Framework for Self-Healing and Dynamic Construction: Applications of the Software Matrix

BY

ANIRUDHA KRISHNA

Thesis submitted in partial fulfillment of the requirements for the
Degree of Master of Science in Computer Engineering

ADVISOR:

DR. JAMES FAWCETT

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

SYRACUSE UNIVERSITY

December 2005
Syracuse, New York

Table of Contents

Index of Diagrams and Tables

Chapter 1

Introduction

Reliability has always been a concern in building large software projects. It is defined as 'The ability of a system or component to perform its required functions under stated conditions for a specified period of time' [1]. Failure of a system to carry out its required functions may be due to errors caused by programmer actions or omission, faults in the software or external failures. Since completely eliminating failures is almost impossible, we must consider how systems can recover from them. In the past, the only systems which promised reliability have been complex designs with multiple levels of redundancy. The problem with this approach is that as applications themselves become more complex, building additional failure recovery mechanisms only complicates both development and maintenance. What we need, are tools that provide these features while keeping designs simple. In the foreseeable future, new applications will continue to pose greater demands as they become more distributed and feature rich. Developers need new tools that support this greater need without increasing the complexity involved in building them.

What we need then, is tools and Framework support that are simple and easy to understand. In this project, we explore this concept using two techniques. The first is to attempt to raise the level of abstraction that a developer works with. This has certainly worked in the past – object orientation and Framework libraries both work very well in this context. The second is to strip applications down to their bare functionality. Complex interfaces and practices can often be replaced with simpler techniques that achieve the

same function. As we will see later, example of this is the messaging mechanism used by the Software Matrix [2]. It follows a simple XML based Message interface that is able to handle even the complex task of exception handling between distributed components. We use the Software Matrix [2] and the Self Healing Framework to explore these ideas and to try to achieve them.

1.1. Research Proposal

In this research, we have two goals; the first is to identify alternate approaches to system development. The second is to develop a framework to that will aid in simplifying system design. In more concrete terms, we consider a process called Dynamic Construction, in which new Applications are built at run time by adding components as they become necessary. We use the same techniques to construct a Framework which supports building Self-Healing applications. As we will see, a lot of current research is being done in automating parts of development and maintenance processes to ease the range of responsibilities taken on by developers and administrators [6][11]. The Self Healing Framework is a step in that direction.

The Software Matrix Framework developed by Riddhiman Ghosh and Dr Fawcett [2] is a versatile idea that can be adapted to suit different needs. Experimenting with the Framework to enable greater support for reuse, we came across different areas in which it could change the way systems are designed and developed. We use this Framework as the basis for both Dynamic Construction and Self Healing. The Matrix Technology and its features are described in Chapter 2.

The Matrix was designed to promote reuse of software components. It proposed building independent Cells that communicate through a Message Passing interface to promote loose coupling. In the course of this research, we also explore how this innovative technology can be leveraged in different areas, how application design is affected and what new techniques can be used in developing software.

Self Healing has been described as "any device or system that has the ability to perceive that it is not operating correctly and, without human intervention, make the necessary adjustments to restore itself to normal operation"[3]. There are three major components to this definition – error detection, recovery, and restoring the system to normal operation. Much work has been done in this field both in the past and in ongoing research. In this thesis we suggest a new approach to dealing with the same problem, one that implements a newly conceived technology to automate the recovery process. The focus of the thesis is on recovery and rebuilding from failure. Although Error Detection is a required part of the project, simple schemes have been used for error detection.

Dynamic Construction is a concept that grew out of the way the Software Matrix was implemented. The dynamic nature of the Matrix Executive allows composing Cells to be added to an application at run time. In this work, this concept was extended to allow complete applications to be built in this fashion.

In addition, we have examined the Matrix Framework itself, made some modifications both in the interest of enhancing performance and efficiency and to enable new functionality. The final goal is to develop a complete framework of Matrix Cells that may change the way Software is built. We take a few small steps in that direction here.

1.2. Automated Computing

An Autonomic program is one that can configure, heal and manage itself without external input [4]. Automated processes can include installation, error detection, recovery from failure, adapting to changing environments and optimizing operation. This is a very broad definition but most programs contain at least some element of automation. The basic exception handling mechanism most modern applications implement is a mechanism for recovery from failure.

The goal of automating applications is to ease the burden of Developers, Administrators and people who maintain large systems. The advantage to developers is that they don't always have to program for the worst case scenario [5].Taking into account boundary conditions of the Self-Healing Framework; a developer can focus on building functionality instead of particular quirks of the environment. The biggest advantage however is to the Administrators and maintainers. If an application is able to automatically detect faults and heal itself, then most of their job is done for them.

Automation has been carried out at various levels in the past [13], although it has mostly been for special requirements and customized applications. It is only recently that

large corporations are investing in research in this area [6][11]. The following two sections describe current research in the area of automating software systems. They very closely reflect the information provided on IBM and Microsoft's Websites. The sections are meant to provide a background to the research topic and are not part of the thesis work.

1.2.1. IBM Autonomic Computing

IBM's Autonomic Computing Initiative [6] is one of the recent forays into the field of automating Software Systems. The main drive for this innovation was reducing complexity in building and operating large Software Systems and reducing their dependence on human intervention for maintenance. Their Autonomic System has four requirements [6] –

- Self Configuring – The system is able to determine the capabilities of its environment and adapt itself to install and operate in that environment.
- Self Healing – The system is able to discover faults and restore itself to recover from those faults. This definition also includes operations performed to prevent faults from recurring.
- Self Optimizing – continuously monitor use of resources and optimize operations to improve efficiency
- Self Protecting – prevent, detect and recover from external attacks.

The system consists of an autonomic manager and managed resources [7]. Resources can consist of single servers, databases, network components, or collections of these. Resources are controlled by embedded sensors and effectors. Sensors are

monitored to identify departure of the system from normal operating conditions and effectors are used to restore systems back to their ideal state. The manager operates on a loop consisting of four stages – monitoring the sensors, analyzing the received information, planning actions and executing them through the effectors.

The model can be used in the construction of new applications or in stages to convert legacy systems. IBM has defined four Maturity Levels, basic, managed, predictive and adaptive to describe the various levels of integration [8]. Integrating systems consists of embedding sensors and effectors and using log adapters to convert logs and events into a generic format. The management operations are carried out by commercial and open source tools from different vendors. For example an IBM tool can be used to monitor sensors while a Toshiba tool can be used to implement the effectors. Complex systems are be built by placing management components at different levels. Having individual resource managers and encompassing system managers allows the management policies to be distributed and thus simpler [9].

IBM has developed a toolkit comprising libraries and tools that work with the eclipse environment to support building autonomic enabled applications. The toolkit consists of tools to convert system resources into managed resources and autonomic managers that monitor the managed resources. [10]

Obviously, the IBM autonomic initiative is much larger in scale than anything we hope to achieve in this project. We are dealing only with the self healing property and

even that in only a limited fashion. But our goal is to show the even such complex tasks can be implemented using simple designs and technologies like the Matrix and to use the self healing framework to prove it.

1.2.2. Microsoft Dynamic Systems Initiative

This section describes Microsoft's efforts towards automation. The material is collected from the information provided by Microsoft at its Dynamic Initiatives Website [11]. The Dynamic Systems initiative is Microsoft's approach to building self-managing applications. While IBM's research is focused on developing Frameworks to support for building self-managing applications, Microsoft is involved in building Software that manages custom built applications. The core focus is to leverage knowledge about the system to build, modify and operate the system more efficiently and reliably. DSI has three main areas [11] –

1. Building model based software tools to help capture system knowledge to efficiently design applications and IT systems.
2. Making a more operationally aware windows platform
3. Building easy to use model-based management tools.

The technology is based on the System definition model, a modeling language used to describe complex distributed systems. The language is supported by the Visual studio 2005 developer to integrate management parameters into new Applications. The

knowledge captured through this model can be directly converted into management packs that are handled by the Microsoft operations manager tool.

Microsoft's approach is in some ways similar to IBM's autonomic computing in that is has the same goals but the difference is in the strong focus on knowledge collection and representation using the system definition model.

1.3. Jini

Jini is a Java technology that supports distributed computing [12]. It offers some elements of the dynamic composition of services that we are trying to achieve with the software matrix. A Jini Service consists of a custom service, a client and a lookup service. The custom service registers itself to the lookup server at startup. When the client contacts the lookup to locate the custom service, the registered service object is passed back to it. The client then uses this object to interact with the custom service.

Jini is very similar to other distributed component technologies but the reason it is presented here is because it shares a few things in common with the Self Healing framework we developed. As we will see later, the address server used in this implementation is based on a similar model as the Jini Lookup service. In addition, Jini supports the dynamic discovery process that we are trying to achieve by using the address server.

There are however some key differences. The most important one is that the matrix was built for reuse and is very good at it. All Matrix components can be moved from one application to another without any effort while Jini services are usually customized applications. The second is that the dynamic discovery process is only part of the healing framework's design. The construction process allows components that were not previously available to be brought online which means that new applications can be started whether or not the services they require are available at that time. It is only required that supporting Cells exist in the Repository.

1.4. Our Approach

Looking at the direction in which computer technologies are heading, we attempt to build a framework that will support development of distributed network applications. Using the Matrix technology allows us a significant advantage over other efforts – simplicity in design. As discussed previously, a large problem with building reliable systems is in handling complexity during development and operation. Simplifying the design allows developers to build more versatile systems with less effort. The goal is to build a Framework of Matrix Cells that when complete will encapsulate almost all general operations required by software developers. This will allow applications to be built by 'putting together' Cells rather than writing code, greatly simplifying development. Other enhancements like a Self Healing Framework built from Cells and Analysis tools can be built to support post development maintenance efforts. In this research, we focus mainly on building a framework to allow distributed network applications to be dynamically constructed and repaired using a Cell Repository. We also

examine different areas in which the Matrix can be utilized to ease burdens on developers. Since the Matrix is a new technology, we also constantly evaluate its features and try to build on it to create a more robust and efficient tool.

Self Healing frameworks are a part of the Autonomic computing concept [6]. We use a combination of traditional method, of maintaining redundant components along with Matrix specific technologies to build a system that automates component revival after failures. The system is mainly a test of the Matrix's capabilities and provides ideas for future research.

Chap 2

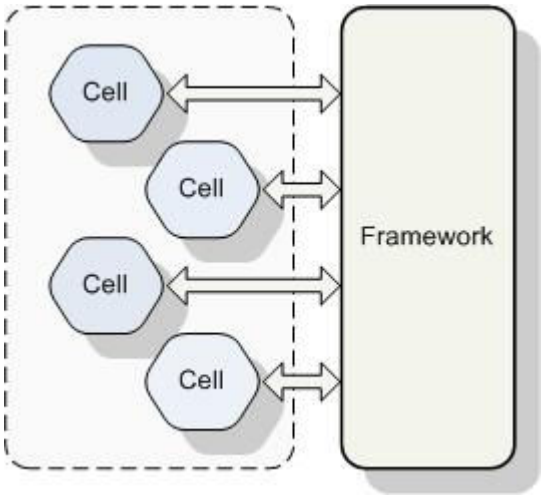
Tools and Architectures Used

This chapter describes the tools and technologies used in the applications that were built. The Matrix Framework is described in detail since it is a nascent technology and is not widely published.

2.1. Software Matrix

The Software Matrix is an Architectural Framework designed jointly by Dr. James Fawcett and Riddhiman Ghosh during the process of Riddhiman's Thesis research [2]. It provides support for building components that can be reused with almost no transformation cost. The significant advantage provided by this model over more traditional component based architectures is the simplicity with which existing Cells can be integrated into new projects. Although not covered in the original research, one interesting side effect that we discovered and make use of in this thesis is its ability to compose applications at run time by adding Cells.

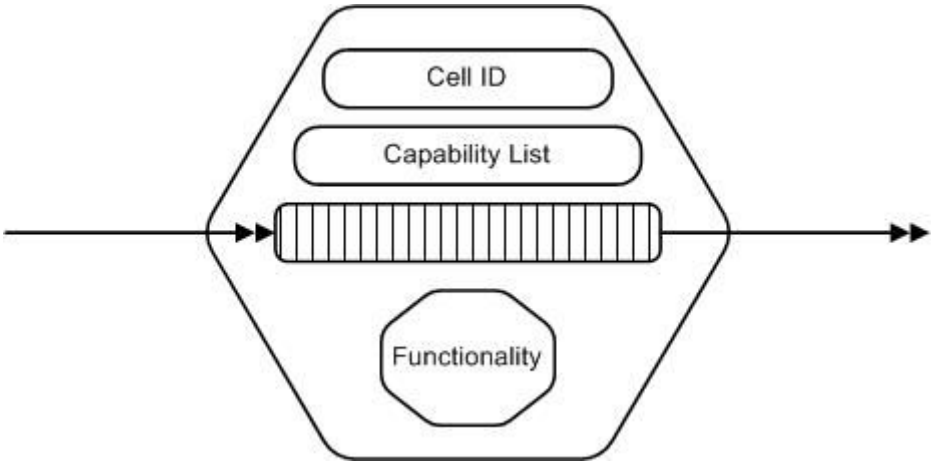
The Framework itself consists of well defined components called Cells and a Mediator based communication framework that allows interaction between Cells. This combination allows loose coupling between components to such an extent that Cells can be integrated just by placing their files into a pre defined folder. The Matrix has three important components – the Cell, Message Passing infrastructure, and dynamic construction. These topics are briefly described in the following sections.



2.1. The Software Matrix

2.1.1. The Matrix Cell

The Cell is the basic building block of the Software Matrix. It is similar in design to Classes in Object Orientation but is considerably larger in scope, usually demonstrating component level functionality. Applications are built by combining Cells that possess the desired characteristics.



2.2. The Matrix Cell Structure

Since all components are Cells, they can function either as Servers or as Executives. A server Cell provides certain services. Requests are collected, processed and replied to. Executive Cells control the flow of the Application by making requests to servers, and displaying results. Most UI components are implemented as Executive Cells. Every Cell, however, consists of some basic components –

Capability List

Each service provided by a Cell is given a name called the Message Type. The list of Message Types that the Cell can process is its Capability List. To make a request of this service, a message, with the desired Message Type, is sent to the Cell.

Cell ID

Each Cell instance created at run-time is assigned a Globally Unique Identifier (GUID) called the Cell ID. This uniquely identifies the Cell and is used to aid delivery of Request and Reply messages to the Cell.

Message Queues

Cells can accept incoming request messages from multiple locations at different times. A queue is used to buffer incoming requests so that none will be lost when the Cell is busy processing a previous request. The Cell also holds a response queue to save response messages.

