack

Adaptation in Ensemble is done transparently from the application. The lowest layers of the stack attempt to adapt first. If they cannot respond to an environment change then they pass the notification to the layer above (The application may eventually be notified). A reconfiguration is performed by the Protocol Switch Protocol (PSP). PSP is a protocol that installs the newly generated stack across all the participants.

JavaGroups [Ban, 1999] is an alternative toolkit for reliable group communication, implemented in the Java programming language. Therefore, it benefits from the portability that this language provides. Clients can join a group, send messages to all members or single members and receive messages from members in the group. Each group is identified by its name. The system tracks the current members in a group and notifies the members of joins, leaves or crashes. To join a group a process has to create a *channel* and connect to it using the group name; all channels with the same name construct a group. While connected, a client can send to and receive messages from all other members in the group. Channels are similar to BSD sockets; messages are stored in a channel until a client actively removes the next one. If no message is available, the client blocks until the next message is received.

JavaGroups provides three channel implementations: an Ensemble based channel, an iBus based channel and its own channel based on a Java protocol stack, called the JChannel. The JChannel is a protocol stack containing a number of protocol layers in a bi-directional list. All messages sent and received from the channel have to pass through the protocol stack. Each layer may modify, re-order, pass or drop a message and/or add a header to the message. An example is the fragmentation layer that breaks up a message into several smaller messages, adding a header with an ID to each message and reassembles the fragments on the receiving side.

3.2.3 Quality of Service frameworks

The growing need for multimedia applications has meant that distributed systems need to deal with the changing network circumstances to ensure that the required Quality of Service is provided to the end-system application. Two protocol configuration frameworks were designed as methods to provide the required Quality of Service to applications: Da Capo and iBus.

Da Capo [Plagemann, 1999] is a framework for dynamic protocol configuration with the goal of creating a run-time protocol for each connection. It was created as an alternative to traditional end system protocols, which were not able to support the quality of service requirements of current and emerging multimedia applications. Da Capo was designed to allow protocol configuration that combines fine granular software and hardware building blocks at run-time. Its aim is to provide an optimised protocol between application and network services automatically at run-time and at the same time be as light as possible.

Da Capo uses one process per protocol and is split into three different layers: A, C & T. Layer T is the transport layer i.e. it represents the existing and connected communications infrastructure. Layer C adds functionality to the Transport layer and layer A corresponds to the distributed applications that access the services of the underlying communication layers. The A-C layers specify which type of services are needed and quantify the Quality of Service required by the application. Application requirements are specified in the form of tuples on single attributes types such as delay, jitter, packet loss etc. This model of this framework is shown in figure 3.2

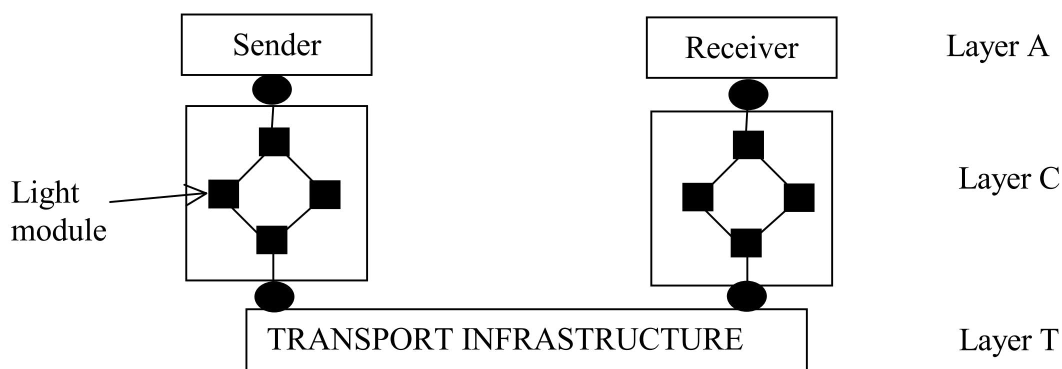

Figure 3.2 The Da Capo framework

Layer C selects a protocol configuration that fulfils the requirements during the establishment of the connection. Layer C services are decomposed into a set of protocol functions. Each protocol function encapsulates a typical protocol task e.g. error control, flow control, encryption etc. The configuration is lightweight in that layer C uses fine granular building called *light modules* to implement the protocol layers.

Applications specify their QoS requirements in the form of an objective function. This specification is forwarded to the Configuration and Resource Allocation (CoRA) [Plagemann, 1994] component. CoRA selects the most appropriate modules from a functional and resource use point of view and ensures that sufficient resources are available.

The iBus [Softwired, 1999] system from Softwired AG is a commercially available Java Messaging Middleware system based on the publish/subscribe communication paradigm. IBus resides between the operating system and the application software. It allows the efficient asynchronous transfer of messages or Java objects between communicating parties.

The bus system is unlike an Object Request Broker used in classic client-server applications. In most cases ORBs work with synchronous request-reply patterns and provide an "Information pull model". IBus operates through an "information push" between producers and consumers of information. [Kruthoff, 1999] identifies the three main components of the iBus system as:

- Producer: a component that feeds information (Java objects) into a channel. A producer first registers with a channel before publishing its information
- Consumer: a component that receives information from a channel: A consumer subscribes to a channel to receive from one or more producers.
- Channel: a component that moves information between the producer and consumer components.

An iBus channelURL specifies the channel. It is made up of a QoS String and a destination address. The QoS String defines a Quality of Service for the channel. It is possible to use reliable or unreliable IP multicast, TCP, HTTP or wireless protocols. Depending on the data being transmitted the QoS can be adjusted by the developer e.g. if low jitter is required in the case of video data, a less complex QoS would be defined.

A stack of "Protocol Objects" represents the QoS. Each Protocol object describes one part of the iBus protocol stack. The stack for reliable IP multicast looks like this:

- DISPATCH : PULL : FRAG : NAK : REACH : IPMCAST

If a programmer wants to add further QoS requirements such as compression, he would have to define and implement a compression Protocol Object first. There is a Java class offered to simplify this process.

Protocol stacks are dynamically created depending of the information provided in the ChannelURL. There is no concept of insertion and deletion of individual protocol objects

during run time. The system only deals with adaptation by selecting the appropriate linear chain of objects depending on the required QoS of the channel.

3.2.4 An alternative approach

The FlexiNet platform [Hayton et al, 1999] is a Java middleware platform that provides a component-based approach to distributed application development. It places strong emphasis on the use of reflection within the protocol stacks. There are four key elements of the FlexiNet architecture: software components, transparent component binding, policy definition, and automated deployment. The architecture and methods of this system provide a different way of looking at the problem the previously described systems are attempting to solve.

In order to program with components, there must be a model of binding between components, so that a programmer may locate one from another. Components may pass references to other components between each other in a transparent way. In these circumstances FlexiNet associates the implicit binding request with the relevant policies and ensures that the constructed binding respects the policies. A reflective protocol stack is related to the binding to carry out the call process. FlexiNet provides a layered protocol stack, in which the layers can be viewed as reflective meta-objects that manipulate an invocation using Java Core Reflection [Sun, 1999].

The use of reflexive layers is an alternate method from the previously described systems. Reflection allows the component to have an open implementation. Depending on what is required, the component can adapt itself by adding or removing sub-components that provide a degree of functionality. This means that rather than altering a stack of micro-protocols, the more complex layers of the FlexiNet architecture adapt themselves to changes in the environment.

3.3 Choice of Toolkit

The following features were identified as the most important in the choice of a communication toolkit.

- Interoperability– the ability of the genetic programming toolkit to communicate and work with the communication applications.
- Protocol layers – the availability of a wide range of building blocks that will provide interesting protocols.
- Implementation language – How easy it is for the programmer of the system to use and implement applications that feature the toolkit.

The early systems of STREAMS and X-kernel are discounted because they do not provide an interesting set of building blocks, rather a model for implementing user modules. The iBus and FlexiNet, also suffer in that they do not have a set of purpose built protocol layers that can be used by the Genetic Program. This makes them ineffectual to the genetic program,

which cannot work without the individual elements. The implementation of a set of modules is not feasible in the time available.

The group communication systems from Cornell, however, all have a set of layers that are used that can be put together to create group protocols. Their interoperability with the GP and the implementation language determine the decision between the three. Ensemble suffers from the ML language implementation, which as a functional language makes it difficult to create applications in. Also, the communication between a Java genetic program and an ML group communication application is difficult to implement.

The JavaGroups toolkits meets all three requirements and therefore, was chosen to be used for protocol configuration within the genetic program. It has a set of twenty protocol modules that can be used to create group communication protocols. It is implemented in Java, a programming language that the implementer is experienced with, making application development an easier process. The use of Java also makes the communication between GP and protocol application simple through the technique of RMI (all of the GP can be implemented using Java).

3.4 JavaGroups Architecture and Implementation details

This section examines in detail the architecture of the JavaGroups system and what is required to implement a group based application using its features.

In order to create the simple JavaGroups applications that will be used in the genetic program the following steps must be carried out.

- Each member must create a JChannel using the **JChannel()** constructer, providing the protocol stack as an argument (described later).
- The member must then connect the channel to the group it wishes to participate in using the **Connect()** method, providing the group name as an argument.
- Once connected the process can send and receive messages to/from the channel using the **Send()** and **Receive()** methods. A message can be unicast to an individual if its address is provided; otherwise it is multicast to the entire group. The receive command detects messages and view changes. A view change indicates that a member has joined, left or crashed.
- When a member wishes to leave a group, the **Disconnect()** method removes the channel from the group and the **Close()** method destroys the created channel.