Functionality

Functionality is the processing that the cell performs on receiving a request. The parameters contained in the request message are used to perform the required operation and generate a response. For example, a File Handler Cell might, in response to a getFile

Message, look at the Filename parameter, retrieve the requested file and construct a response message consisting of the file contents.

Design

Matrix Applications are composed of Cells. To allow Cells to effortlessly integrate into applications, they need to follow some conventions. Each Cell is required to derive from a Cell interface. This interface consists of functions that implement the basic Cell characteristics like the Unique ID, the Message Queues and Capability lists. It also includes message passing support and the installation information required when the Cell is registered. Most of the implementation for these functions is independent of Cell Functionality and can be generated through a Cell builder wizard. The only requirement for building new Cells is to name the Messages Types that it handles and to implement the functionality required to process those types.

The Server and Executive styles of operation are controlled by two functions running on different threads. The Process function receives incoming Messages and redirects them to user defined functions for processing. The Start function is run by the Matrix Mediator when the Cell is first installed into the Application and can be used to implement the Executive. Some other functions include Register and Unregister functions used to load and unload the Cell from an Application. Extract, Accept, Send and SyncSend are used in sending and receiving Messages. Send delivers an asynchronous message to the specified handler Cell while SyncSend is used when a reply is required. Extract and a corresponding ExtractResponse are used to decipher incoming requests and

responses. The Accept function deposits a message into the Cell's message queues. GetId and QueryCapability functions are used in Addressing and Message Handler Discovery.

2.1.2. Message Passing

The Matrix Architecture relies on a Message Passing mechanism to enable communication between Cells. Message Passing allows Cells to interact with each other without having to bind together as in a Procedure Call oriented design. All messages follow a defined XML format. Message generation and parsing structures are generated by the wizard and need not be re-implemented for each Cell.

The communication is based on a Mediator style architecture. A central data structure maintains references to all installed Cells and forwards messages between them. All messages are directed toward this Mediator and replies are delivered through it.

Message Interface

The Message Structure used is based on a simple XML Format. Message Calls are semantically equivalent to Function calls. The Message contains an input parameter list, a name (as defined by the Message Type) and a response object list. In addition it contains addressing information to locate the sender and the destination. Each Message also identifies itself uniquely with a GUID to enable tracking the request and response and to allow synchronized message transfer.

```
<?xml version=\"1.0\" encoding=\"utf-8\"?>
<Message>
      <Type>Request</Type>
      <NetworkSend>false</NetworkSend>
      <Source>0</Source>
      <Destination>0</Destination>
      <RequestID>c2076107-2e85-4bac-b55c-fb8898db988f</RequestID>
      <RequestCellGUID>e44d2e74-0369-4b47-aa7e-e2773b8dc0dd</RequestCellGUID>
      <ResponseCellGUID>NOT_SET</ResponseCellGUID>
      <Method>
            <Name>Matrix.Testing.TestApplication.TestMessage</Name>
            <ReturnVal>
                  <ReturnType>Not_Needed_3_21_04</ReturnType>
                  <ReturnValue>NOT_SET</ReturnValue>
            </ReturnVal>
            <NoOfParams>1</NoOfParams>
            <Param>
                  <ParamType>System.String</ParamType>
                  <ParamValue>AAEAAAD/////AQAAAAAAAAAGAQAAAARUZXN0Cw==</ParamValue>
            </Param>
      </Method>
</Message>
```

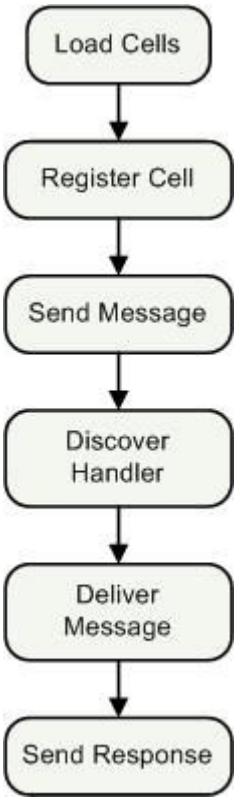
2.3. Matrix Message Format

2.1.3. Matrix Executive and Dynamic Nature

Matrix Applications are built by adding different Dll and Exe files containing the Cells into a specified directory. This directory is called the Plugin Folder. The Matrix Executive initiates a Loader module that monitors this directory and identifies all new Cells. Once a new File has been identified, reflection is used to create a new Cell instance. This object is registered into the Matrix Mediator. This loading process is the base of the Dynamic Nature exhibited by Matrix Applications. New Cells can be added to the Plugin folder at any time and they will be loaded into running Application. Since all Cells use the same interface, they immediately gain access to all of the previously installed Cells. A complete Matrix Application consists of a number of Cells packaged with the core Framework components (Matrix Mediator).

Walkthrough of Program operation

A Matrix Application consists of different cells placed in the Plugin folder. A Cell is first identified by the Loader and installed into the Mediator. The Loader creates an instance of the Cell and calls its Register function to retrieve the Cell's Capability List. The list is stored into the Mediator table for message addressing. The register function then creates a new thread to run the Start function allowing the Cell to perform any initialization that it requires. Executive Cells may initialize the User interface or program control. The Cell may send messages to other Cells using either the Send or SyncSend operations. The function calls is supplied with a list of arguments to be included in the message.



2.4. Sequence of Operations

These are used to construct an XML message that is delivered to the Matrix Mediator. The Mediator checks the Message table for the Message Type and locates the Cell capable of handling the type. Once the Message is delivered to a Handler Cell, its Process function is called to perform requested operations and a reply is delivered back to the Mediator. The Mediator propagates this reply back to the original requesting Cell. The complete sequence is enacted every time a request is made between Cells.

2.1.4. Reasons for using the Matrix

Dynamic Nature

Although the Matrix was designed to support reuse, one of its important properties is that it allows Cells to be added to an application at run time. This allows dynamic creation of an Application by putting Cells together. The other important feature is the ease of the installation procedure. The Self Constructing approach uses these properties to build a system that automates the construction process. Healing is handled in the same fashion by reinstalling Cells that fail.

Server based Applications – Distributed Systems

The Matrix architecture facilitates building of Server-Client Architectures due to inherent Server like characteristics of a Cell. Each Cell behaves like a server accepting incoming messages, processing them and sending out replies. The XML Messaging format used also allows the Matrix to easily be adapted to handle distributed systems.

2.2. Extensions to the Matrix Framework

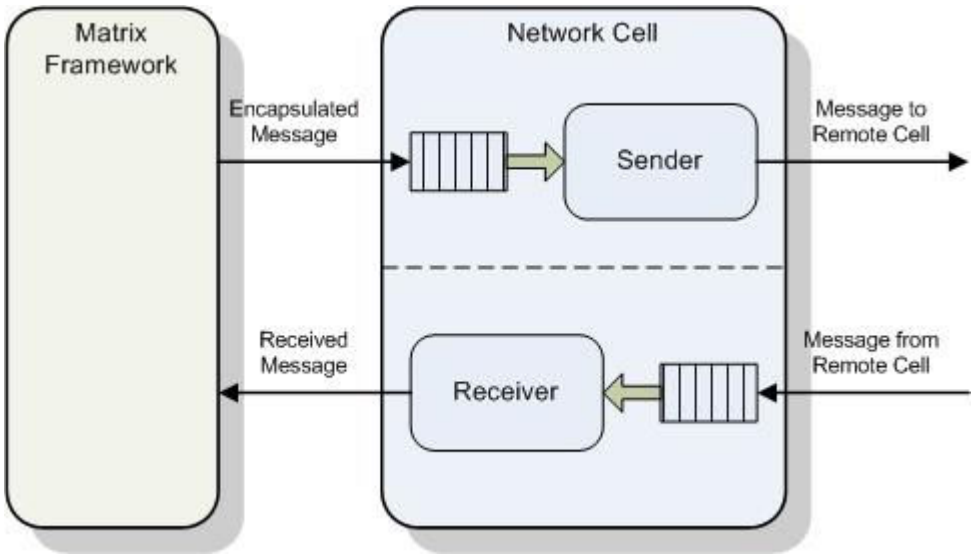
Although the Matrix is an extremely adaptable Framework, some changes were made to the basic architecture to adapt it for this research. One of the first considerations was to optimize the framework implementation to make it as lightweight and efficient as possible. The second motive for making the change was to enable seamless integration of distributed applications into the framework. This allows both single machine and networked applications to be built in the same style. The final change was designed specifically to support recovery from failure. Although many of the changes add features to the basic Framework, the main theme of simplicity, reusability and configurability were maintained so as to allow the new framework to adapt to the requirements of the program being built. In this section, we discuss the changes made to the framework.

2.2.1. Directory Watcher for the Loader Module

The Matrix Loader monitors a specified directory for Cells. As soon as a new Cell is added or an existing one modified, it identifies the change and loads the new Cell into the running application. The original implementation used a polling mechanism to implement the Loader and scan the directory. This was modified to use a Directory Watcher Class. A Directory watcher blocks and waits on events signaling a change in the directory contents. This is more efficient than a sleeping thread and improves performance by a slight margin.

2.2.2. Integration of Network Access – the Network Cell

The Matrix Framework contains a Mediator which registers Cells and allows messages to be transferred between them. This works very well for applications that run on a single system but one of the greatest strengths of the Matrix – the XML based message passing interface makes it ideal for building distributed applications that run in a networked environment. To allow the Matrix to take advantage of this feature, the Framework elements were modified to allow messages to be addressed to both Local and Remote Cells in exactly the same fashion.



2.5. The Network Cell Structure

The Network support is, in the Matrix style, implemented as a Cell. It creates a channel between different Mediators through which messages can be transferred. A message arriving at the first Mediator is delivered to its remote counterpart just as if it were a local request. The generated response is sent back to the Mediator at the source and is treated similar to a local response. The Cells themselves have no knowledge of

where the message was handled. To implement this channel, the Network Cell contains both a server and a client. Both components have their own input queues. The client behaves as a sender. It retrieves the destination address from the message and creates a connection to the Network Cell at the specified address the message is then transferred and the connection is closed. The next message prompts a new connection since it may or may not be to the same destination. The server component on the other hand is a single service that runs for the duration of the Network Cells lifetime. It accepts connections, retrieves messages and deposits the complete message into the local Mediators input queue.

The channel concept is further enhanced by the design of the Network Message. When a message is destined for a remote computer, the Mediator encapsulates it into another message destined for the Network Cell. The Cell only reads the wrapper to obtain the information it requires. The original message is extracted and transferred in its entirety without any modification. At the server side, the same message is converted from the byte array required for socket communication to the original string format and delivered exactly as it first arrived.

The sender and receiver are implemented using TCP sockets. This was considered ideal for efficient communication since delivery is the only required operation. Other network formats can also be used as discussed in the configurability section.

2.2.2.1. Format of a Network Message

The Network Message, as discussed above forms a wrapper for a Matrix Message. It contains the addressing information required to deliver the message to its remote destination. Although any Cell may construct a message in this format, the addition of Network Send and SyncSend functions into the Cell interface automate this process.

Matrix.Framework.Network.SendMessage – The Network wrapper message

Parameters

1. Message Type – The Message Type of the message to be delivered.
2. Destination – The IP Address of the destination Node.
3. Message – The complete message in string format.

Return Values

None

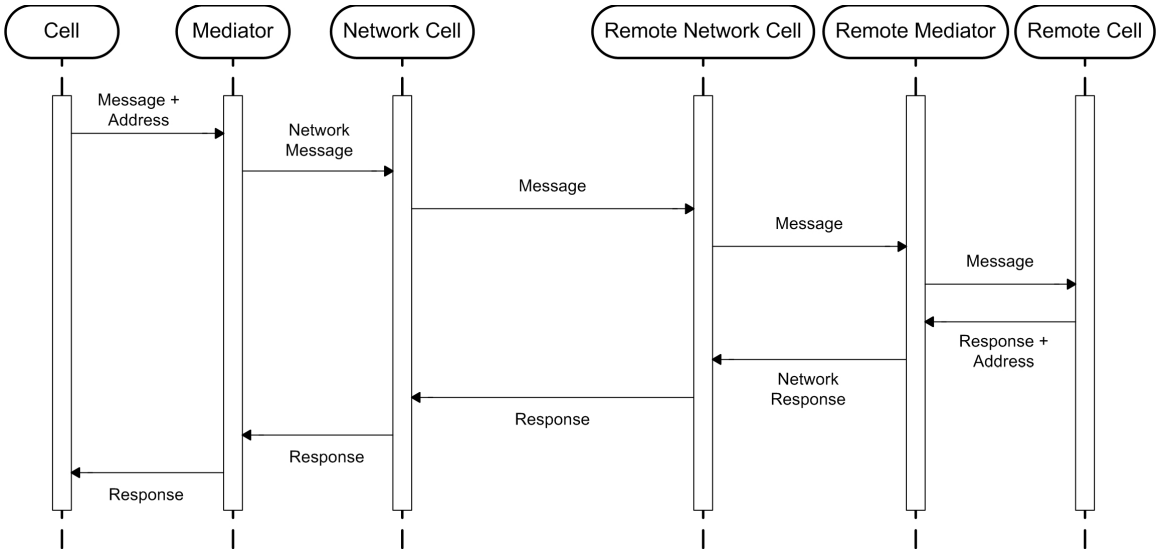
2.2.2.2. Adding Network elements

The Network communication component was implemented as a Cell to promote reuse and replacement in the Matrix style. The Network Cell provides a handler for the Matrix.Extensions.Network.SendMessage Type. Any Cell providing this same functionality can replace the Network Cell with no side effects. Modification only involves replacing the files in the Plugin folder. For example, the Network channel can be implemented using Remoting or Web Services. Additional operations can be added. Encryption and Decryption of messages can be handled at either end to allow more secure communication. Since messages are already encapsulated, they can be handed

over to other Cells before they are allowed to enter the Matrix system. These can provide authentication or screening operations. A lot of different options are available to improve and diversify the communication framework. We have taken the first steps by building the TCP Socket Network Cell.

2.2.2.3. Synchronous Message Passing

The Message passing implemented by the Communication Cell is not synchronous. Incoming messages are forwarded to the Mediator and outgoing messages to the remote location without waiting for a response. Any synchronicity must be implemented at the individual Cell. The SyncSend function already implements a synchronized local Message call so it was logical to extend this to support synchronous Network communication. The requests have the destination and source addresses built into them. To use this, the buildResponse function was modified to construct response messages that included this information. The addresses are then used to transfer messages back to the Node which made the request. The SyncSend function waits until a response with a matching ID is received, so the operation is completed when the response is delivered. This implementation allows local and remote Message Calls to be made in exactly the same manner with the only difference being the time required for Network transfers.



2.6. Sequence of events during Synchronous Message passing

2.2.2.4. Changes to the core Framework

Some changes had to be made to the Framework to integrate the Network components and allow new Cells to access their functionality.