These techniques show that it is possible to quickly create applications for group communication using the JavaGroups toolkits. However, the complexity in application design lies in the choice of the protocol stack that is passed as an argument in the creation of a channel. There are a number of layers available and the stack must be selected from these and then placed in the correct order. The available layers are listed in table 3.2.

Using the layers in table 3.2 a protocol in constructed as a stack of these modules on top of one another. The following are some example protocol stacks that can be used for group communication applications:

- Reliable, ordered group multicast communication:
  UDP: PING: FD: STABLE: NAKACK: UNICAST: FRAG: FLUSH: GMS: VIEWENFORCER: TOTAL
- Reliable Multicast:
  UDP: NAKACK: UNICAST: FLUSH: GMS

Protocol Name	Description
DELAY	Delays incoming and outgoing messages by a number of milliseconds.
DISCARD	Drops a random percentage of incoming or outgoing messages.
FD	Failure detection based on a simple heartbeat protocol. It regularly polls group members for liveness.
FD_RAND	Failure detection by testing random group members for liveness.
FLUSH	Flush all pending messages out of the system.
FRAG	Fragments messages larger than a given size into smaller packets.
GMS	Group membership protocol. Handles joins, leaves and crashes and emits new views accordingly.
MERGE	Simple merge protocol. Periodically multicasts a hello message with group address. When received by member of same group but not in group the merge is performed.
NAKACK	Negative acknowledgements paired with positive ACKS.
PIGGYBACK	Combine multiple messages into a single large one.
PING	Responsible for finding the initial membership of a group,
QUEUE	Layer that queues messages.
STABLE	Computes the broadcast messages that have been received by all members
TOTAL	Sequencer based total ordering protocol layer.
TUNNEL	Uses TCP connection to route messages. An alternative to UDP as bottom layer.
UDP	Bottom-most layer of stack that transmits packets using the User Datagram Protocol.
UNICAST	Reliable unicast layer
UNIFORM	A message is delivered by all members if it is delivered by at least one member. Dynamically uniform failure atomic group multicast.
VIEWENFORCER	Discards all messages until client becomes member of a group

Table 3.2 Description of the available protocol modules in the JavaGroups toolkit.

To illustrate clearly how applications are created using the properties of the JavaGroups toolkit previously, described figure 3.3 contains the code for a simple send and receive to a group.

```
import java.util.*;
import JavaGroups.*;

//Simple connect, send 50 messages to group then wait for a different 50 to receive

public class ChannelTest {
private Channel channel=null;

String props="UDP:PING:FD:STABLE:NAKACK:FLUSH:GMS:TOTAL";

public Start() throws Exception {

        channel=new JChannel(props);
        channel.Connect("ExampleGroup");

        for(int i=0; i < 50; i++) {
                System.out.println("Casting msg #" + i);
                channel.Send(new Message(null, null, new String("Msg #" +
                                                              i).getBytes()));
        }

        while(m<50) {
             try {
                 obj=channel.Receive(0); // no timeout
                 if(obj instanceof View)
                     System.out.println("--> NEW VIEW: " + obj);
                 else if(obj instanceof Message) {
                     msg=(Message)obj;
                          m++;
                     System.out.println("Received " + new String(msg.GetBuffer()));
                 }
             }
             catch(ChannelNotConnected conn) {
                 break;
             }
         }

        channel.Disconnect();
        channel.Close();
}
```

Figure 3.3 The code to implement a simple sending a receiving group member using the JavaGroups toolkit

3.5 Conclusions

The use of lightweight components for the creation of a protocol is a possible method to provide improved performance over the use of complex protocol modules that deal with a range of protocol functions. This is because the protocol stack can be geared to perform only the protocol functions that are needed for the particular communication. There is no redundant functionality that leads to an increased overhead in the protocol processing. The method of structuring protocols from fine-grained elements, therefore, leads to an improved performance in the protocol.

The second advantage of configurable protocols is the flexibility they provide. A number of the systems described in this section have the ability to adapt the protocol stack during run time and, therefore, during the length of a communication. It is possible to change the protocol to meet new requirements of the application or respond to changes in the network environment ensuring that the required service is still provided. This is done using the "lightest" protocol stack possible, guaranteeing that the best communication performance is provided.

The use of protocol configuration is especially suited to the provision of quality of service requirements of distributed multimedia applications. [Plagemann, 1996] states this is because it is the most flexible method and is able to support the range of realistic QoS requirements. It also allows new quality of service requirements to be met by integrating new modules into the system.

[Hayden, 1998] cites one possible problem of configurable protocol stacks is that the drive for flexibility may not meet the needs for high performance protocols. Increasing flexibility increases the modularity of the system and this in turn prevents low-level optimisations being made to the protocol. Therefore, the protocols are not providing the maximum possible performance.

The examination of how to construct applications using the JavaGroups toolkit has shown that this is a non-trivial task. The user must be experienced in protocols to understand the features that are provided by each of the separate layers. They must then design the appropriate layout of the stack that will meet the requirements of the communication. Without experience this is a very difficult task and therefore, is suitable to the application of automatic programming. Without the need for expertise in use of the toolkit it should be possible to state the protocol requirements you need and have the correct stack layout returned to you.

Chapter 4 Application of Genetic Programming to Protocol Configuration

4.1 Introduction

This section examines the design of the framework of the system that will automatically generate the JavaGroups protocol stack that meets the requirements for a given communication type. The design of this system involves applying the techniques and theory of genetic programming to the domain of configurable protocols. This can be split into three distinct problems to overcome:

- The representation of stacks within a genetic program.
- The evolution and creation of new stacks.
- The testing of a stack to provide it with a fitness measure and then in turn the selection of fitter stacks for further evolution.

The following section examines how the principles are applied to the domain to solve these three problems and how they were implemented using a combination of the GPsys and JavaGroups toolkits to create the basic genetic programming system that is extended later in future experiments.

4.2 Key Design choices

4.2.1 Representing a stack within the Genetic Program

For communication to take place using the JavaGroups toolkit, each sending and receiving member of a group must subscribe to a channel called the JChannel. When creating a JChannel, the properties of the underlying protocol stack can be specified as an argument. For example, the property specification may state a reliable UDP based channel and another may detail a virtually synchronous, total order channel.

The argument passed to the channel to define its properties is called the *property string* and it has the following format:

- "<prop1>(arg1=val1):<prop2>(arg1=val1;arg2=val2):<prop3>:<prop n>"

It consists of a number of properties separated by colons. Each property relates directly to one of the protocol layers available from the toolkit. The first property becomes the bottom-most layer, the second is placed on top of the first etc; the stack is created from the bottom to the top as the string is parsed from left to right. Each layer may have 0 or more arguments passed as name/value pairs to set any properties required by the layer.

Therefore, if the genetic program is to generate the protocol configuration for a channel to meet a set of requirements then it must generate the correct property string, before passing it to a JavaGroups application so that a JChannel can be created for communication.

The property string is a linear structure; it consists of a chain of independent building blocks. Chapter 2 discussed linear and tree representations of members of genetic programming

populations. Both of these are possible methods for representing the property string. The linear method is the most natural technique; replacing the linear chain of program instructions by a linear chain of protocol modules performs this. The alternative is to create a tree structure that represents each stack, simulating the linear nature that is required. In order to implement the linear method, a complete GP algorithm would need to be designed and implemented to deal with the variances of linear evolution and selection. The constraints of time mean that it is not feasible to implement a specialised genetic programming solution of this type. All currently available toolkits are based on tree-based solutions; therefore, the choice taken was to *simulate the linear nature of the property stack using a tree*.

The next step of the GP structure design is to identify the functions and terminals that will be used to build the tree for each population member. The terminal set consists of the names of the protocol layers; this is because, like program terminals, they are the basic elements of the string. There is only one function needed in the representation of strings. It is a simple function that takes two arguments and appends them together (named (:) to help picture the string). It is this introduced function that will simulate the linear nature in the genetic program. The diagram in figure 4.1 shows an example of a stack in a tree structure.

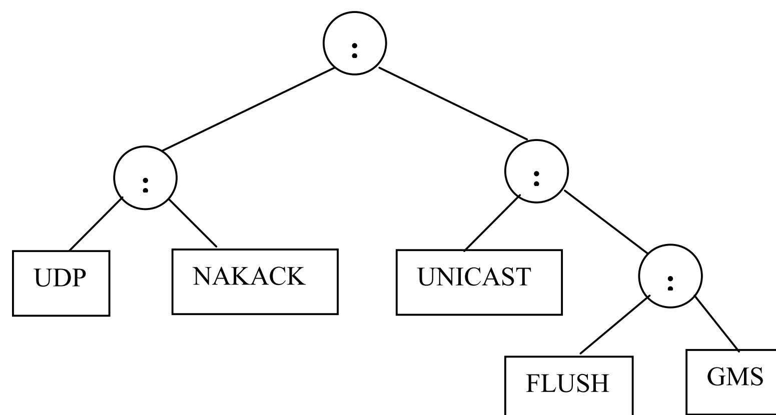

Figure 4.1 Tree representation of UDP: NAKACK: UNICAST: FLUSH: GMS

The only problem with this method is that different members of a population may represent identical stacks. This is because differently shaped trees can hold the same protocol layers (e.g. UDP: FLUSH: GMS can appear more than once, because two trees can create it). This is not a critical flaw, because in a small size population the probability of repeated structures is small. Also, Genetic Programming deals with evolving strong members of the population, therefore, similar strong stacks may be evolved differently or if unfit both removed.