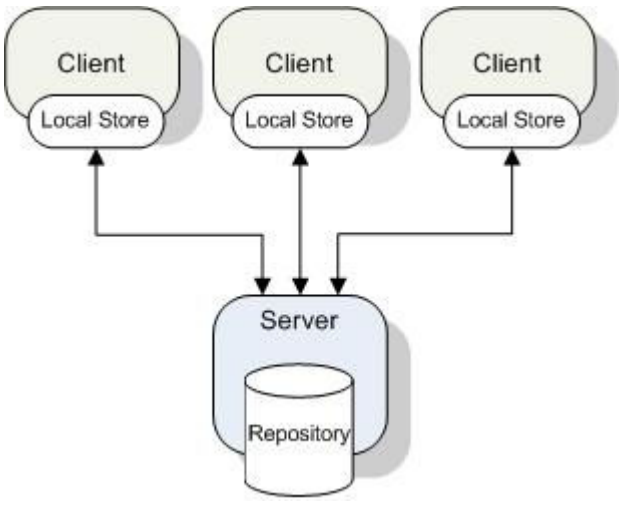
- The Send function was overloaded to allow two kinds of message delivery. The original Send requiring a Message Type and parameter list delivers a Message to a local Cell. The addition of a Network address as a parameter delivers the same Message to the specified address.
- The Mediator was modified to forward messages containing network address to the Communication Cell. It was also changed to accommodate response messages that arrived from remote Cells.
- Changes were made in the XML format of a Message. The basic Matrix Message was changed to allow it to include addressing information. Three fields, a source address, a destination address and a flag indicating whether a particular message needs to be transmitted to a remote Mediator or delivered to a local Cell were added.

2.2.2.5. The Matrix node

The Network Cell brings with it the concept of a Matrix Node. A Node in the Matrix is defined as the collection of required components that run in a single process. Each Matrix Node must contain at least one Mediator structure and the Network Cell. The concept of a Node is usually used with reference to a distributed system with multiple computers where each one can be considered a Node. The Mediator is required for communication within the process and the Network Cell for inter-process communication. The node is particularly important in the healing framework where a Node is the basic unit for which failure is detected.

2.3. The Repository Model

A Repository is a store of objects. A Repository Server allows a large number of objects to be stored persistently and retrieved efficiently.



2.7. The Repository Model

A lot of Applications operate on persistent data. This is information that is constantly stored and retrieved and must be available beyond the lifetime of the currently running program. Although data that is being currently modified may be stored locally at the client, large applications need enormous amounts of data which can only be stored in specialized data banks. A repository is one such Model. It stores information in a way that is properly indexed and easily retrieved. A Server-Client style is often used where the Application making the request behaves as a Client while the Repository is the Server which hands out the requested data.

2.3.1. Use of the Repository

The Self Healing Framework uses a Repository to create a persistent Cell store. When new functionality is required by the system or failed operations need to be restored, the required Cells are extracted from the Repository and installed into the system.

Chapter 3

The Healing Framework

3.1. Introduction

The focus of this research is to build an architecture to support developing Self Healing Systems using the Matrix infrastructure. Two ideas – simplicity and minimalism were suggested in the introduction. Here we attempt to use the Software Matrix which we believe to be both simple and minimal to construct more complex frameworks. The Self Healing framework brings to the Matrix world a property that was previously only part of much more complex systems. The Dynamic construction property it adds allows new ways to build applications by adding functionality to applications at run time.

This chapter describes the Self Healing Framework developed during the course of the research. The design and working of the Framework is described in the first section. The Framework is a composite of distinct elements each of which are described in technical detail. The later sections of this chapter explain the design strategies used to make the framework more configurable.

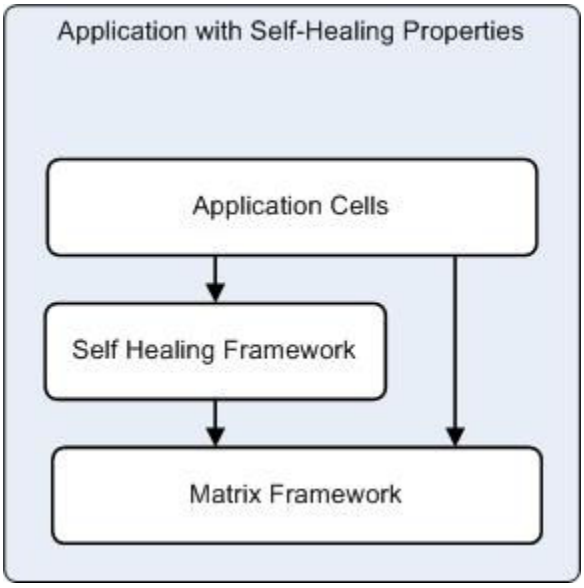
3.2. Design of a Self-Healing Architecture

The focus of this research is to build a Matrix Framework to support Self Healing and Dynamic Construction of Software. Self-Healing allows Applications to restore any lost functionality after failure of one of its components. Dynamic Construction is the process of building applications at run time. Required Cells are installed and made available only as and when their services are requested. Given a sufficiently large

Repository with a comprehensive collection of Cells, any new application can be built without writing any code at all.

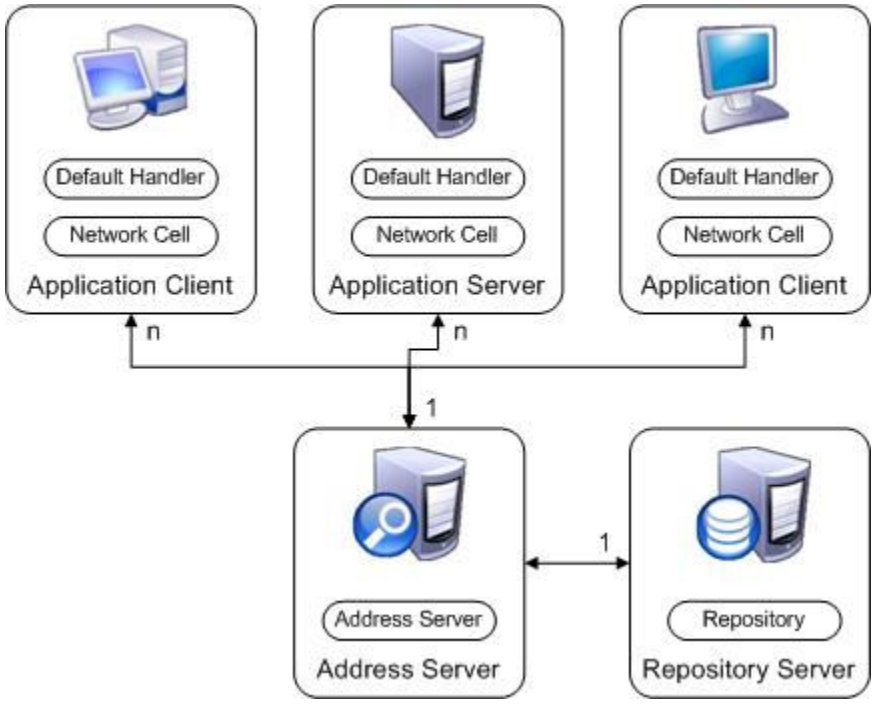
Both Self Healing and Dynamic Construction follow the same principle. They make use of the Dynamic Nature of the Matrix Technology to identify and install new Cells as they become necessary. The installation occurs after a failure is detected in the case of Self Healing and it is after a request for a new Message Type is made, for Dynamic Construction. The same Framework components and operations are used for both cases.

The Self Healing property as utilized in this context considers only loss of functionality, not loss of state. Any state information must be maintained and restored by the Application Cells themselves. For example, if a File Server in a network environment goes down, the Framework immediately restores the Server but the data it was managing is the Application's responsibility. However, we also discuss different techniques in which applications can be designed to support complete recovery of both data and operations.



3.1. The Self Healing Layer

The Self-Healing Framework follows the fundamental ideas of the Matrix. Its properties are built from Matrix Cells. The Default Handler, the Address Server and the Repository Server are the apparatus of the Framework. Any Matrix Application can add the Self-Healing mechanism simply by adding these three Cells – with absolutely no other modification. The Framework is targeted toward distributed applications with multiple client and server installations. The Healing property allows a client or server to be automatically restored if any one of them fails. Construction allows new servers to be installed as they are required.



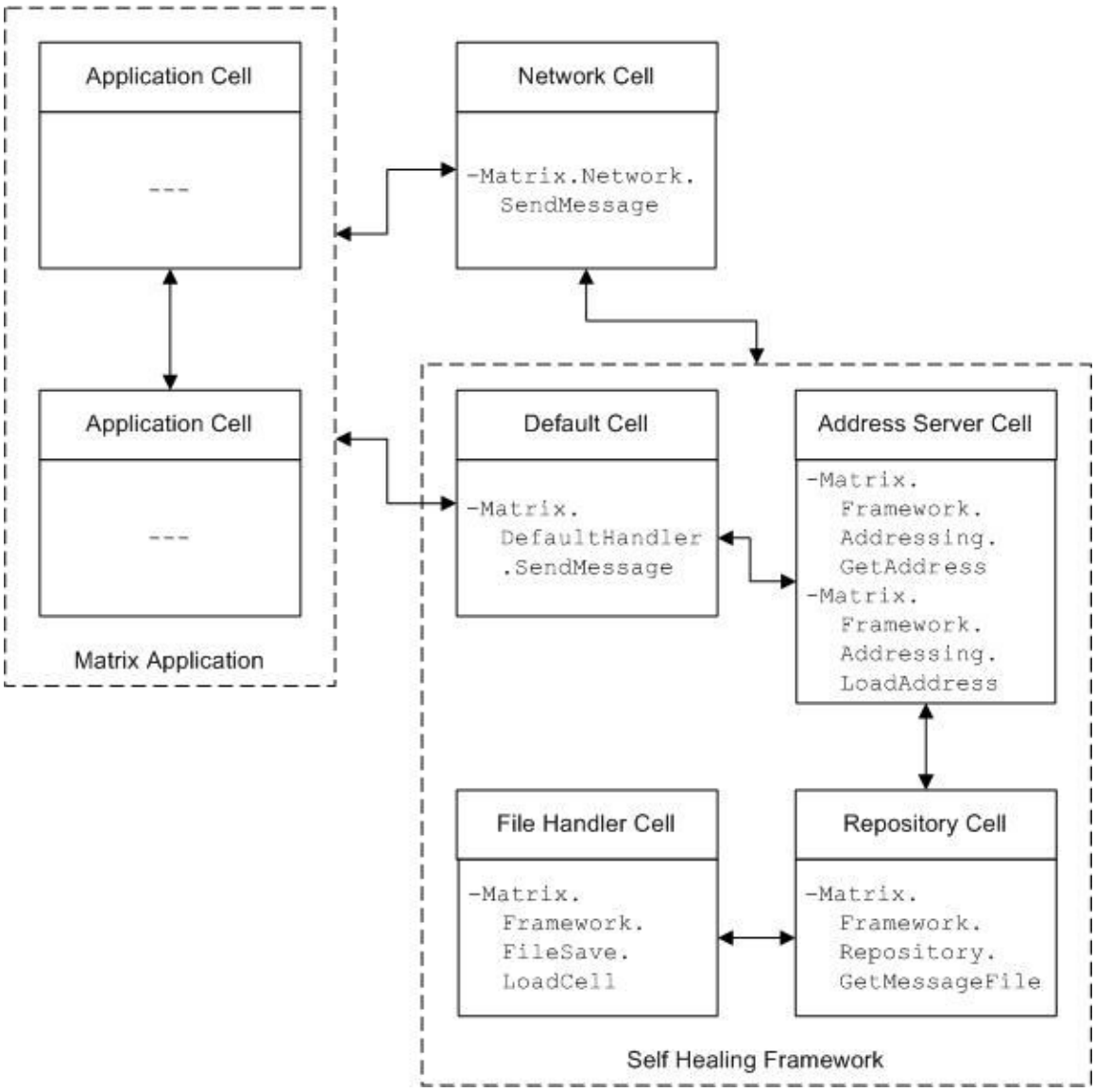
3.2. The Self Healing Framework Structure

3.2.1. Framework Installation

To implement the Framework, each Matrix application must install one Address Server and one Repository Server. These may though not necessarily run on a single machine. In addition, each Node that is part of the system must contain a Default Handler and a File Handler. All of these components are implemented as Cells and installation only consists of adding the Cells to the selected Plugin directories. The Repository must be populated with all the Cells that are currently part of the system and any Cells that may be required in the future. In this implementation of the Repository Server, adding new Cells only requires that the file containing them be saved into the Repository Directory.

3.2.2. Cell Diagram

A Cell Diagram is similar to a UML class diagram with a few variations. A Cell is represented by a rectangle with two boxes, one for the Cell Name and the other for the list of Message Types the Cell can process. Interaction between Cells is denoted by an arrow. A single headed arrow describes an asynchronous message while a double headed arrow indicated a synchronous message transfer.



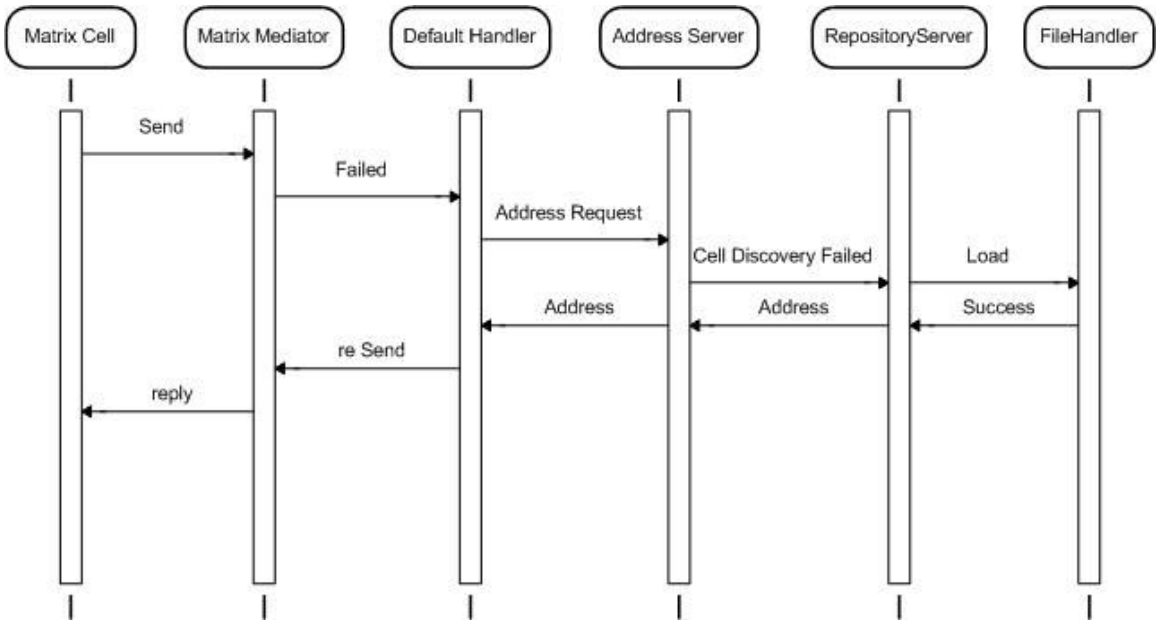
3.3. Self Healing Framework Cell Diagram

3.2.3. Operation of the Healing Framework

The Self-Healing Framework operates on two scenarios, Node failure and Cell unavailability. The first step in remedying these problems is to identify that they have occurred. To accomplish this, a few changes were made to the Matrix Mediator and the Network Cell. Both changes are semantically simple and allow a greater degree of configurability to the Matrix as a whole without affecting its normal operations. The new Mediator directs any message that it cannot handle to the Default Handler. This includes requests to new Message Types that have not yet been installed. Component Failure occurs when a Node in the network becomes inaccessible. Since all Message Handlers on that Node are no longer available, any message directed to them fails. The Network Cell detects these broken links and reports the communication failure to the Default Handler. These form the starting point for the Framework's operations.