There are three steps in the toolkit implementation of this method:

- Create a terminal class for each of the protocol layer. That is for each module implement a java class that will provide the name (and properties) for the terminals of the tree. For example UDP.java and FLUSH.java
- Create a function class for the append operation, which takes two terminals and appends them together. Implemented in STACK.java

- Set the function set to hold the one STACK function and set the terminal set to hold the set of protocol modules needed to build the required protocol.
- Set the GP parameters: Population size, Number of generations, grow method, crossover probability, mutation probability, tournament size.

Using these classes and method the GPsys toolkit will generate an initial population of stacks and perform the selection and evolution of new generations of stacks until a solution is returned.

4.2.2 Genetic Operators acting on the Property stack

It is important to next consider what effect the genetic operators have on the protocol stack. Therefore, the three operators (crossover, mutation and reproduction) identified in chapter 2 are discussed to ensure that the stacks are suitably evolved for future generations in line with the properties of genetic programming.

The desired effect for crossover evolution on protocol stacks is *identical to the linear crossover method*, described in section 2.2.3. That it, a subset of layers from one parent should be swapped with a linear segment of the other parent to create two new children for the next generation. The use of tree crossover by the toolkit, however, has an identical effect; this is due to the structure of the tree defining a stack. Each sub tree within the tree, identifies a smaller part of the stack {example UDP: FLUSH}; therefore, when sub-trees are switched in the tree crossover method of the GPsys toolkit, it is still linear subsections that are being swapped.

Linear mutation requires the alteration to occur within the individual elements of the chain i.e. the mutation of separate instructions. However, in the case of protocol stacks the individual layers are fine-grained building blocks that cannot be altered. The desired effect of mutation is to introduce random changes to a fit individual to identify what effect this has in the evolution. Therefore, the way to introduce mutation into protocol stacks is *to mutate sets of layers*. Using the tree mutation operator provided by the GPsys toolkit performs this. The use of sub-trees to represent sub-sections of the stack's layers means that a random tree of new layers can be inserted in place of a removed set of layers.

The reproduction operator, that simply copies a fit individual to the next generation without change, is identical for both linear and tree structures. Therefore, the reproduction method of the toolkit can be used without change.

4.2.3 Fitness Functions

In order for the best stacks to be selected from a population for evolution, the concept of how to determine what represents a fit stack compared to another one must be defined. There are three different measures that can be taken for how fit one individual stack is. These are a test

for *semantic viability*, a test to check that the *stack meets the required communication properties* and a test of the *quality of service provided by the protocol*.

A JavaGroups protocol stack can be made up of any protocol layers fitted together in any order and still be syntactically correct. That is, when a stack is passed to a channel it cannot cause a syntax error to occur. However, due to the obvious number of possible combination of layers, there are a number of stacks that do not perform any meaningful operation. For example, any stack with repeating layers will not allow messages to be sent to the channel. JavaGroups provides exceptions if any of these stacks cannot communicate on the channel. Therefore, a fitness measure of stacks can determine between stacks that are meaningful and those that aren't.

A second type of fitness test is a measure of how well the tested stack meets the requirements of the protocol being searched for. This means that a channel must be created and the stack passed to it before tests for reliability, group communication, quality of service and ordering can occur. The final type of fitness test measures how well the non-functional requirements of a communication are met. Therefore, performance measures like throughput and delay are calculated and transformed to a fitness value.

The fitness tests produce a *standardised fitness measure*. That is, a meaningful stack that fully meets all of the communication requirements has a fitness value of zero. While, a stack that has no sensible operation is given the highest value. The range between represents the amount it partially meets the communication requirements. The following section describes the design and implementation of a test harness that carries out this testing.

4.3 Design & Implementation of Fitness Tests

For every stack that is generated by the genetic program, a JChannel must be created in order for it to be tested for semantic and performance properties. There are two possible methods for controlling these tests:

- Implement the creation of the channel and sending/receiving process from within the fitness test of the genetic program. That is, the code for fitness test consists of the method calls to the JavaGroups toolkit to create the channel and sender and receiving applications to simulate the tests. The separate send and receive processes are created using exec() calls.
- Create a distributed system that separates the genetic program from the JavaGroups applications. That is, a stack is passed to the separate process (maybe on a different machine), which controls the tests using JChannels, and then calculates a fitness measure that is then returned to the genetic program.

The first method has the advantage of being the simplest to implement; there is no complexity in the implementation of the fitness tests. However, genetic programming is extremely computationally intensive and the addition of complex fitness tests that also require large amounts of CPU time and memory mean that this method is infeasible, without powerful computer performance. The use of exec processes for the separate communication

processes is a messy method because communication between the GP process and these must use shared memory, which is difficult to control for a large amount of processes. Performing these tasks run on a standard workstation will quickly exhaust resources.

The second method of distribution overcomes this problem. The genetic program process can be implemented on one machine, while the testing processes may run on another. This allows a greater pool of resources to be shared, because the system can execute over a number of computers rather than an individual machine. The system is also extensible; new processes for sending and receiving parties can be easily added on new machines, without worrying that they may use up valuable resources. Further advantages include the improved performance due to concurrent execution of the software. Due to the number of advantages this method provides, it was chosen over the exec solution.

The distribution was achieved using *Java RMI*. This method was chosen over alternative distributed system solutions, such as Corba, because Java is already the principle implementation tool being used (both toolkits are Java based) and therefore, no extra software is required for the creation of the new system.

The model in figure 4.3 shows the design of the core of the distributed system that will form the basis of the overall genetic program.

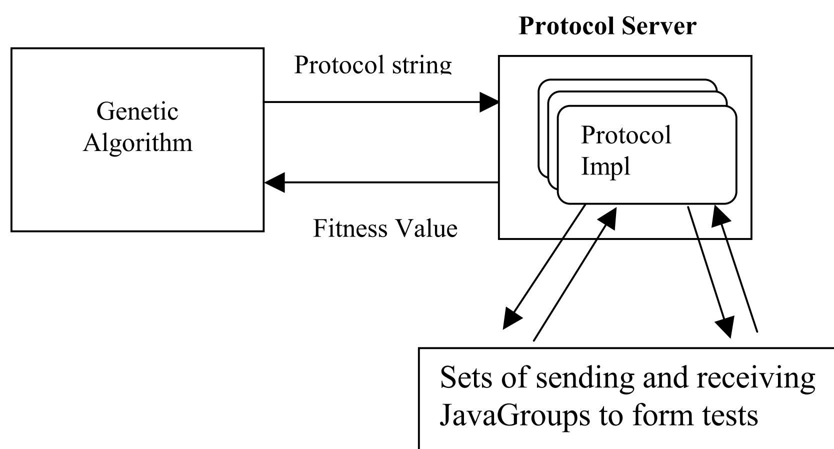

Figure 4.2 Core section of genetic programming system

The genetic programming algorithm generates a population of protocol stacks and then calls the fitness function to provide a value that will determine if it is selected.

The fitness function carries out the following process:

1. The creation of a new ProtocolImpl object on the protocol server to test the stack.
2. Remotely call one of the test methods available from the ProtocolImpl object, passing the protocol property sting as the parameter. For example, there is a method to test for reliable point-to-point communication and there is also a method to test for reliable multicast etc.

3. Block waiting for the return of a value from the test harness representing the fitness measure.

Once the fitness values have been obtained from all members of a population the genetic program crates a new generation of stacks by the process of evolution and the process is repeated for the required number of generations.

The basic system has been designed to allow extension in the addition of tests controlled by the protocol server. Future experiments described later in the report will use this provided architecture as the basis of their genetic program. For example, a sending server can provide a process for sending messages and a receiving server for providing a process for receiving process. The ProtocolImpl object remotely calls the methods to set up tests simulating the receiving of messages for a given protocol and obtains a fitness measure from these tests before returning this value back to the fitness function of the genetic process.

4.4 Conclusions

This chapter has identified methods to allow GP to be fitted to the domain of protocol configuration. This consists of a number of basic mechanisms of the GP system to provide the basis for further experiments later.

- The simulation of a linear stack using the tree methods of a genetic programming toolkit.
- The use of a distributed system to separate the genetic processes from the protocol tests. This has the advantages of:
    - The ability to use a pool of resources to deal with the computationally expensive processes involved.
    - Ease of simulating the distance between the processes within group communication e.g. a sender and receiver on separate machines.
    - New tests can be created and added easily.

The most interesting part of this system occurs in the design of fitness tests and the design of what it means to measure the fitness of a stack in terms of communication requirements. The test harness allows tests to be added, removed or changed to provide variety in the protocol stacks that are being searched for. For example, a set of fitness tests can be designed, implemented and added to the system to test for one type of protocol functionality. This can then be redesigned to deal with the needs of a different communication type. Therefore, the next stage is to examine how to design the tests for one simple communication type: reliable point-to-point communication.

Chapter 5 A First Experiment: Reliable Point-to-Point Communication

5.1 Introduction

The simplest form of communication that can take place using the JavaGroups toolkit is point-to-point communication. That is, an individual process communicates directly with another individual by sending and receiving messages. In this context it is possible to create a protocol stack that provides reliable communication, i.e. no messages are lost and the receiver is guaranteed to receive all of the messages sent. The main goal of this first experiment is to successfully generate a protocol for this straightforward task that can be easily created by a human designer, assessing the previously described framework. The design, implementation and testing of a genetic programming system to perform this task is presented in this chapter.

5.2 Design of the Genetic Program

5.2.1 Introduction

The design of the genetic program follows the description in chapter 4. The two main parts are for creating stacks and for measuring the fitness of the stack. The method for generating stacks follows closely the description given previously. The important refinement is the choice of the terminal set. The fitness test for this case is an extended mechanism of the basic test harness to cope with point-to-point communication and provides a fitness measure of a stack's ability to provide reliable unicast. The following two sections describe the design of these two main elements of the system

5.2.2 Generation of Stacks

The most important design decision in the creation of the genetic program to generate and evolve the protocol stacks is in the choice of the terminal set. The set must contain at least the layers that are required to generate a protocol for reliable unicast communication; however, the more layers that are introduced will affect the performance of the GP. Extra layers do provide the possibility of different implementations, but the GP will take longer to find a solution because the search space is increased and there is the increased probability that a solution may not be found. The following is the set chosen:

- {UDP, NAKACK, MNAK, UNICAST, GMS, FLUSH}

UDP is included because a transport layer is needed; without it no transmission would be possible. The three types of reliability layer are included; NAKACK, UNICAST and MNAK provide methods for ensuring that the messages are not lost. GMS and FLUSH are included because they provide mechanisms for the control of groups (point to point can be considered as a special case of the group communication model). The remaining layers that deal with extended group control features and virtual synchrony properties are excluded because they are not needed for point-to-point transmission and will only introduce unwanted protocol overhead.

The remaining genetic programming parameters are chosen as follows:

- Function set {:}. Simply holds the function for stacking layers on top of each other

- Tree grow method – *Grow*. Grow provides the method for creating trees of different depths instead of all at same depth; this is suitable, because the GP must examine different lengths of stacks (not all the same).
- Crossover probability – *70%,* Mutation probability *–30%.* A higher probability is chosen for crossover than mutation because evolution by crossover is more desired as it retains the properties of fit individuals as opposed to the random introduction of new elements.

5.2.3 Fitness Function and Tests

The key design decision in the design of the fitness function lies in the measure of what is a fit stack. The program is searching for a stack that reliably transmits messages from one process to another; therefore, a stack that is semantically correct and can transmit and receive 50 messages under a normal loss less environment and also 50 messages over an environment that drops packets. The following is the standardised fitness measure of the stack, chosen for use by the fitness function.

- The fitness lies in the *range of 0 to 100*. If a stack transmits and receives 100 messages it has a fitness of 0. If it receives 0 then it has a fitness of 100.

This is as simple a measure as possible, so not to overly intricate the design of this experiment. Possible other tests include a measure of the performance of the stack in terms of delay and throughput. However, due to this being an initial experiment these are not included so that results are not affected by these extra parameters, ensuring that GP can create a solution. However, these properties are included in the extended experiment described in the next chapter.

In order for the fitness measure to be applied to each of the generated population a point-to-point application must be generated. Figure 5.1 is the complete design of the test harness for unicast communication. The core system of a genetic algorithm and protocol server described in chapter 4 is extended by the addition of a sending server and a receiving server.

The ProtocolImpl has two methods for use in this experiment:

- PtTest (String props) – Method for controlling a send of 50 messages from one point to a receiver in a normal environment.
- DiscardTest (String props) - Method for controlling a send of 50 messages from one point to a receiver in a lossy environment.

The creation of a lossy environment is produced using the *insertion of the DISCARD layer* into the protocol stack. It simply has the property of randomly dropping messages by not passing them to the next layer. It must occur above the UDP layer, otherwise no messages will be dropped. Therefore, in the discard test the property string is parsed and “UDP:” is replaced by “UDP: DISCARD”.

Using the methods and RMI structure described previously, the testing of one stack that is repeated for all members of the population is carried out using the following steps:

1. Fitness function called for the individual stack in the genetic program.
2. Fitness function calls the ProtocolImpl method PtTest() passing the generated property string as a parameter.
3. PtTest() remotely calls the Receive() process of the ReceiveImpl object passing the property string as the argument.
4. The Receive() method creates a JChannel using the stack and then connects to it.
5. The Receive() method remotely calls the Send() process of the SendingImpl object and then blocks waiting to receive messages on the channel.
6. The Send() method creates a JChannel and connects to the same group as the receiver, before sending 50 messages to the receiving process.
7. The receive() method counts the number of messages received and then return this value to the fitness function.
8. If the value is greater than 20 (ie. It can send messages then the test in a lossy environment is carried out)
9. The DiscardTest() of the ProtocolImpl object is remotely called by the fitness function. It carries out steps 3-8 with the changed protocol stack. The number of messages received is returned to the fitness function.
10. The two returned values are used to create a fitness measure between 0 and 100.

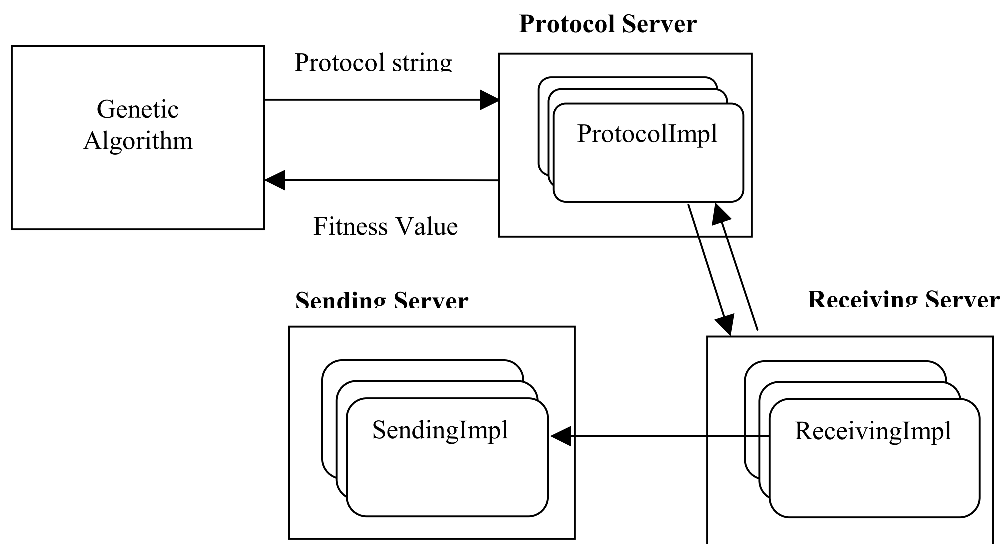

Figure 5.1 Genetic Programming system for generating a reliable point-to-point protocol stack

5.3 Results of the Experiment

The system described in section 5.1 was implemented as a series of Java programs using the GPsys and JavaGroups methods; these are found in the supporting documentation.

In order to test the system the four separate processes were run on two individual NT workstations, each with 128Mb of memory (to share the resources ensuring the program does

not crash). The genetic algorithm and sending server were initialised on one machine and the Protocol server and Receiving server were started on the other machine.

Due to memory restrictions the maximum population size was 500 members (This was found by experimentation). The system was tested using a range of different population sizes and number of generations. The results from these tests are shown in table 5.1

Population Size	Number of Generations	Generation completed in	Result	Time (min)	Fitness
10	3	3	UNICAST	3	100
10	3	3	PING	4	100
10	5	5	MNAK:UNICAST	1	100
10	5	5	PING	8	100
10	10	2	NAKACK:UDP:MNAK:PING	7	0
10	10	10	NAKACK:PING	13	100
50	3	3	NAKACK:UDP	55	15
50	5	2	UDP:MNAK	40	0
250	5	3	UDP:MNAK	>120	0
250	5	2	UDP:NAKACK:UNICAST:FLUSH:GMS	100	0
250	5	5	UNICAST:NAKACK:UDP	>120	13
500	5	2	UDP:UNICAST:MNAK	>120	0
500	5	2	UDP:UNICAST	>120	0

Table 5.1 Testing results for reliable, unicast communication

5.3 Analysis of Results

The results in table 5.3 show the following:

- A range of different protocols all providing reliable unicast are generated by the genetic program. The same result is not always returned.
- The correct result is not always provided. A completely unfit solution is provided in the low population sizes. On two occasions a stack that sends, but not reliably is produced (fitness cases 15 and 13)
- The population size affects the performance of the GP. The tests using a population of ten members only provided a solution once. However, as the population size was increased to 500 the probability of failure reduced.
- The number of iterations used did not have as great an effect on the result as the population size. Increasing the number of generations for a small population size did not affect the chance of a correct solution. The use of a larger population size provided a solution in an earlier generation.
- A large amount of time is needed to find a solution. To find a solution quickly a small population can be used, but the probability of finding a solution is low.
- Similar patterns occur in the results. It can be seen that the reliable layers are placed above UDP (e.g. UDP:MNAK, UDP:NAKACK, UDP:UNICAST).
- The less complex protocols e.g. UDP:UNICAST or UDP:MNAK are produced more often. However, the more complex (but still correct) solution of

UDP:NAKACK:UNICAST:FLUSH:GMS is produced once. This is due to the large population size generating this stack early in the process, rather than it being evolved to.

5.4 Conclusions

The results from these experiments have identified that the domain of configurable protocols can be related to genetic programming. This has shown to be the case by generating stacks that allow reliable point-to-point communication from learning and evolving stacks.

The results from the experiments using the genetic program have shown that different stacks are generated that provide the same solution. These need to be *differentiated* between; for example they will provide different quality of service features. The next chapter examines the use of QoS test to select between protocols.

The use of a distributed solution provided a number of benefits. The creation of the sender and receiver application over a local network was easy to implement. The genetic program was spread over more than one workstation; therefore, improving performance and preventing crashes due to lack of resources.

It can also be seen that the genetic program is not reliable; there is a probability that it will not provide a solution with zero fitness. The population size relates to this probability because a low population size means a high probability of failure. The GP is also time consuming; again this is dependent on the population size and generations. If they are decreased the time taken to find a solution reduces. There is therefore a balance between time and reliability.