The Default Handler is responsible for making recovery decisions at a local level. It maintains a Address table identifying the location of remote Message Handlers and tries to locate the requested service. If no suitable Handler is found, a request is made to the Address Server reporting the reporting the failure. The Address server is responsible for keeping track of Message Handlers at a global level. It checks to see if the required Handler is listed in its Address Table. If found, the relevant address is returned to the original Node. If no suitable entry is found, the Address Server contacts the Repository requesting a Message Handler for the missing Type. The new Cell is installed into one of

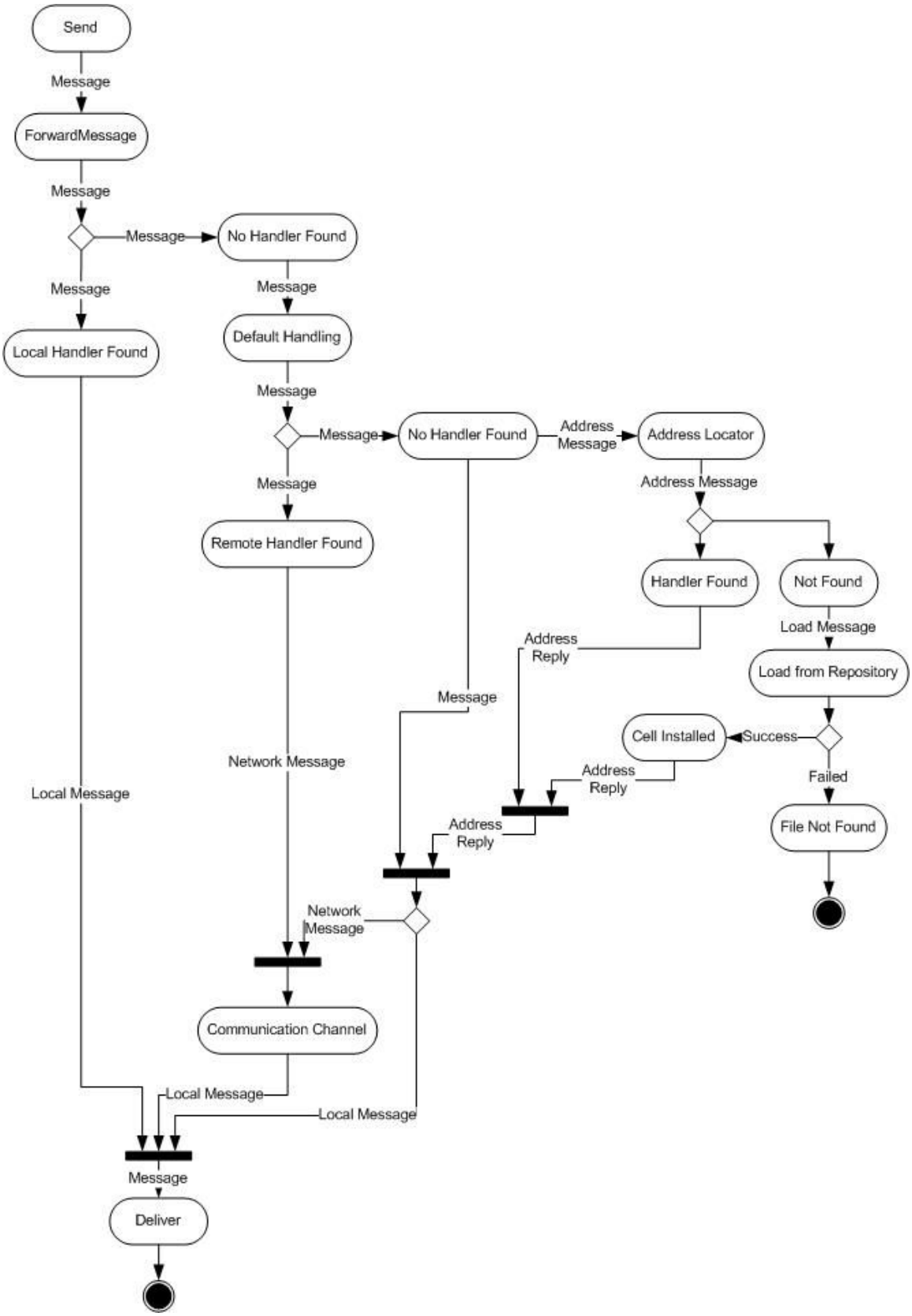
the Nodes and its address is returned to the Default Handler. The original message is then forwarded to the new destination.



3.4. Sequence of Events after Failure

Diagram 3.4 shows the sequence of events that occur in a complete Cell installation cycle. The File Handler Cell performs basic file operations like saving and retrieving files. It is used to save the new Handler Cell into the Plugin folder of the destination Node.

The complete data flow through the system is described by Diagram xx. The first forward operation is handled by the mediator. Operations of the Default handle show that the results of an address request are combined with the message to forward it to the identified destination Cell. The only point of failure is if the required Cell does not exist in the Repository.



3.5. Data Flow in the Framework

3.2.4. Configuration options

In a complex system like the Self-Healing Framework, there are many options that can be set to configure certain properties. Some operations require knowledge of the environment that has to be supplied by an external source. The Framework uses a configuration file to specify the options and collect the data. The Config file is an XML document with different tags for different options. The tags are intuitively named and can easily be set using a text editor. The file includes fields for local IP Address, directories for the Repository and Plugin Folder, Addresses of the Repository and Address Servers and all Nodes included in the Application, backup servers where new Cells can be installed and port through which the Applications communicate.

3.3. Framework Components

The Self Healing Framework consists of three important components, the Address Server, Default Handler and the Repository Server. The framework also uses a File Handler Cell to transfer files between locations. The Cell performs a generic file read and save function and is not described in detail.

3.3.1. The Address Server

The Address is one of the most important components of the Self Healing Framework. It forms a locus for all distributed applications and coordinates communication between nodes. It has a major role in the two important operations of the framework – Message Handler discovery and recovery from node failure. The Server behaves in part like a name registry where Message Handler addresses can be looked up.

It maintains an Address table with all the Message Types currently used in the system has the capability to install other handlers as required. The table is initialized at server startup by querying the different nodes and obtaining their capability lists. The server tables can be updated by any node as its capabilities change.

3.3.1.1. Server Initialization

As implemented, the Address Server obtains its information during startup by contacting each nodes listed in the Config file. Requests are made to the default handler's Matrix.Extensions.DefaultHandler.QueryNodeCapability Message Handler. The reply contains all the capabilities of that node and these are entered into the Address table along with the node's IP Address. For the sake of simplicity, duplicate entries are overwritten.

Two modifications can be identified for future work,

- The Address table can contain multiple entries for a message Type. This will involve using a more complex data structure but has the advantage that if one fails; the system can switch to the next immediately without having to install a new Cell. A load balancing scheme can also be implemented to make the system more efficient.
- A slightly more complicated scheme of requests can be implemented. The idea is that not all network nodes may be online at the time that the address server is initialized. The server can maintain a list of all network nodes which could not be contacted at startup and compare each request that it receives against this list. If the request comes from one of the nodes which have not been queried, a queryCapability request can be sent to the node before the address request is satisfied.

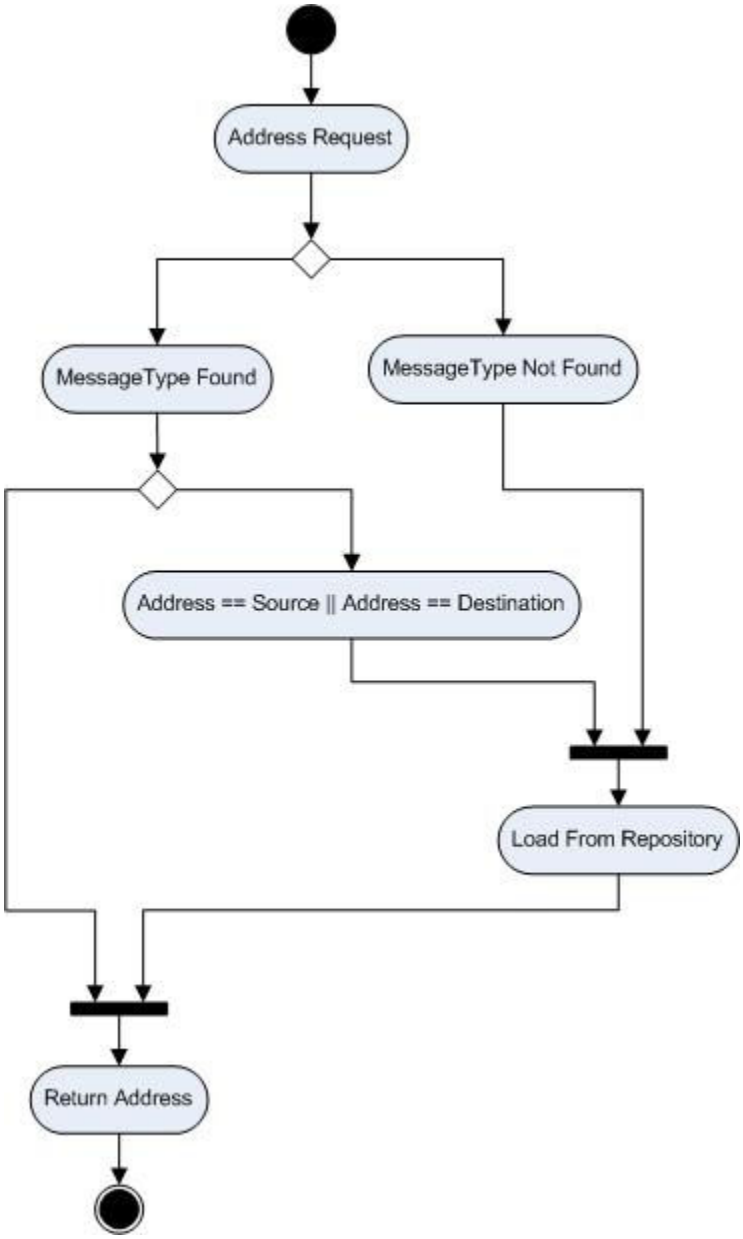
3.3.1.2. Design of the Address Server

The Server logic consists of a simple sequence of conditional queries to determine whether or not an address request requires a new Cell to be installed. Installation is required in two cases – the Cell does not exist in the framework or it was part of a node that failed. If a valid handler exists, its address is extracted from the table and returned.

The parameters supplied to the Address Request Message contain the IP Address of the sender. If an attempt was made to contact a remote Handler, its address is also included. Based on this information, the operations required of the Address Server can be inferred.

- The first check is to identify whether the Address table has an entry listed for the Message Type. If no entry exists, the cause can only be either a failure or non-existence of the Cell and an attempt must be made to install it.
- If a Destination Address has been specified, it is because the sender held a previously valid address for the Message Type but it failed.
    - If the Address Table has the same location as the one specified in the message, the cause is a failure of the remote Cell and it must be reloaded.
    - If the Address table entry is different from the specified address, a working Cell is assumed to exist in the new location and its address is returned. The assumption is based on the fact that the mismatch is most likely because the failure was previously reported and a new Cell has been installed.

- If a Destination Address has not been specified, it is because the sender did not have an Address for the specified Message Type.
    - If the Client's Address is the same as the Address listed in the table, the message is the result of a localized Cell failure and the Cell must be reloaded. This case is not often seen since we only consider node failure for the purpose of testing and failure of a Cell internal to a node is not recognized.
    - If entry in the table is different from the Client's address, the new location is assumed to contain a working Cell and its address is returned. In this case the sender's lack of knowledge does not imply that a Cell has failed. Only lack of information at the Address Server prompts an installation, as noted in the first case.



3.6. Address Server Logic

3.3.1.3. Installation

Once the decision to install a new Cell has been made, the Address Server must handle installation of the new Cell. The first task is to obtain the Cell from the

Repository. A Matrix.Extensions.Repository.GetFile is sent to the repository and the Message Handler Cell is obtained in the Return object. This file is then installed into one of the Matrix Nodes and its address entered into the Address Table. Once the process is complete, the new address is sent to the original requesting Cell. This process also introduces the two cases in which the Address request can fail. The first is if the required Handler does not exist in the Repository. The second is if installation of the Cell into a Node fails. In either case, the failure is reported back to the requesting Cell.

3.3.1.4. Address Server Message Types and Message Formats

The Address Server handles two kinds of Messages. The first is a request of an Address and the second is to inform the Address Server of changes in the capability list of one of the nodes.

Matrix.Extensions.Addressing.GetAddress - The GetAddress Message is a request from a Cell to the Address Server for the IP Address of a Message Handler.

Parameters

1. MessageType – a string containing the Message Type whose address is to be located
2. Source – the Address of the Cell making the request
3. Destination – the Address of the original destination of the message before it failed. It may be null if no destination was determined.

Return Values

1. CellLocation – the Address of the required Message Handler

Matrix.Extensions.Addressing.LoadAddress – It is sent to indicate that one of the nodes has made changes to the list of Message Types that it can handle. The Node can send a LoadAddress message to the Address Server to allow it to update its tables.

Parameters

1. Node – the address of the node whose list is to be refreshed.
2. MessageList – string array consisting of the new Capability List of the Node.

Return Values

None

3.3.1.5. Load Balancing

A simple load balancing scheme has been implemented in the installation of new Cells. Since this operation is handled by the Address Server, it is discussed here. There are two ways to determine the address at which the new cell is to be loaded Selection between the two schemes is made through a variable, DistributedBackup, in the Config file. DistributedBackup is a Boolean value and is set to true for the first case and false for selecting backup machines.

- The load balancing scheme used is to install new Cells into the Node that made the address request. When an installation is required to satisfy a request, the specified source address is used to select the target node. This allows new Cells to be installed close to the clients requiring their operation. The disadvantage is that the scheme does not provide any guarantees. It is possible, though unlikely that all new Cells might be installed into the same Node.

- The second option is to identify a backup server onto which all new Cells are installed. The location must be defined in the Config file and the selected location must contain at least a minimal node along with a Default Handler. This option is most effective if all the Cells of the failed node constitute a single logical entity like a server. An extension of this concept is where a list of Message types can be specified along with the addresses at which they should be installed if their Handlers fail.

3.3.2. The Default Handler

The Default Handler Cell is the generic interface to a Node. A Node may contain different Cells but each one must have a Default Handler. Simply put, it handles all default messages, i.e., any message that the Mediator is unable to deliver. These are either messages which cannot be delivered due to Node failure or messages for which no local Handler is found.

A part of the Default Handler's job is to cache Network Addresses of remote Message Handlers. A large part of the communication in distributed applications involves network messages that are not addressed at the time of creation. This is especially true in dynamically constructed applications where the addresses must be determined at run time. To prevent requests being sent to the Address Server every time, the Default Handler maintains a local cache of Message Handler addresses and forwards messages with listed Message Types automatically.

The Default Handler also allows querying of the node's capabilities. It extracts the list of Message Types from the mediator and returns it to any requesting Cell. This function is used by the Address Server to initialize its tables at startup.

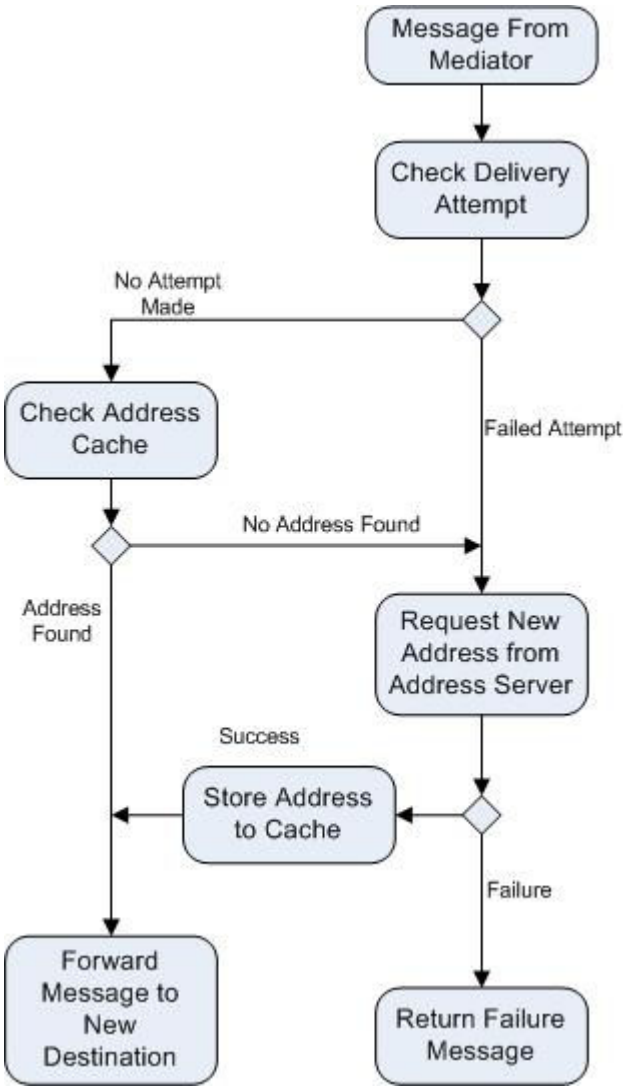
3.3.2.1. Design of the Default Handler

The Default Handler is responsible for three important operations. The first is to allow the system to recover from failure of components. It detects that failures have occurred by examining incoming messages. It is responsible for building new functionality into the system by requesting installation of new Message Handler Cells. Lastly, it is partially responsible for network addressing and Handler discovery by maintaining a cache of Network Addresses.

There are two kinds of messages that arrive at the Default Handler. The first is directly from the Mediator. This message does not contain any addressing information and is forwarded because the Mediator could not find a Handler listed in its registry. The second is from the Network Cell reporting that the Destination Node could not be contacted. This message may have been supplied a network address either by the original Cell or by the Default Handler itself. The first type indicates either a lack of addressing information or new functionality that must be installed. The second can only be due to the failure of a Node.

The Default Handler's job is to determine how these messages are to be processed. Only one of the above mentioned situations can be handled internally. If the

incoming Message has no addressing information and this can be supplied out of the Default Handler's cache, the message is immediately forwarded to the Network Cell for delivery. In all other cases, the Address Server is contacted to either install a new Cell or repair a broken Node. The return Value from both these operations is the IP Address of the new Cells location. The Default Handler uses this new address to forward the original message. The address is also cached in a local table for future reference.



3.7. Default Handler Sequence of Operations

In addition to these operations, the Default Handler can be queried about the Nodes Capabilities. The Mediator's registry contains the complete list of Message Types that the Node can handle. The Default Handler queries this registry and forwards the returned result to the requesting Cell.

3.3.2.2. Default Handler Message Types and Message Formats

The Default Message delivers failed messages along with some additional information to the default handler. Any messages that cannot be delivered by the Mediator are encapsulated and forwarded to the Default Handler. In addition, the Cell also accepts queries for the Node's Capabilities.

Matrix.Extensions.Default.DefaultMessage – Accepts failed messages and determines how to process them.

Parameters

1. MessageName – the Message Type of the message whose delivery failed.
2. Destination – this field is used to identify attempts made to deliver the message. If it contains a network address, the message delivery attempt failed. If it contains '0', the address in the Default Handler table is still valid.
3. Message – the complete failed message in string format.

Return Values

None

Matrix.Extensions.Default.GetNodeCapability – this message is used to request the Node's current Capability List.

Parameters

None

Return Values

1. CapabilityList – A string array containing the collection of Message Types handled by the Cells currently registered to the Node.

3.3.2.3. Configurability

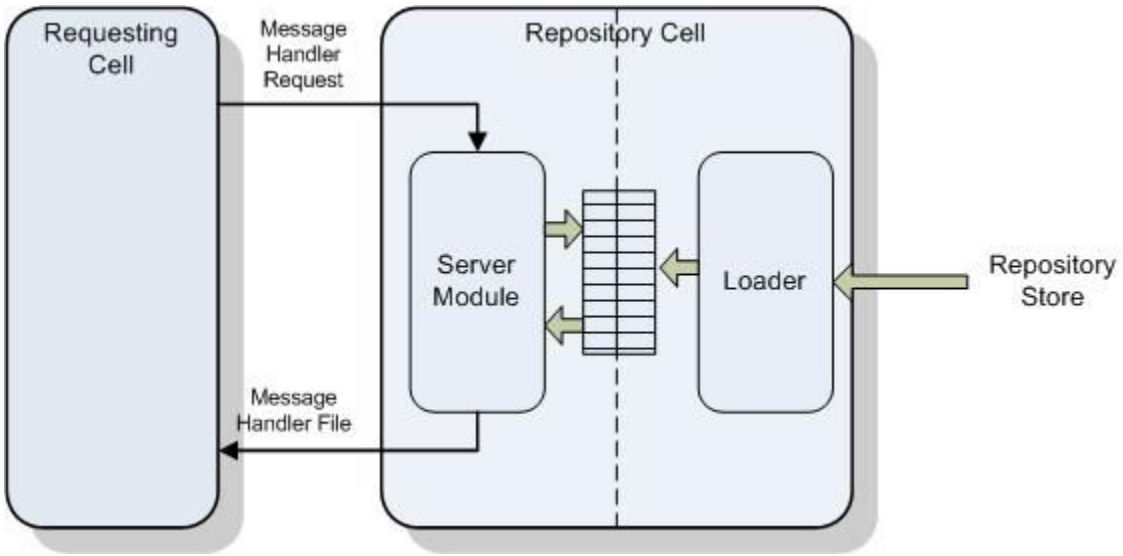
Since the Default Handler is a part of every node, efficiency and speed are important considerations. This implementation of the Handler is more complicated due to the requirements of the Framework that is being designed. If, however, a more simple architecture were required, not all of its functionality has to be included. The basic Default Handler must be able to process the two Message Types listed above and follow the same interface, but can simplify the implementation. An ordinary Message Delivery Failure exception can be returned to the original Cell if no Healing Framework were desired. The Network Address Cache can be eliminated along with the logic for determining the operations required to respond to incoming messages. This simplified Cell can be used in combination with other framework Cells to build a more efficient system which does not require Self Healing or Dynamic Construction.

3.3.3. Repository Server

The Repository Server serves as a backup for the Matrix system. It maintains a persistent store of the files that contain Matrix Cells. The store can be managed in different ways depending on the implementation of the Repository Cell. In this

Application, it is a collection of files stored in a directory that is monitored by the Repository.

The Repository is one of the more simple Cells in the Framework. It handles a single Message that returns a Handler Cell when supplied a Message Type. In the Self Healing Framework, it forms an extension of the Address Server's operations by supplying Cells as they need to be installed. It is an important part of the Dynamic Construction process since the required Cells must exist in the Repository for new applications to be built.



3.8. Repository Server Structure

The Repository Server consists of two parts, a Loader and a Server. The Loader handles the data management operations by monitoring a directory for Matrix files. It uses a Reflection mechanism similar to the Matrix Loader to verify that the files contain valid Matrix Cells. The Message Types handled by these Cells are entered into a table

and referenced to the file names. When a request is received, the table is used to identify the file name related to the requested Message Type. The selected file is then extracted and returned to the Address Server.

3.3.3.1. The Repository Message Types and Message Formats

The Repository Server fields only one Message Type. It accepts a Message Type name and returns the file of its Handler if one exists.

Matrix.Extensions.Repository.GetMessageFile – Used to request a Handler Cell from the Repository.

Parameters

1. MessageType – The Message Type for which a Handler Cell is required

Return Values

1. Filename – The name of the file containing the requested Handler Cell
2. File – A byte array containing the contents of the requested file.

3.3.3.2. Different Implementations

One important consideration for the Repository is redundancy. This is discussed in a Framework context in the next section but the repository has some features that make it unique. As it contains a large persistent store, failure of the node can cause a loss of valuable data. Since Handler Cells are no longer available, both Self Healing and Dynamic Construction properties are lost. Even if the Repository Cell can be brought back online, its data needs to be restored before it can be operational again.

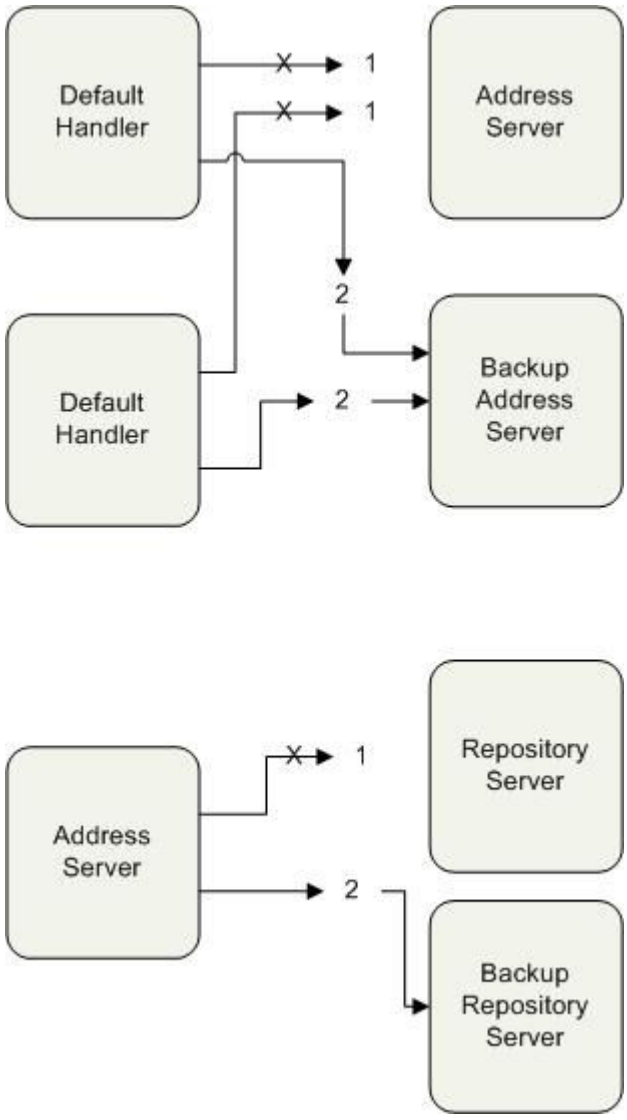
The two subsections of the Repository can be considered separately. The server can easily be restored by the framework since it is a stateless operation. This is discussed in section xxx. The Loader can be modified to keep the persistent data safe. The data can be maintained separate from the Repository Node. One way is to store the files and Message type information into a database. The database connection information can be stored into the Config file to allow the new Repository to access the files. This scheme is simple and can be easily implemented but requires the services of a database. The second method is to store the data on another machine or persistent disk. In this case too, the connection information can be saved in the Config file.

3.4. Additional Levels of Redundancy

The combination of an addressing server and Repository allows applications to restart in case a component fails, but what if the machines running the addressing server or Repository fail. There are different ways to deal with this kind of situation.

The first is to install mirror addressing and repository cells at multiple locations. Their addresses can be specified in the Config file to allow the system to switch to the new cells in case of a problem. The second solution is to modify the default handler and Addressing Server implementations to allow them to rebuild the Addressing Server and Repository Cell respectively. We discuss the details of both solutions. In each case, we consider the failure of the Address Server and the Repository Server separately.

3.4.1. Installing multiple Cells



3.9. Using Multiple Cells to add Redundancy

The Address and Repository Servers are constructed as Cells just like all other Matrix Components. They can be installed into a machine by just adding the files containing the Cells, into the Plugin folders of the required computers. The Config file

contains the address of the machine on which the Cells were installed. Mirror servers are installed by adding the Cells into more than one Plugin folder and listing each location on the Config file.

Repository – According to the design of the Self Healing framework, the Repository Server is only accessed by the Address Server. So the Address Server contains the logic for switching between Repositories. When a request to the Repository fails, the Address server automatically replaces its listed address with the next Repository location and continues operation as before.