Chapter 6 An advanced experiment: Reliable, Ordered Multicast communication

6.1 Introduction

The initial experiment showed in a simple case that genetic programming could provide a protocol stack to meet the requirements of a communication. However, JavaGroups provides much greater functionality than simple reliable point-to-point communication. It allows the ability for protocol stacks to provide reliable and ordered multicast communication.

The exchange of single messages is not the best method for communication within a group of processes. A *multicast message* is more appropriate; this is a message that is sent by one process to all member processes of the group. Reliable multicast has the property that if a message is sent then every member is guaranteed to receive it. Ordered group multicast occurs when every member sees the sent messages in the same order. Reliable, ordered multicast must ensure that both these properties are met within group communication.

This section describes the design of a genetic program to create possible protocol stacks to meet the properties of reliable, ordered multicast. This problem is more complex than the initial experiment; more protocol layers are considered and more tests are carried out on the stacks. However, the basic system architecture is still used and the design follows closely that of chapter 5. Having identified in the last section that QoS requirements should also be considered in the differentiation between fit stacks, the design of this type of test is also introduced.

6.2 Design of the Genetic Program

As described previously the most important decision in setting the parameters of the GP run is in the choice of the terminal set. This experiment considers a greater number of communication properties than reliable unicast, so the set is extended to provide layers to meet these new requirements. The chosen set is as follows:

- **Terminal set** {UDP, NAKACK, UNICAST, GMS, FLUSH, FD, FD_RAND, MACK, MNAK, PING, STABLE, VIEW_ENFORCER, TOTAL}

The UDP layer is included as the transport layer, without which no message transmission can occur. The NAKACK, UNICAST, MACK & MNAK all provide reliability mechanisms, ensuring that lost messages are retransmitted. The inclusion of both multicast and standard acknowledgement layer types investigates if there is an advantage between these two types for group communication. GMS, PING, FLUSH, FD & FD_RAND provide the properties for controlling the elements within a group, ensuring that all members know about joins, leaves and crashes and can respond to them. Finally, the TOTAL layer is included to provide the total ordering of messages within a group. This is a sequencer-based method, however there are other algorithms that can be used to provide ordering layers but are not included in the JavaGroups implementation.

The remaining parameters of the GP run are identical to those of the previous experiment. This is because the method correctly and efficiently chose stacks to be presented to the tests and evolved them in a manner that provided a solution. These parameters are described as follows

- Function set {:}
- Crossover probability – 70%
- Mutation probability – 30%
- Grow method – GROW

The design of the fitness measure is important in ensuring that the stack meets the requirements of the communication, but also allows for a preference between one stack and another depending on the service it provides. The choice of measures are described as follows:

- The stack's ability to send and receive messages.
- The stack's ability to send and receive messages reliably.
- The ability to deal with group changes.
- The ability to send and receive message in order.
- The performance of the stack in terms of throughput.

The first measure takes into account that a stack that is able to send messages under a normal environment is fitter than one that cannot send any; if it does not meet any of the requirements it still provides layers that must be included. The second measures identifies that a stack is fitter if it provides reliable transmission of messages. The third measure identifies whether a stack has the properties to deal with members joining and leaving a group. A fitter stack is therefore, one that does not send to members that have left and sends to new members. The order measure ensures that a stack is fitter if all the members see the messages in the same order, providing they have been jumbled in some manner. Finally, a performance measure on the stack ensures that those stacks that provide a better quality of service are deemed fitter.

The weighting of these measures is an important design decision. The weighting of fitness deals with identifying which of the measures is more important in selecting the protocol properties. For example, if reliability is measured between 0 and 100, while ordering between 0 and 50; the ordering is weighted less therefore, is not as important.

The overall standardised fitness of each stack is a measure between 0 and 400 with 0 representing the fittest stack and four hundred a stack that meets none of the requirements. The most important needs of the protocol are that of reliability, ordering and group communication. These are therefore given higher weightings. QoS requirements are not deemed as important at this point, because the functionality requirements must be provided first before performance differentiates. However, the concluding section of this report examines why this may not always be the case. The splitting of the measure is given below.

- Communication 0-50, Reliability, 0-100, Group communication 0-100, Ordering 0-100, Quality of Service 0-50

In order for a stack to be given a measure of this type, it must have five tests performed using it. The following section describes the architecture of the genetic programming for carrying out these tests.

6.3 System Architecture

As mentioned above, the architecture of the genetic programming system is similar to that presented in the previous chapter. It is based on a number of distributed servers and processes communicating using RMI to perform a set of tests on the stacks generated by a genetic algorithm. The difference lies in the servers that perform the tests for each of the stacks. Previously, there was one sending server for creating processes to send messages and another to create processes to receive messages.

All of the tests involve groups of processes. This means that there must be a sender process to more than one receiver process. Figure 6.1 shows the layout of this new architecture. The Protocol server controls the creation of a set of 1 to n multicast servers (these provide the distribution of receivers) modelling a set of n group members, on which can be created a process for receiving multicast messages. Each of the group member objects communicates with the ProtocolImpl object by returning a value that measures the stack's performance. Like in the previous experiment there is a single process for casting messages to the remainder of the group. There is no need for the sender to be controlled by the protocol server because it provides no fitness measure. Therefore, a sender process is simply started by the first receiving member remotely calling it to send to the set of members.

Table 6.1 provides a description of the objects and methods related to the distributed genetic programming system described above. It outlines the elements that will be used for the five tests on each stack. Section 6.4 describes the design of each test and implementation in terms of these available methods.

6.4 Protocol Tests

For each of the five individual measures described previously, an individual test is carried out on the protocols to identify its fitness for that property. For example, there is a separate test measuring the ordering fitness of a stack from the test for its reliability property. This section describes the design of each of these tests and their implementation using the objects and methods described in section 6.3. The five tests are: a communication test, a reliable multicast test, a group test, an ordered multicast test and a quality of service test.

6.4.1 Communication Test

A fit stack is able to send and receive multicast messages in a perfect environment. To test this a sending process casts 25 messages to a group of two other members. The number of messages received by each member is counted and returned as a value between 0 and 50.

This is implemented using the following steps:

- The fitness function calls the MCastTest method from the ProtocolObj passing the stack it has generated as a parameter.
- The MCastTest method creates two threads, one for each receiving process. Each thread calls the member process of MulticastImpl objects on separate servers and waits for the value returned by that function.
- Both members connect to the same group name using channels constructed with the protocol stack passed as a parameter. The first member calls the TestSender method of SendingImpl object; both processes wait to receive messages.
- TestSender connects to the same group name with a channel of the same properties and then casts the messages to the channel.
- After the two have received all twenty-five or timed out, they return the number of messages received.
- These values flow back to the fitness function to give the measure between 0 and 50 (0 implies all messages received).

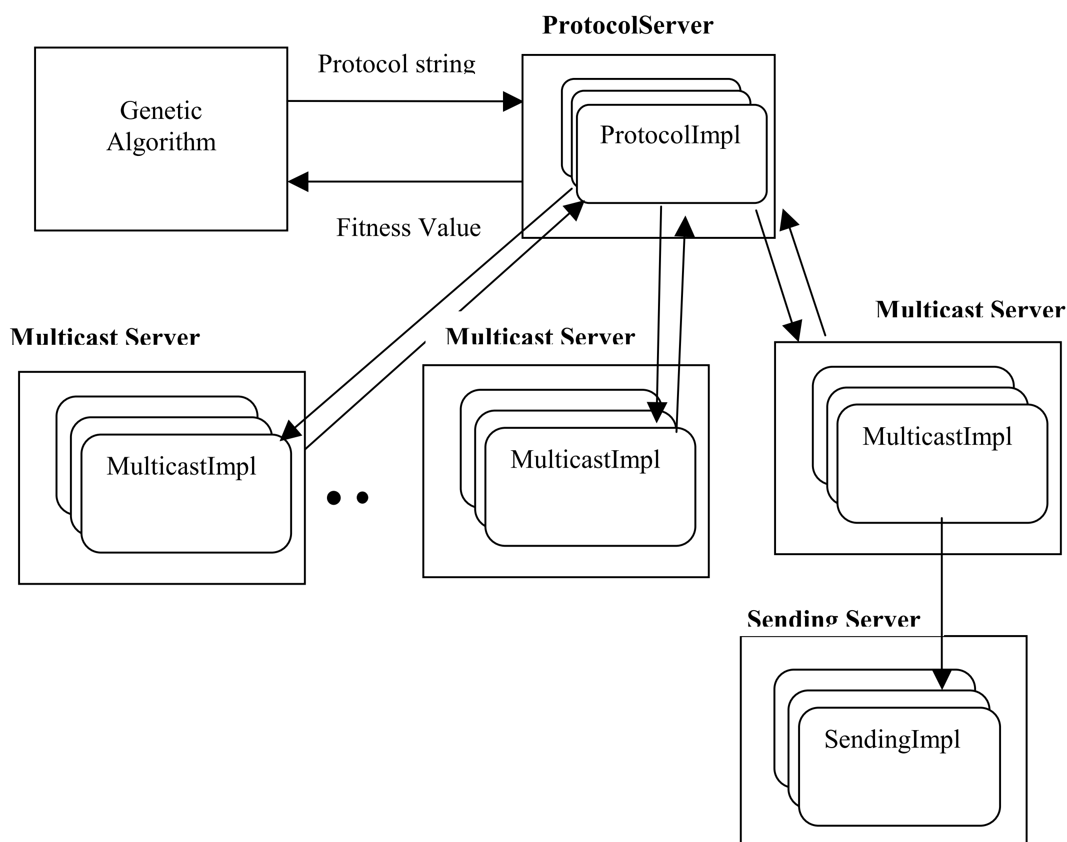

Figure 6.1 Architecture of the Genetic programming system to create reliable, ordered multicast protocols