Address Server – The Address Server can be accessed from any Default Handler Cell. Since no static information is maintained, each Default Handler must decide for itself whether or not or not an Address Server has failed. If an address message is returned, the Default Handler pulls up the next address server location from the Config file and forwards the request to the new server.

3.4.1.1. Server Startup

Restarting the failed server is only part of the solution to restore framework operation. Both the Repository and Address Servers maintain information stores that they require to function. The Address Server holds a table containing addresses of all message handlers and the Repository Server references a database of Cells. This static data must be restored for the Servers to be completely operational again. When mirror servers are maintained, the data must be replicated on both servers. All address information must be saved to both Address Servers and Cells must be stored in both Repositories.

3.4.1.2. Advantages and Disadvantages

Advantages

- The chief advantage of this scheme is speed. Servers can be switched almost instantaneous once failure has been detected as no extra work is required in restarting servers.
- Existing Cells need not perform any extra operations except for providing the capability to switch between servers. This does not affect efficiency of normal Cell operations.

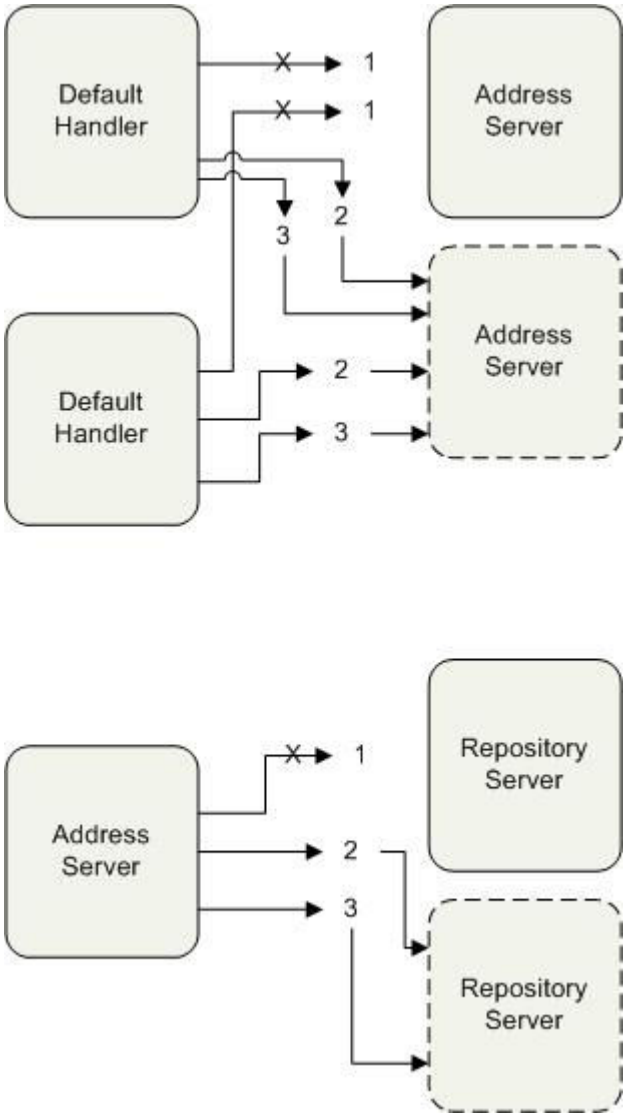
Disadvantages

- The first disadvantage is that multiple servers need to be maintained. Additional resources are required to support them.
- The reliability of the system is limited to number of servers pre-installed. If only two servers are set up, although highly unlikely, failure of both will bring the system down.
- Additional cost of updating two stores of information will affect the speed of operation of the system as a whole.

3.4.2. Modifying Existing Cells

The second method is almost recursive in nature. The system uses its own self healing property to heal itself. When a failure is detected, the Cell detecting the failure uses the Self healing technology to restore the failed Cells. In this case, the default handler is responsible for restoring the Address Server and the Address Server is responsible for restoring the Repository. Significant changes must be made to these Cells

to allow them to support complete reinstallation of the failed components. Since no centralized information store exists, the locations of the Address Server and any new location where it can be installed must be specified in the Config file. This is to prevent multiple installations of the server by different Default Handlers as they individually discover the failure.



3.10. Adding Redundancy by Modifying Existing Cells

Repository – As before, since only the Address Server requires access to the repository, it is in charge of both detecting failures and restoring the server. The destination can be any computer which has an active Matrix service and an open Network channel. The Repository Cell can be installed by depositing the Cell into the destination Plugin folder just like any other Cell installation.

Address Server – the installation of the Address Server is slightly more complicated since it is accessed by multiple Default Handler Cells. The Config file must contain the address where new Address Server can be installed. This is to prevent multiple Address Servers being installed by different default handlers as they discover the failure independently. The first handler to discover the failure installs the Address Server into the specified location by setting up the Address Server Cell. When other Default Handlers discover the same failure, they must first verify whether the Address server has already been installed before trying to access it. This requires that each Default handler contain additional logic to determine how to proceed when it discover the failure of the Address Server.

There is one unique problem that arises. Since the Repository Server is not available to load the Address or Repository Cells, these files must be stored separately. Each default handler must be able to restart a new Address Server so each node must hold Repository and Address Cells that it has access to at any time.

3.4.2.1. Server Startup

The problem of startup also has to be handled differently. Since failures cause loss of data, it must be restored before the new servers are fully functional. The Address Server data can be restored by making requests of all Default Handlers to provide capabilities of their respective Nodes. This information can be used to populate the Address server's tables. In effect, the design is changed to decentralize the Addressing information. The Repository is more complicated since a data store of Cell files has to be replaced. The only sure way of doing this is to keep a backup data location and allow the repository to access it at startup; the backup location can be specified in the Config file. One way of setting up the repository is to extract the cached Cells at each node and use those files to temporarily populate the repository until the store is manually repopulated. Another option, as suggested before is to use a database to store Cell files.

3.4.2.2. Advantages and Disadvantages

Advantages

- The biggest advantage is that there is no limit on number of failures that can be recovered from. As many addresses as required can be specified in the Config file and servers can be brought online as long as there is even one machine running the Matrix service.
- The second advantage is that fewer resources need to be online at any time since multiple servers do not have to be maintained.

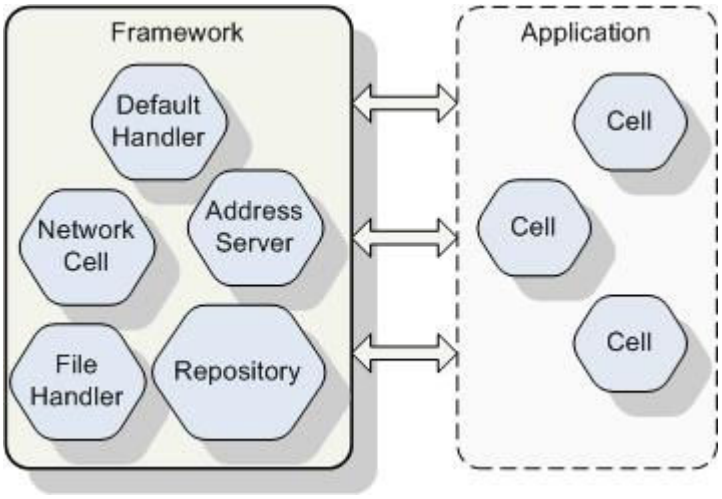
Disadvantages

- The biggest problem is the added complexity since the Default handler and address servers need to provide installation capabilities. They also need to maintain the files required for the installation.
- Installation time is added to recovery time after failure since the servers have to be installed and set up before they become available.

3.5. Framework Configurability – Adapting to different needs

The Software Matrix is an excellent platform for building a diverse range of applications. The Self healing framework only adds to this base by making systems more reliable. However many applications may not require this extra functionality. Simple applications which have no need for Address Servers or Repositories should not be burdened with having to provide resources for them. This section looks at how the Framework can be adapted to provide different levels of support based on the application's requirements.

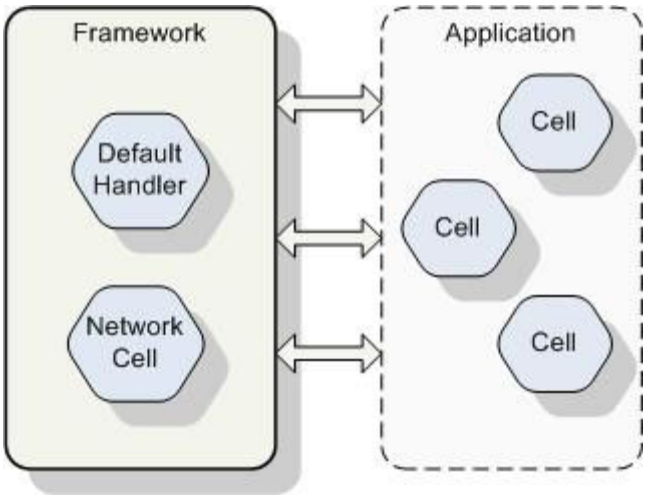
The Matrix allows Applications to be built by placing Cells in the Plugin folder. Removing Cells is just as easy. Just removing Cells from the Plugin Folder removes them from the Matrix Application. Due to the exception handling capabilities, any Messages sent to non existent cells are effectively dealt with and the system can continue operating. In most of the cases discussed below, the Framework Cells need no modification when they are removed although making a few alterations to remove the interaction completely will make the Application more efficient.



3.11. Configurability – Complete Framework Application

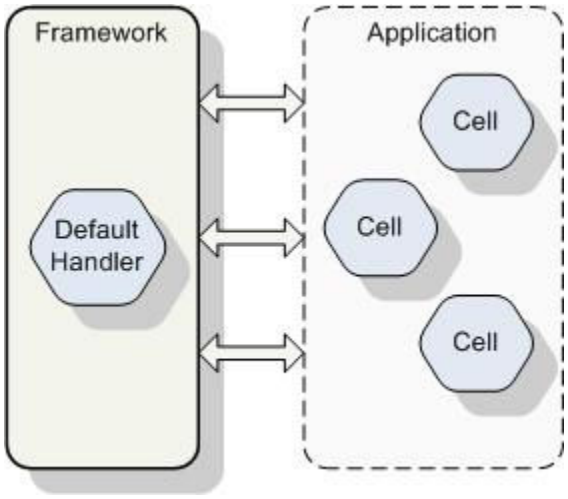
The complete Self healing framework consists of the Mediator along with the Default Handler, Network, Address Server, Repository and File Handler Cells. This allows complete support for applications to be built dynamically, Message Handlers to be discovered at run time and Cell failures to be restored.

Since the Repository and File Handler are only concerned with replacing failed Cells and building new applications they can be removed if the application does not require healing or construction but still wants dynamic addressing over a distributed environment.



3.12. Configurability – Network Enabled Application

If a simple network enabled application is to be designed, it can be built from just the Network Cell and the Default Handler. The Network Cell allows communication and the Default Handler takes care of exceptions in the Messaging process.



3.12. Configurability – Minimal Matrix Application

The most basic Matrix Framework consists of a simplified Default Handler. The Cell would have all its internal addressing structures and network access references removed and automatically respond to all requests with a Message Failed reply. The pared down Framework is ideal for single machine applications that would like to leverage the Cell technology to enable reuse.

3.6. Observations

The design of the Self Healing Framework based on the Software Matrix was discussed in this chapter. There are two important aspects of this development. The first is in the Framework itself which provides a valuable aid to the software construction process. The Dynamic Composition and Self Healing properties can be added to any Matrix application with absolutely change to existing code or design. Both are useful to any developer as they can automate some of the safety mechanisms that distributed applications need. The second observation is the ease with which the framework was built. The Software Matrix allows construction of such systems with very little effort from the developer mainly because of the simplicity with which components can be reused. No information about Cells is required to build and compile new Cells except the name and arguments to be used in the message call. It behaves like a service library as compared with object libraries provided by modern IDE's. The Dynamic Composition feature of the new framework only enhances this property by providing transparency in service location.

Chapter 4

Testing the framework

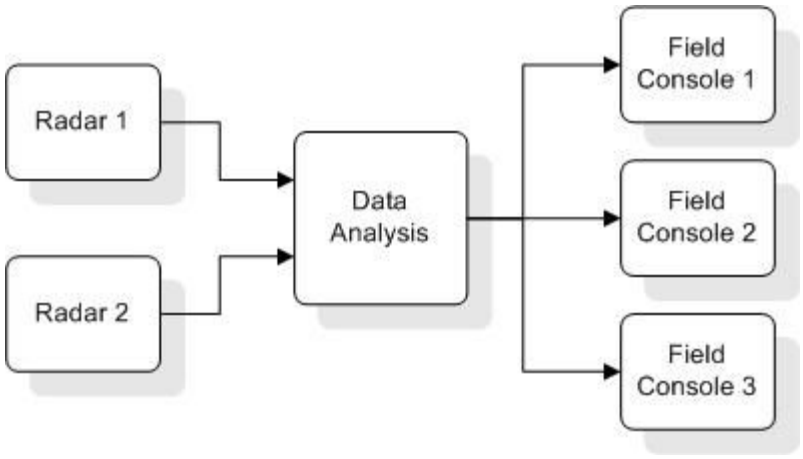
As discussed previously, failure is a condition in which one of the system's Nodes is no longer accessible. This can be due to the failure of the network or the Node itself. When a Node goes down the system loses all functionality provided by the Cells that it contained. This can be restored by the Self-Healing Framework by re-installing the Cells that were lost.

4.1. Framework Test Application

To test the operation of the Self Healing Framework and to simulate a real working environment, we built a Simulated Radar Application. The system was built under the Software Matrix guidelines where each component is a Cell.

4.1.1. The Simulated Radar Management System

The Simulated Radar System consists of three components. A Radar Display which in a real environment is connected to the Radar antenna. This identifies any targets that come within range and displays them on the Console. The second is a Data Analysis component which collects information from one or more Radar Displays and compiles the data to generate target paths and speed information. The last is a Field console unit which displays targets paths local to a specific area as identified by the user.