Object Name	Method	Description
ProtocolImpl	MCastTest	Initiates the creation of a set of three group members by calling member() twice with a stack. Waits for the return of the number of messages received by both receivers under normal network conditions.
	MCastDropTest	Creates a set of three group members by calling discardmember() twice. Waits for the return of number of messages received under lossy network conditions. Also waits for the QoS measure of stack to be returned.
	MCastGroupTest	Creates a set of three members using groupmember(). Waits for the return of number of messages from both. Adds a new group member by calling groupmember(), then reinitiates cast of messages. Waits for the return from three processes of number of messages. Finally, disconnects the first member and reinitiates the send and then waits for two remaining to return.
	McastOrderTest	Creates a set of two members, one receiver (OrderMember()) that tests the order of messages and waits for the return of this measure
MulticastImpl	Member	Creates a process that connects to the group with a channel constructed using the stack passed as a parameter. Waits to receive messages from the channel. When complete returns the number received.
	DiscardMember	Creates a process that connects to the group with a channel constructed using the stack passed as a parameter. Inserts the discard layer into the stack to simulate loss. Waits to receive messages from the channel. When complete returns the number received. Creates a timing thread and calculates the throughput; returns this measure.
	GroupMember	Same process as Member, however it does not close down after one sending set and uses RMI to pass the number of messages back to protocolimpl between each send.
	OrderMember	Creates a process that connects to the group using the stack passed as a parameter. Inserts the jumble layer to simulate out of order transmission. Receives messages and calculates a measure of the ordering received. Returns this measure.
SendingImpl	TestSender	Constructs a process, connects a channel using the stack passed as a parameter and casts 25 messages before disconnecting
	DiscardSender	Same as TestSender, but inserts the discard layer into sending channel's protocol.
	GroupSender	Same as TestSender, but does not close after one send. Waits for indication to send again. Contains two Boolean flags go1 & go2 that are changed by RMI calls. When changed the method sends again.
	Change	Simple method that alters the value of Boolean flags go1 & go2 when called.

Table 6.1 Description of the methods used in the genetic programming system

6.4.2 Reliability Test

The reliability test assesses how well a stack provides loss-less transmission of messages to all members of a group in a non-perfect environment. The tests are run on a local area network; therefore, the probability of packet loss is very low. This means that the dropping of messages must be created to simulate a lossy environment. As described in the experiment in chapter 5, the DISCARD layer provided by the JavaGroups toolkit is inserted above the transport layer of the protocol. This ensures that a random number of messages are removed from the stack before they reach the bottom layer.

The structure of the test is identical to the communication test of 6.4.1. However, each group member inserts the DISCARD layer into the protocol being tested. One sender multicasts 25

messages to a group containing two more elements. Each receiver counts the number of messages it has received. These are totalled to provide a scaled rating between 0 and 100. If 50 messages are received then the protocol provides fully reliable multicast communication.

The implementation uses the following steps:

- The fitness function calls the MCastDropTest method from the ProtocolObj passing the stack it has generated as a parameter.
- The McastDropTest method creates two threads, one for each receiving process. Each thread calls the Discardmember process of MulticastImpl objects on separate servers and waits for the value returned by that function.
- Both Discardmembers connect to the same group name, but the protocol stack is parsed and the DISCARD layer is inserted above the UDP layer (This must be here, because test 1 was passed). The first Discardmember calls the TestDiscardSender method of SendingImpl object; both processes wait to receive messages.
- DiscardTestSender connects to the same group name, inserting the DISCARD layer in the channel stack and then casts 25 messages to the channel.
- After the two have received all twenty-five messages or timed out, they return the number of messages received.
- These values flow back to the fitness function to give the measure between 0 and 100 (0 implies all messages received).

6.4.3 Group Communication

One of the properties that the stack must provide is that of being able to deal with the dynamics of a group of processes. It must correctly respond to new member joins as well as leaves. To assess this, the test is designed to simulate the changes that incur when a group communicates.

The test initially creates a group of three members and multicasts 25 messages. The group is then changed to four members and the initial sender multicasts another 25 messages. Finally, one of the original receiving members is removed from the group and the sender multicasts 25 messages.

Each of the receivers counts the number of messages they correctly receive and these are totalled out of 175. The weighting requires a measure between 0 and 100; therefore, the count is scaled to lie within that range.

This is implemented uses the same steps as test 1 but uses the McastGrouptTest, GroupMember and GroupSender methods in place of the standard methods.

6.4.4 Ordering Test

This test creates the measure of the fitness of a stack in providing ordering of messages between group members. That is, a fit stack ensures the receiver member sees the messages

in the same order that they were sent by another member. An unfit stack may partially order the messages or not deal with the order in any manner.

The testing environment is within a local area network, therefore, the probability of messages being jumbled during transmission is low. There must be a method for creating out of order transmission. Two possible methods are available:

- A separate process that alters the order of messages during transmission. This could take the form of removing a message from the channel and then waiting a period of time before replacing it.
- A protocol layer can be created and inserted into the stack. The layer has the functionality of changing the order of the messages that go through it. That is, it has local state for storing messages and sending them later.

There are two problems with the initial method. It is difficult to implement, because the process of placing messages on the channel must be altered in some way. Secondly, it is difficult to control because an extra process is running in the background and must be synchronised with the numerous other processes that deal with the channel communication. The protocol layer method is less complex to implement; JavaGroups provides general outlines of how new layers should be implemented so they fit easily into the stack architecture. There is no extra control structure needed; as shown with the DISCARD tests the layer is simply inserted into the protocol being tested. Due to these benefits the method of *protocol module insertion* was chosen. The design of this module is described in section 6.5.

To test for ordering, a group of one sender and one receiver is created. The testing of the correct order in more than one receiver is desirable; however, ordering is an extremely time intensive operation, so one is chosen to keep the testing time at a minimum. If one receiver orders correctly then it is assumed that all members will do the same because each uses the same protocol.

The sender casts a total of 25 messages each with a number indicating its position in the sequence. The receiver uses the following algorithm to measure the order. It examines if the message sequence number is one greater than the one it just received. If it is then the fitness is increased by one. This will total a score out of 25; however, the weighting requires it to be between 0 and 100, therefore, it is multiplied by four.

The implementation carries out this test with the following steps:

- The fitness function calls the MCastOrderTest method from the ProtocolImpl Object passing the stack it has generated as a parameter.
- The MCastOrderTest method calls the Ordermember process of the MulticastImpl object on and waits for the value returned by that function.
- The Ordermember inserts the JUMBLE protocol to the top of the stack and then connects to the group using a channel with the updated protocol. It then calls the TestSender method
- TestSender connects to the same group name, inserting the JUMBLE layer in the channel stack and then casts 25 messages into the channel.

- The receiver calculates the order fitness and returns this.
- The value flows back to the fitness function to give the measure between 0 and 100 (0 implies all messages received in order).

6.4.5 Quality of Service Test

A number of different stacks may meet all of the functional requirements i.e. they provide reliable, ordered, group communication. However, some of these stacks can be determined as being fitter than the others. This measure depends on the performance of the stack in providing a certain quality of service.

The quality of service test measures the *throughput* of each protocol stack. [Fluckiger, 1995] defines throughput to be the number of binary digits that the network is capable of accepting and delivering per unit time. In the case of these tests, it is the number of messages sent and received per unit time.

To create the throughput test, a timing thread is initialised in parallel with one of the receiving processes. This thread is started after the first of the 25 messages is received and stopped after the last has arrived. The time the thread accumulates in this interval measures the throughput. However, a function is needed to turn this into a fitness value between 0 and 25. The graph in figure 6.2 shows how to calculate this measure. If the throughput is equal to or lower than a given value (in this case the throughput of UDP i.e. the best throughput) then it is a given a value of 0. Otherwise the throughput is scaled to be between 0 and 25.

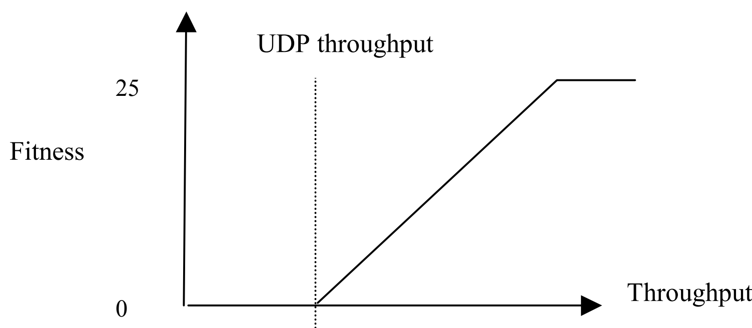

Figure 6.2 Graph representing the fitness measure for a given throughput value

The implementation is carried out as part of the reliability test. It is therefore implemented within the discardmember method. This decision was taken to reduce the time of testing by placing two tests in parallel. One of the receiving members carries out the QoS test while it is performing the reliability test.

6.5 Design and Implementation of a micro-protocol

In order to provide the test for ordering of messages, a protocol layer needed to be created to jumble the order of messages as they are transmitted through the stack. This section describes the design of a new protocol layer to perform this called JUMBLE.

The initial design decision was whether to physically alter the messages or simulate the alteration by altering sequence numbers in the headers. JavaGroups adds separate sequence numbers for each layer of the stack e.g. the TOTAL layer has a different sequence number than the NAKACK layer, therefore, it is difficult to change the order if these layers aren't in the protocol stack. Therefore, a decision to physically alter the messages was taken.

A protocol layer has two basic methods that must be created: *up* and *down*. These specify the operations carried out on the message (e.g. add header, remove header) as it is passed up or down the stack.