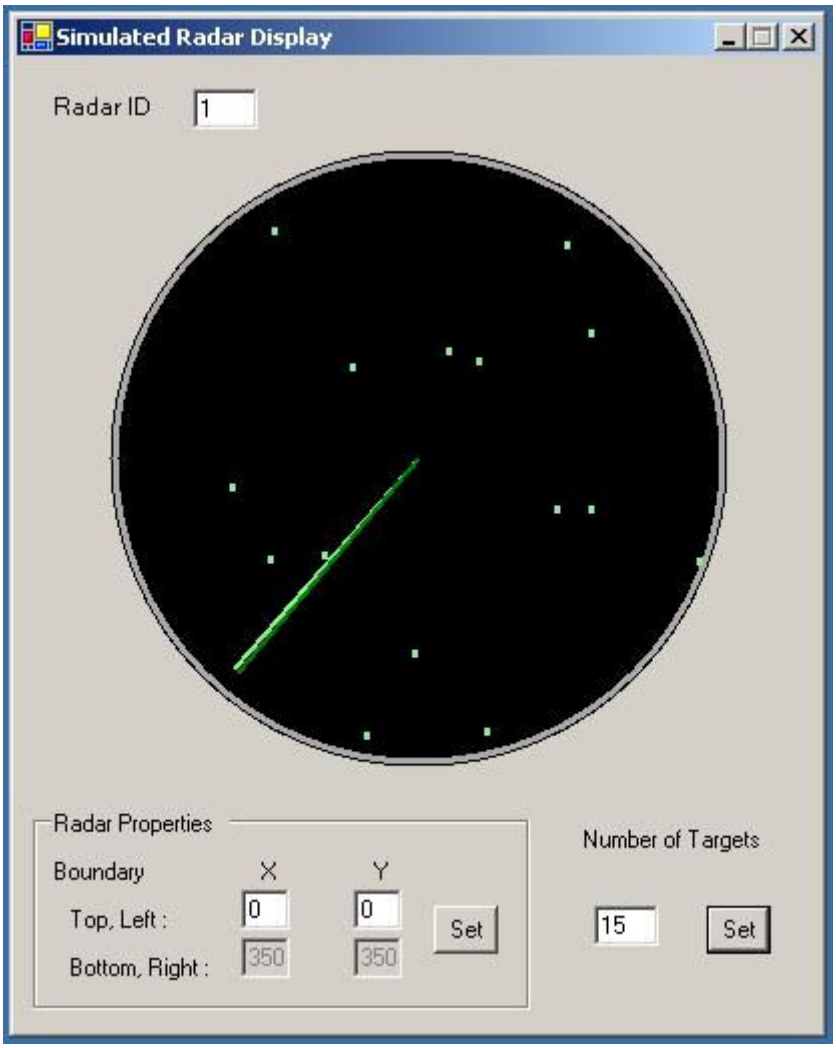
4.1. Simulated Radar Management System

The way the Radar System is designed is very similar to a generic client – server application with the Radar and Data Analysis Components requesting services from the Data and Field Console respectively. This can easily be modified to fit into the Matrix Framework by building each component as a Cell. The application can then be used to test the Self Healing Framework. The Application as built contains the same three Cells – Simulated Radar Display, Data Analysis and Simulated Field Console.

4.1.2. Simulated Radar Display

The Radar Display Cell creates a simulation of a Radar Display that shows targets identified by a physical Radar device. The Cell defines the area covered by the Radar and displays all targets identified in that space. Since a sweep of the radar can only identify the instantaneous position of the target when the sweep was conducted, movement of the targets is depicted as a sequence of points along the path that they are traversing. After each sweep, the position of each target is plotted and forwarded to the data analysis Cell

for processing. Since this is a simulated application, the targets are generated by target objects created in the Display Cell.



4.2. Simulated Radar Display

The Display allows the Radar ID and scan boundaries to be specified to allow mapping to different physical devices. The Number of Targets field is to aid the simulation by specifying the number of targets displayed at any point in time.

4.1.2.1. Design of the Radar Display

The display class consists of a simple Windows form which has been overlaid with GDI graphics. The graphics are generated by overriding the forms OnPaint function to provide custom graphics. The continuous motion of the Radar objects is created by using a thread to call the Invalidate function every 100 milliseconds to repaint the form with the new positions.

The targets displayed on the Radar are generated by the Target Class. Every time the number of targets on the screen goes below the specified number, a new object is created to replace it. The constructor generates a random start location, speed and direction for display. Since the targets must move over the scanned surface, the calculate position function is called on each visible target once every 5 seconds. The newly calculated position is used to redraw the target point on the next paint. If the new point is beyond the Radar's display boundaries, it is removed from the list of visible points to allow the next target object to be added.

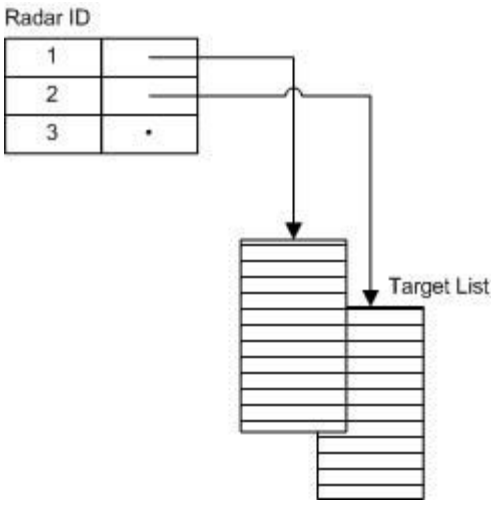
At frequent intervals, the list of points currently visible on the radar display is encapsulated into a Matrix conformant message and sent to the data Analysis Cell. The theory is that since the radar sweeps across the area it is monitoring, it can only identify the current position of the target. The analysis Cell is responsible for calculating the path and speed of the target from the location information.

4.1.3. Data Analysis Cell

The data analysis Cell collects instantaneous location information from different radar consoles and uses them to plot the path of the targets being tracked. It identifies the location points corresponding to each target and calculates its speed and direction. The generated information is collected in a targetpath class and displayed on the Analysis display screen. The path information can be either saved into storage or forwarded to field displays that might require the information.

4.1.3.1. Design of the Data Analysis Cell

The data Analysis Cell collects data from the Radar Consoles and determines target path information. The data is organized in a customized data structure consisting of a hashtable indexing an array of targetpath objects. Each array corresponds to the targets identified by a particular Radar Console. When a new message is received, the targetlist corresponding to the message sender is retrieved and updated with the new data. Current positions of existing targets are updated, new targets are added and old ones that have moved beyond the Radars tracking capability are removed. This information is also updated to the Data Analysis display screen.



4.3. Data Analysis Cell Data Structure

Speed of the target is calculated by measuring the distance between consecutive points. Since the Radar's scanning interval is known, the distance by time formula can be used. In this particular application, the speed of each target is assumed to be constant and is calculated only once but more accurate determination can easily be implemented by calculating speed using the last two detected positions for each calculation. The start location is the first point of the new target detected by the radar. The time of detection is measured at the data analysis Cell as each new target is identified. The direction of the target's motion is calculated using any two points along the path, (x_1, y_1) and (x_2, y_2) with the formula $(y_2 - y_1) / (x_2 - x_1)$.

4.1.3.2. The Data Analysis Message Types

The Data Analysis Cell receives one type of Message. These are sent by the Radar Console and used to collect updated target information.

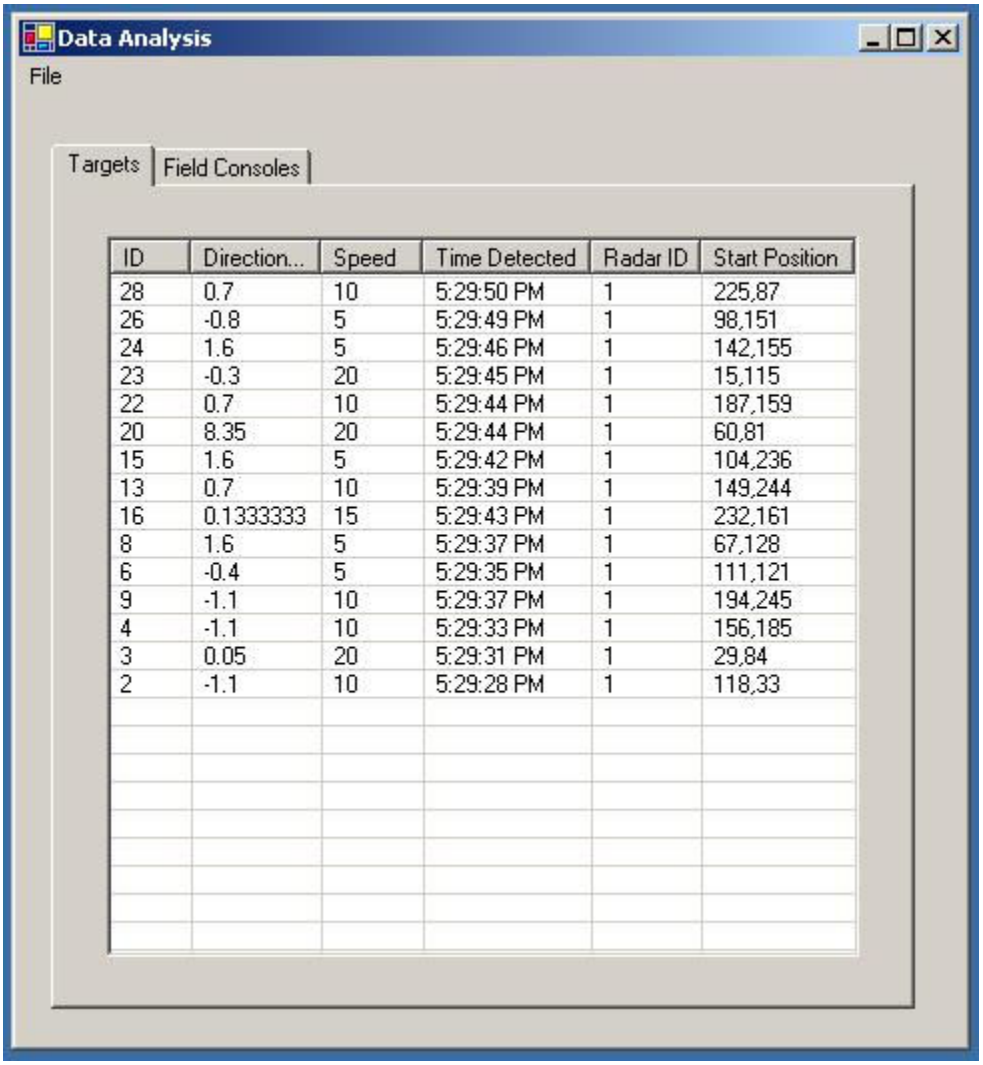
Matrix.Application.Radar.AnalyzeTargets– Used to update Target information.

Parameters

1. Radar ID – an Integer value containing the ID of the Radar sending the message
2. TargetList – a hashtable containing the list of current positions of each identified target. The table entries are indexed by target ID's generated by the Radar.

Return Values

None

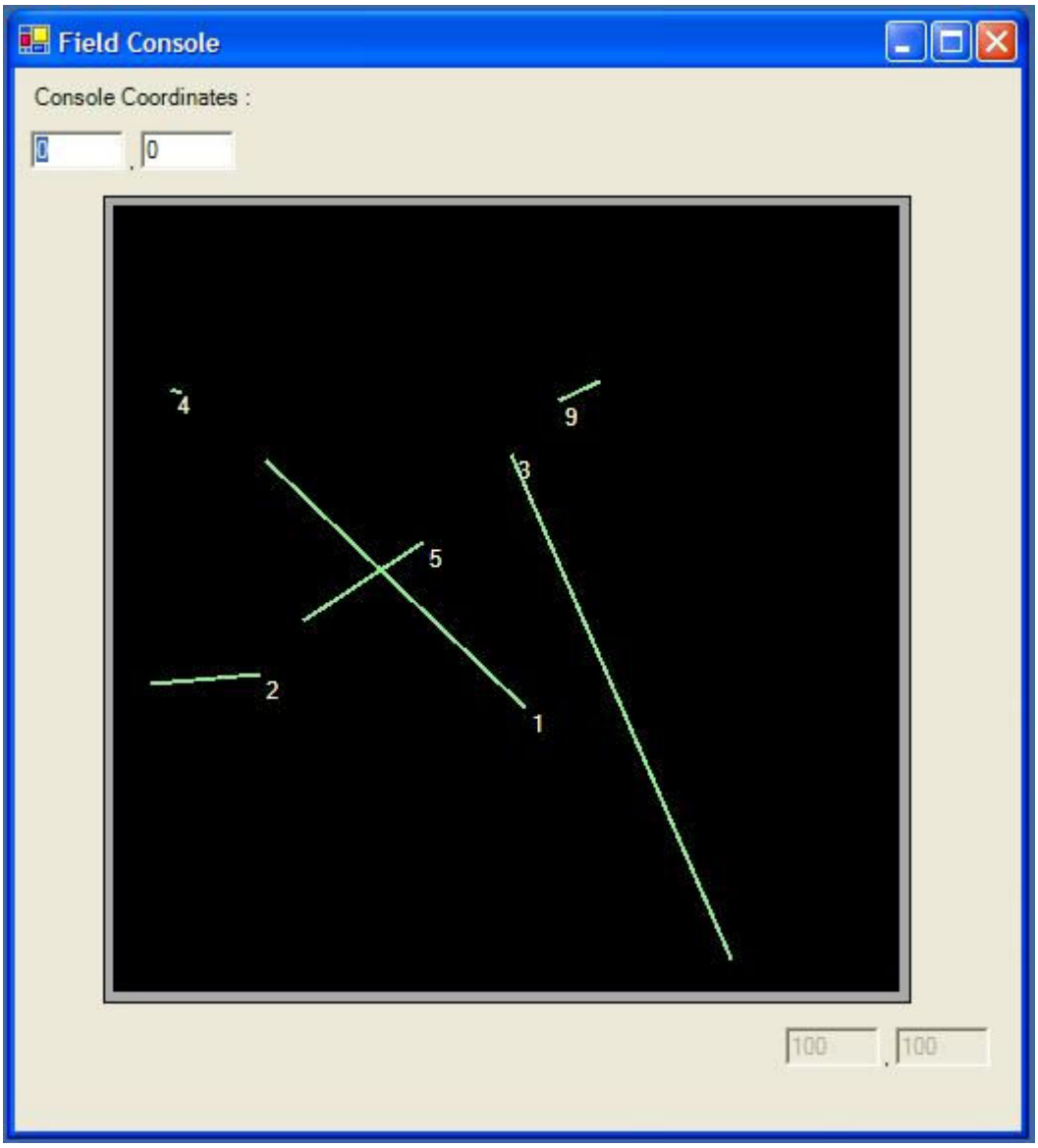


4.4. Data Analysis Display

4.1.4. Simulated Field Console

The Field Console Cell is intended to simulate Field console units that receive target information from the Radar system. Each console displays information relevant to its location. Field displays usually cover a smaller area than Radar units but might display more details. In the test program, each Radar unit covers an area of about 300 sq miles while each Field unit only covers 100 sq miles. For this reason, the data gathered from the radar must be projected onto the Field unit's boundaries. The speed and direction

calculations from the data analysis cell are used to project the targets path onto the unit's display.