To re-order the messages only the up method needs to be created (The down method simply passes messages on with no processing), so that messages are jumbled before they are sent. In order to do this the layer contains an array of 10 messages. A random number of messages are placed in the array and not passed to the next layer when received. In their place one of the current members of the array is sent instead. However, there must be a dummy message in the array or the first change will have nothing to send.

```
public class JUMBLE extends Protocol {

    public Object[] saved = new Object[100]; // Array of stored messages

    public void Up(Event evt) { //Method that deals with events passing up stack
        Message msg;
        Int pair1, pair2 ;     //Pair1 holds the first value to start switching on
                               //Pair2 holds the next switch number
       int pointer =0;         //Pointer to next message to send
        case Event.MSG:                        //If a message occurs do the following
            Object temp_obj = evt.GetArg();

            if ( temp_obj instanceof Message ) {
                msg = (Message) temp_obj;
                y--;           //increase number of messages received
                u++;           //increase pointer to array position

                saved[u]=temp_obj;     //Store message in array
                if (y==pair1)
                        return;        //First switch so simply don't pass

                if (y==pair2){         //Pass a message in array instead
                        Message msg2 = (Message) saved[pointer];
                        msg.SetBuffer(msg2.GetBuffer());
                        msg.SetDest(msg2.GetDest());
                        msg.SetSrc(msg2.GetSrc());
                        msg.SetHeaders(msg2.GetHeaders());
                }
                pointer++;
                pair=pair+5;
            }
                break;
        }
        PassUp(evt);            // Pass up to the layer above us
    }
}
```

The JUMBLE.java code (part of which is displayed above) shows the implementation of this algorithm within the framework of a protocol layer. To test the layer, a standard send-receive application was used with UDP:JUMBLE as the stack. The messages were received out of order, identifying that it functioned correctly.

6.6 Testing and Results

To test the system it was run using three workstations. Therefore, the genetic algorithm process and one multicast server were started on one machine. The protocol server and one multicast server were initiated on another machine. Finally, the remaining multicast and sending server were placed on the last workstation. This provided an appropriate distributed basis for the testing of group communication.

Due to the length of time for one test to be completed, the GP was run five times. In the first three runs, the test for ordering is not included as a fitness measure; therefore, it is a test for reliable group multicast. The last two represent the complete solution of reliable, ordered multicast. The results from these five runs can be found in table 6.2 and 6.3.

Test Number	*Population size*	*Result*	*Generations*	*Fitness*
1	100	UDP:NAKACK:PING:FLUSH	2	71
2	200	UDP:PING:MNAK:NAKACK:FLUSH:GMS	3	35
3	300	UDP:PING:FD:MNAK:FLUSH:GMS	3	0

Table 6.2 Results of GP runs for reliable multicast protocol generation

Test Number	*Population size*	*Result*	*Generations*	*Fitness*
1	300	FD:UDP:MACK:PING:NAKACK:FLUSH:GMS	3	83
2	500	UDP:PING:FD:NAKACK:FLUSH:GMS:TOTAL	4	0

Table 6.3 Results of GP runs for reliable, ordered multicast protocol generation

6.7 Analysis of Results

There are a number of points that can be identified about the results:

- The tests show that the genetic program can be successful. The third case in table 6.2 and the second case in table 6.3 show that the protocol stacks for reliable multicast and reliable, ordered multicast have been effectively generated.
- The number of generations chosen to iterate through is low. This is because the time taken to process each generation is large (although it depends on population size). For

example to test one stack using all five tests takes approximately 15 minutes. Therefore, the increase in the number of generations makes the time to complete infeasible.

- The solution provided is not always zero. This is explained in the case of each of the tests by table 6.4:

Test No.	Explanation
1	There is no GMS layer to control group changes. Therefore, the fitness reflects the stack's inability to recognise new members and discard old ones. It may also perform poorly in terms of quality of service.
2	All layers for functionality are included, however, there is repeated functionality for reliability. This means that the quality of service requirement has not been met completely.
3	The fitness is zero and therefore provides a stack that meets both the performance measures as well as the functional requirements.
4	TOTAL is a complex layer to introduce, because it depends on a number of layers below it (PING:FD:FLUSH:GMS). Therefore, it is difficult for the genetic program to find the solution. In this case, TOTAL has not been introduced correctly and the fitness is high because it has failed the ordering tests, as well as some of the others.
5	Extending the population size has ensured that more evolutions have found the correct position for the total layer and introduced it in the correct place. Providing the required functionality.

Table 6.4 Explanation of fitness value returned for the five tests. Tests 1-3 are for reliable multicast. Tests 4 and 5 include an ordering test

- The time to run a GP is extensive, although no exact measures were taken. The worst case (test 5) was estimated to take over 8 hours, while the best case (test 1) took around 2 hours.
- The testing quickly consumed system resources; the use of a population size of 300 was chosen because the system ran out of memory in the initial generations for larger values than 500.
- The extending of the problem to include more functional and non-functional requirements shows that the probability of finding a perfect solution is lowered. In this case, the GP only found the solution two times out of five runs.
- Test 3 shows that the introduction of a quality of service measure ensures that protocols can be chosen that do not have redundant processing. For example, the repeated layer types of result 2 are not found.

6.8 Conclusions

The created system has shown that it is possible to generate a protocol stack that meets the needs of reliable, ordered communication automatically. However, the GP problems that were identified in the previous experiment resurface here. The complex layer structures are more difficult to evolve to and, therefore, reliability is a problem. The time taken to perform the tests is longer than the previous experiment and more system resources are used.

This test has also identified the important introduction of quality of service tests. The GP provides a number of stacks that meet the functional requirements, but the introduction of throughput measures ensures that stacks that perform better are selected.

Chapter 7 Conclusions

7.1 Overview of Work

The initial goal of the project was to investigate how to apply genetic programming to the domain of protocol configuration and to identify the benefits it may bring in this area. Conclusions from the project can be drawn from both the design of a genetic program for protocol configuration and the analysis of the results obtained from the runs of this genetic program.

A genetic program was designed to provide the protocol stacks for JavaGroups applications. This genetic program was tested to identify if it was possible to automatically generate a protocol for reliable unicast communication. It was then further explored by attempting to provide reliable, ordered multicast communication. This needed a more complex stack chosen from a greater number of layers. The testing of the genetic program showed that it was successful in generating the stacks for both cases.

7.2 Major contributions

The project has identified what it means to represent a protocol stack as a structure for GP to evolve. The use of linear stacks of individual building blocks provides a basis for the evolution operators to identify and evolve fit portions of the stack. The experiments have identified how genetic operators should affect a protocol stack. Crossover and mutation either swap or change linear segments of the stack's clearly defined building blocks to evolve.

The important issue of what is the fitness of a stack been identified. The first step identified that a protocol stack is fit if it meets its functional requirements. That is, the fitness measure is a range of values for each of the requirements. Each of these requirements is tested by an individual test and an overall value provides the fitness measure.

However, the results from the initial experiment showed that the genetic program provides a number of different protocol stacks that meet the functional requirements and there may be a need for the user to select the fittest from this subset. Their new requirements are the non-functional requirements that are met by the stack. That is, which stack meets their quality of service demands? The extended experiment added tests and a fitness measure that identified the fitness of a stack in terms of the QoS parameters of throughput and delay.

The use of a suite of five tests in the extended experiment identified the need for weighting of measures for each requirement. Stacks that score highly for reliability and ordering must be considered more important than stacks that score highly for throughput and reliability, because the winners are the basis of future generations and these are the properties that must be passed to provide a result. However, the issue of weighting to meet changing requirements is considered in the further work section.

7.3 Other notable contributions

An original aim of the project was to identify how reliable and expensive GP was in the area of protocol generation. These are two of the main problems associated with the technique and both were significant features in the testing of the system.

The results from both experiments showed that genetic programming is a time intensive task. In the case of the simple generation of a reliable point-to-point protocol the fastest time to generate a solution was only a few minutes. However, the tests also provided a solution that took 2 hours. This problem is shown more clearly by the extended experiment for reliable, ordered multicast communication; the time for the five tests ranged between 2 hours and over 10 hours.

Another feature of the genetic program is that it is resource intensive. Each test creates a number of processes distributed between two or three workstations with a high amount of system RAM. This limits the system to be used by only high-end systems, not low performance terminals such as mobile devices.

[Koza, 1992] identifies that GP is not a completely reliable method for finding the solution to a problem. It is possible that the solution given is not correct, however, it will be the closest to the solution from the generated set. This is also seen to be the case within the two experiments carried out. Solutions were given that did not provide a zero fitness measure (i.e. all fitness requirements were met), but did indicate fitness in some of the measures.

The problem of reliability in genetic programming is only dependant on the situation it is used in. If it is used to generate stacks for a programmer attempting to implement a group communication system in JavaGroups, then reliability is not a serious factor as the GP can be run several times if necessary. However, the GP could be running in an adapting environment that generates stacks during run time to be used by an application when the needs of its communication change. The GP must provide the correct solution in this case, or the user's requirements will not be met. For these situations time is also a factor. In the first case time is not critical, however, in the second 10 hours to generate a solution is infeasible.

The results from the two experiments showed that the choice of population size and the number of generations is an important factor. If the initial population of stacks does not contain the properties that will be needed, then the probability of a correct solution being reached is low.

The results of the two experiments also identified a trend between the time taken and the reliability of a solution being generated. If a small population and number of generations were selected then the time taken to provide a solution was shorter than for a greater population size and number of generations. However, the probability of the correct solution being generated in the later case was much higher. The trade off between time and reliability is useful in situations where a solution is needed quickly without all of the properties.

Finally, the experiments have shown that using the appropriate fitness tests the designed genetic programs will generate the correct stack for the communication type. GP also has the advantage of generating different stacks that may have not been thought of by a human designer, which may in turn provide better performance. For example, the generation of UDP:MNAK for point to point communication; this uses an alternative acknowledgement layer as opposed to the standard NAKACK layer. GP does suffer from the problems of reliability and resource use that hinder it in terms of a real time solution provider.

7.4 Further Work

7.4.1 Future testing of current work

Due to limitation of time within the project the second system only has five results generated. This is not a large enough measure to completely identify all trends and properties of the genetic program. Therefore, the system should ideally be run another say 20 times with a range of initial population sizes (because of the range of modules available the initial size will have a greater effect) from 10 to 1000.

[Koza, 1996] identified that in four application areas genetic programming out-performed a human designer. Further testing of the system, therefore could identify if this is the case for protocol design. The following could take place:

- Ask a human designer to create a stack for a given communication type and then test it for protocol performance properties.
- Perform the same tests on a stack generated by the genetic programming system and then compare the results to see which performs better.

This should be repeated for a range of communication stacks to ensure that the results are realistic.

7.4.2 Performance improvement

The performance of the individual tests identified that the protocols for group communication in JavaGroups are slow. [Hayden, 1998] identifies that Ensemble's protocol layers provide a high performance. Therefore, a possible measure to increase the speed on the genetic program may be in changing the performance of the building blocks and communication channel. To do this the Genetic program could be re-implemented in another toolkit e.g. Ensemble and the same experiments performed. The timing of the tests would identify whether this measure has any effect on the performance of the genetic program.

An alternative method for improving performance is to run tests in parallel. Therefore, the system could be altered to run on a specialised parallel architecture that allows each of the five tests for one stack to be run together or the testing of each stack to be carried out in parallel.

7.4.3 User specification of the Genetic Program's output

The main goal of genetic programming is to provide a system that allows the user to specify what they require and then have the system return the solution for this. The system in its present state does not do this. It has been designed to provide one solution in the case of reliable unicast and reliable, ordered multicast. Therefore, further work in the project area must investigate the possibility of providing solutions for a range of different communication types using one genetic programming system that takes as input the functional and non-functional requirements. This could be done by:

- Creating a method for a user to specify what they require from a protocol. This may take the form of a formal language description of the properties to include.
- Changing the fitness weightings to simulate the needs of the user. That is, the process must take the requirements and map them onto the weightings. For example, if a high throughput and low reliability multicast protocol was needed. Then the reliability weighting would be set of zero and the quality of service rating would be given the highest weighting.
- The changing of the weight ensures the performing from the test suite of only the tests that meet the requirements. For example, only running the tests for reliability and quality of service.

7.4.4 Dynamic adaptation

The area of dynamic protocols has identified the need to adapt at run time to the changing needs of a communication. That is, generate a new protocol stack or change the current one so that it still meets the requirements within the changed environment. The designed genetic program is static and run off line to meet the needs of an individual communication type. Having investigated the mapping of a range of specifications into a system. The next step would be to investigate if it feasible to apply this in a real time communication. The following would need to be performed:

- Design how to obtain the current environment conditions and requirements of the communication in real time. Once obtained, map these specifications into the genetic program to generate the adapted stack.
- The investigation of using only the current stack to populate the GP. It will be close to the solution and may reduce search time; this is due to only a small number of evolutions from this stack to the required stack.
- Creating increased performance of the genetic program. In its present state, it is not feasible to perform in real time. Therefore, an improved implementation is needed to deal with real time requirements.

7.5 Concluding remarks

This dissertation has identified that genetic programming has provided a successful technique for automatically generating communication protocols. The important issues of genetic programming have been fitted to the domain of configurable protocol stacks and protocols for determined communication types have been generated. However, the main problems of time and reliability within GP have been identified. Therefore, future investigation into this domain must provide suitable solutions to overcome these and make the method useful for real world applications.

Chapter 8 References

[Ban, 1999] B. Bann. JavaGroups User's Guide. http://www.cs.cornell.edu/Info/Projects/JavaGroupsNew/userguide/html/user/index.html

[Banzhaf et al, 1997] W. Banzhaf, P. Nordin, R. E. Keller & F. D. Francome. Genetic Programming: An Introduction – On the automatic evolution of computer programs and its applications. Morgan Kaufman. 1997.

[Birman et al, 1994] K. Birman & R. Van Renesse. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press. 1994

[Colouris et al, 1994] G. Colouris, J. Dollimore & T. Kindberg. Distributed Systems, Concepts and Design. Addison Wesley. 1994.

[Cornell, 1999] Cornell University. JavaGroups - A Reliable Multicast Communication Toolkit for Java. http://www.cs.cornell.edu/Info/Projects/JavaGroupsNew/

[Crepeau, 1995] R. Crepeau. Genetic evolution of machine language software. In Proceedings of the workshop on Genetic Programming: From theory to real world applications. J. Rosco editor. pp121-134. 1995.

[Darwin, 1859] C. Darwin. On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life. Murray, London, UK. 1859.

[Fraser, 1994] A. Fraser. GPC++ - Genetic Programming C++ Class Library. http://www.emk.e-technik.tu-darmstadt.de/~thomasw/gp.html

[GrefenStette & Baker,1989] J. GrefenStette and J. Baker. How genetic algorithms work: A critical look at implicit parallelism. In Proc. 3rd International Conference on Genetic Algorithms. pp20-27. 1989

[Hayden, 1998] Hayden, M. The Ensemble system. Cornel University technical report. January, 1998.

[Hayton et al, 1999] R. Hayton, A. Herbert & D. Donaldson. FlexiNet – A Flexible component oriented middleware system. In Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems. Lecture Notes in Computer Science, Vol. 1752. Pp 497- end. 1999

[Holland, 1975] J. H. Holland. Adaptation in natural and artificial systems. Cambridge, MA: MIT Press. 1975.

[Hutchinson, 1991] N. Hutchinson. The X-Kernel: An architecture for implementing network protocols. IEEE transactions of software engineering, 17(1), pp64-76. January, 1991.

[Koza, 1992a] J. Koza. Genetic Programming: On the programming of computer by means of natural selection. Cambridge, MA: MIT Press. 1992.

[Koza, 1992b] J. Koza. A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In Proceedings of IJCNN International Joint Conference on Neural Networks.Volume IV. pp310-318. IEEE Press. 1992.

[Koza et al, 1996a] J. Koza, F. Bennett III, D. Andre & M. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In Genetic Programming: Proc. of first annual conference. Editors J. Koza et al. pp123-131. 1996.

[Koza et al, 1996b] J. Koza, F. Bennett III, D. Andre & M. Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. Proceedings of the 1996 IEEE International Conference on Evolutionary Computation pp 1-10. 1996

[Kruthoff, 1999] A. Kruthoff. Jini and Software bus systems. IFI, University of Zurich. 1999.

[Mitchell, 1996] T. Mitchell. Machine Learning. McGraw Hill. 1996.

[O'Malley & Peterson, 1992] S.W. O'Malley & L. L. Peterson. A dynamic network architecture . ACM transactions on Computer Systems. 10(2), pp110-143. May, 1992.

[O' Reilly & Oppacher,1994] U. O'Reilly & F. Oppacher: Program Search with a Hierarchical Variable Length Representation: Genetic Programming, Simulated Annealing and Hill Climbing. In the Proc. Third Conference on Parallel Problem Solving from Nature. pp 397-406. 1994

[Plagemann, 1994] Plagemann, T., Gotti, A., Plattner, B. CoRA - A Heuristic for Protocol Configuration and Resource Allocation, IFIP Fourth International Wokshop on Protocols for High-Speed Networks, Vancouver, Canada, August 1994, pp. 85-102

[Plagemann, 1996] Plagemann, T. Protocol Configuration - A Flexible and Efficient Approach for QoS Provision, (short paper) Fourth International IFIP Workshop on Quality of Service - IWQoS'96, Paris France, March 1996, pp. 235-238

[Plagemann, 1999] Plagemann, T. A Framework for Dynamic Protocol Configuration, in European Transactions on Telecommunications (ETT), Vol. 10, No. 3, May June 99, Special Isssue on ARCHITECTURES, PROTOCOLS AND QUALITY OF SERVICE FOR THE INTERNET OF THE FUTURE, pp. 263-273

[Punch & Zongker, 1998] B. Punch & D. Zongker. Lil-GP genetic programming system. http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html

[Quereshi, 1998] A. Quereshi. GPsys. http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html

[Ritchie, 1984] Ritchie. A Stream Input-Output System. AT&T Bell laboratories Technical journal 63 no8 Part 2 pp1897-1910. October 1984.

[Softwired, 1999] Softwired Inc. iBus – The Java multicast object bus. 1998. http://www.softwired-inc.com/ibus

[Sun, 1999] Sun Microsystems. Java Core Reflection. http://java.sun.com/products/jdk/1.1/docs/guide/reflection/

[Teller & Veloso, 1995] A. Teller and M. Veloso. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101. Department of Computer Science. Carnegie Mellon University. Pittsburgh. 1995

[Teller, 1996] A. Teller. Evolving Parameters: The co-evolution of intelligent recombination operators. In Advances Genetic Programming 2. Editors P. Angeline et al. pp 45-68. Cambridge, MA: MIT Press, 1996.

[Teller & Veloso, 1996] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In Symbolic Visual Learning. K. Ikeuchi & M. Veloso editors. pp 81-116. 1996.

[Van Renesse et al, 1996] R. Van Renesse, K. Birman & S. Maffeis. Horus: a flexible group communication system. Communications of the ACM. April 1996.