4.5. Simulated Field Console Display

4.1.4.1. Design of the Field Console

The Field Console Unit contains a display similar to the one on the Radar Display. It collects the target information forwarded from the Data Analysis unit and displays the path followed by all targets within its specified boundaries. Since the unit may be moved around, its boundaries can be adjusted by entering the current location on the display. The unit behaves like an intelligent display filtering out all targets that do not fall into the monitored boundaries. When a new target is identified by the Analysis Cell and its information is received, the Field display first examines its start location. Two options may result

- If the target is within the unit's perimeter, it is displayed on the screen and its path is traced until it goes beyond the device's boundaries.
- If the target's initial position is outside the perimeter, its direction is examined. If it is moving away from the unit, it is discarded. If it is moving toward the unit, the information is saved until the target comes within the units boundaries. It is then tracked across the display till it goes beyond the borderline.

4.1.4.2. The Field Console Message Types

The Field Console Cell receives one category of Message sent by the Data Analysis Cell to inform the Console of newly identified targets.

Matrix.Application.Radar.Console – Used to update Target information.

Parameters

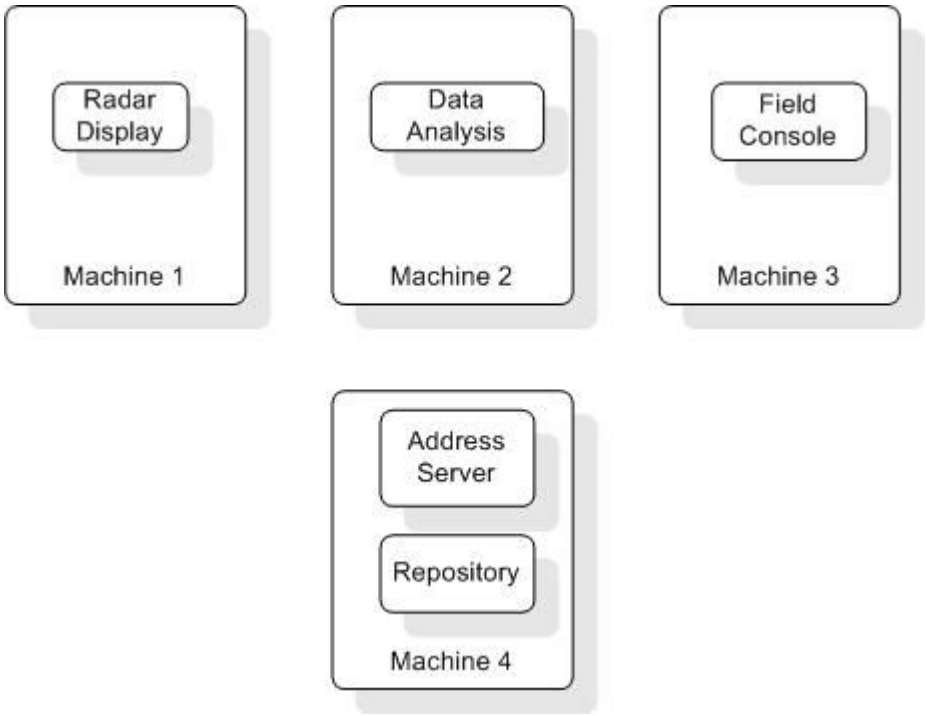
1. Target – an object of the TargetPath class detailing the identified target.

Return Values

None

4.1.5. Failure test

The setup of the Radar system was as follows. The Radar Display, Data Analysis and Field Console Cells were installed into three separate nodes. Each node contained the Default Handler and Network Cells that are described in the previous chapter. A fourth node contained the Address and Repository Servers. The three Radar Application Cells were also stored into the Repository database. The Config files were set for each of the machines and the load balancing option was set to load new Cells into the Calling Node.



4.6. Test Machines Setup

When the application was run, the three displays came up and targets displayed on the Radar screen were picked up by the Data Analysis Cell and their paths were displayed on the Field Console. Once the application was successfully running, machine 2 running the Data Analysis Cell was shut down. After a brief delay, The Data Analysis window appeared on machine 1 alongside the Radar Display and continued sending target information to the Field Console. Since the Analysis Cell was restarted, the target id's displayed were reset. The field console did not lose any information due to the Data Analysis failure.

The second test was to shut down machine 3 containing the Field Console. After a brief delay, it too appeared on machine 1 in response to a message sent by the Data Analysis Cell. Since the new Cell does not retain state information, previous data was lost but the startup requested all current data from the Data Analysis Cell and restored all relevant information.

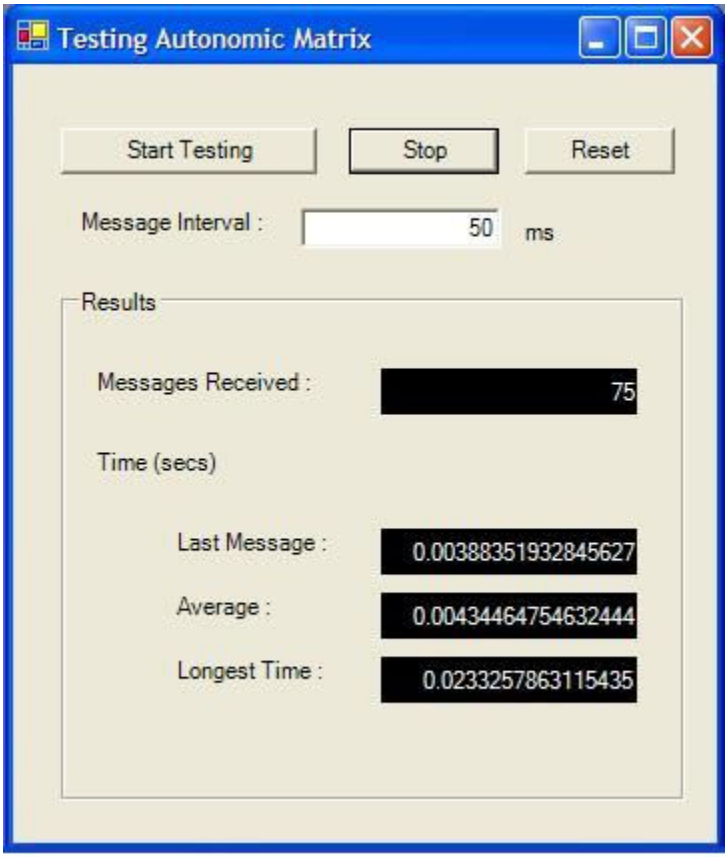
4.1.6 Failure Recovery

Shutting down machines 1 and 2 caused an abrupt asynchronous failure of the communication channel between the Cells. The brief delay in loading the new Cells was due to the time out mechanism. There are two timeout delays. The first one, at the Network Cell is the time required to establish that a connection has failed. The second is the timeout of the Send function. This is essential for synchronous communication to prevent a particular Cell from waiting indefinitely if failure occurs. The next message from the client, either Radar or Data Analysis, failed at the network cell since a

connection could not be established. It was forwarded to the Default Handler which in turn made a request to the Address Server. The Server determined that the node has failed by comparing the Message Type and Destination to the entries in its table. It then contacted the Repository Server to reload the Cell that failed. In this experiment, since the load balancing option was selected, the new Cell was installed into the client node. The Address Server returned the new address to the client and the application resumed operation by sending new messages to the locally installed cell.

4.2. Timing the Framework

To estimate the efficiency of the recovery process, we set up a simple operation to estimate the time taken to recover from failure of a Cell. We consider a simple client server system. The server consists of one Cell that handles messages of type Matrix.Application.Test.SendMessage. The client is a GUI based Cell that continuously sends test message to the server and waits for a reply. It continues to execute this loop with a small timeout in between each message until stopped by the user. The client also times the message send-receive intervals and keeps track of the longest time taken by a single message and the average time taken.



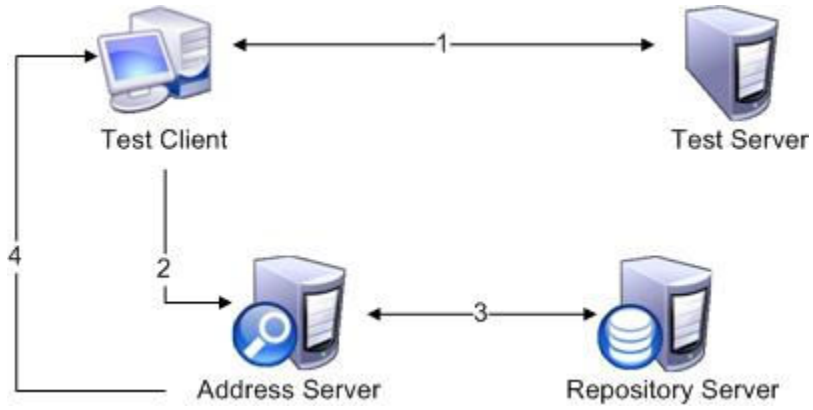
4.7. Timing Test Display

Two different tests were conducted

- The first was by running the application on a single machine. The average time taken by a regular message was compared to the time taken during Dynamic Composition. This is where the required Cell in not already installed and has to be loaded from the repository before the application can start. Self Healing was not tested since all Cells are loaded in the same process and cannot individually fail. The time taken by a regular message after composition was also noted.
- The second test was run on multiple computers. The test client, server and the Framework components were all installed into different machines and the same tests were carried out. In this test the time required by a regular message was compared to

the time required when no Handler existed at application startup (dynamic composition) and when the test server failed. Failure was induced by shutting down the server mid-operation. Time taken by a message after failure was also recorded.

To provide reasonable comparison before and after failure the times for regular messages were recorded using the 10th message after startup or recovery.



4.8. Timing Test Setup

Test 1 – Local Application (time in ms)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Regular Message	3.1989	3.8253	3.1800	3.2134	3.1074	3.305
Dynamic Composition	667.85	587.91	578.31	583.71	580.54	599.66
Regular Message after Composition	2.7194	3.3813	4.0016	3.2049	2.7028	3.202

4.9. Local Machine Timing Results

Test 2 – Distributed Application (time in ms)

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Regular Message	12.256	11.996	12.196	12.009	13.052	12.301
Dynamic Composition	429.34	404.30	436.65	408.10	411.00	417.89
Recovery from Failure	3757.18	3866.30	3605.30	3849.02	3942.93	3804.15
Regular Message After Failure (Load Balanced)	2.510	2.791	4.023	3.414	2.606	3.068
Regular Message After Failure (Designated Server)	12.873	12.224	12.196	12.885	12.122	12.46

4.10. Distributed Setup Timing Results

4.2.1. Results

In the local machine tests, we can see that the time taken by a message after composition is similar to the time taken by a regular message. This is because the application that is constructed works exactly like applications that have been put together before startup. Although the message that initiates the construction process is slow, it is a one time process and the applications operates at normal speed once setup is complete.

The tests with the distributed system are closer to real system operations. The first row show the time taken by an average message sent between two matrix nodes. This is due to both the matrix messaging structure and the network delay. The second row shows

the time taken to install a new Cell into the system. This time is the same as the dynamic construction time in the first test. The third row shows the time taken to recover from a failure. After the test server fails, the client must first detect the failure, request a new Cell from the Address server, wait for the new Cell to be installed and then forward the original message. This time is significantly greater than all the others – an average of close to 4 seconds. The reason for such a long delay is mostly due to the timeouts that are set for detecting failure. Although this time can be reduced, too small a timeout can cause loss of messages especially in slow network conditions. This time is also a one time cost after which the system returns to normal.

There are two different post failure times recorded. The first is when the load balancing option is set. In this case, the new Cell is loaded into the same Node as the caller so the new message times is actually faster since the messages no longer need to be transferred over the network. This new time is close to the time measured in the first test on a single machine. In the second test, the new Cell is loaded to a designated server. This was set to the node containing the address server. After a similar failure recovery time, the messages take an average time which is identical to the original regular message time.

4.3. Observations

The results from the tests show that the operation of the system after failure recovery is exactly the same as it was before the failure occurred. This is to be expected since the Cells loaded work exactly like they did before the failure occurred. This

property is one of the greatest strengths of the self healing framework built on the Matrix Framework. It can restore the system to the same level of operation as existed before the failure occurred.

The experiment also reveals some of the drawbacks of the framework that was created. The first of course is that all programs that use the Self Healing framework must be Matrix enabled, i.e. they must consists only of Cells working under the Matrix Mediator. This is not such a drawback as it looks since converting an existing application into a Cell based design is not very difficult. It is only required that existing components be encapsulated in a Cell interface. The larger problem is the disability of the framework to maintain state. It may be important to many applications that the state information maintained by a component before failure be restored. While this property has not specifically been addressed by the framework itself, it can to some extent be designed into the Cells that need it. As an example, the address server is able to restore its address table by contacting other Cells and retrieving their capability lists.

Chap 5

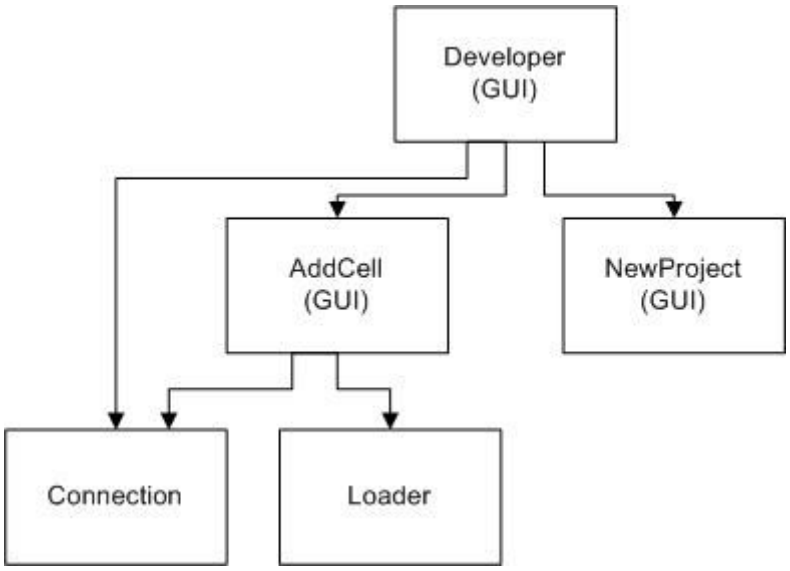
Matrix Developments

In addition to the Self Healing Framework, we experimented with the Matrix Framework to build support tools for developing Matrix Applications.

5.1. Matrix Application Developer

The first experiment was to develop a tool to facilitate building Matrix Applications. The original idea behind this research work was to build a complete Framework for Matrix Applications. An important part of this is to build a development environment. The IDE supported by a complete library of Cells will allow Applications to be built just by putting these Cells together. No coding is required to connect Cells together and testing is simplified since only pre built and tested components are used. The IDE designed as part of the research work allows designers to pull Cells from a repository and create working applications.

5.1.1. Developer Classes

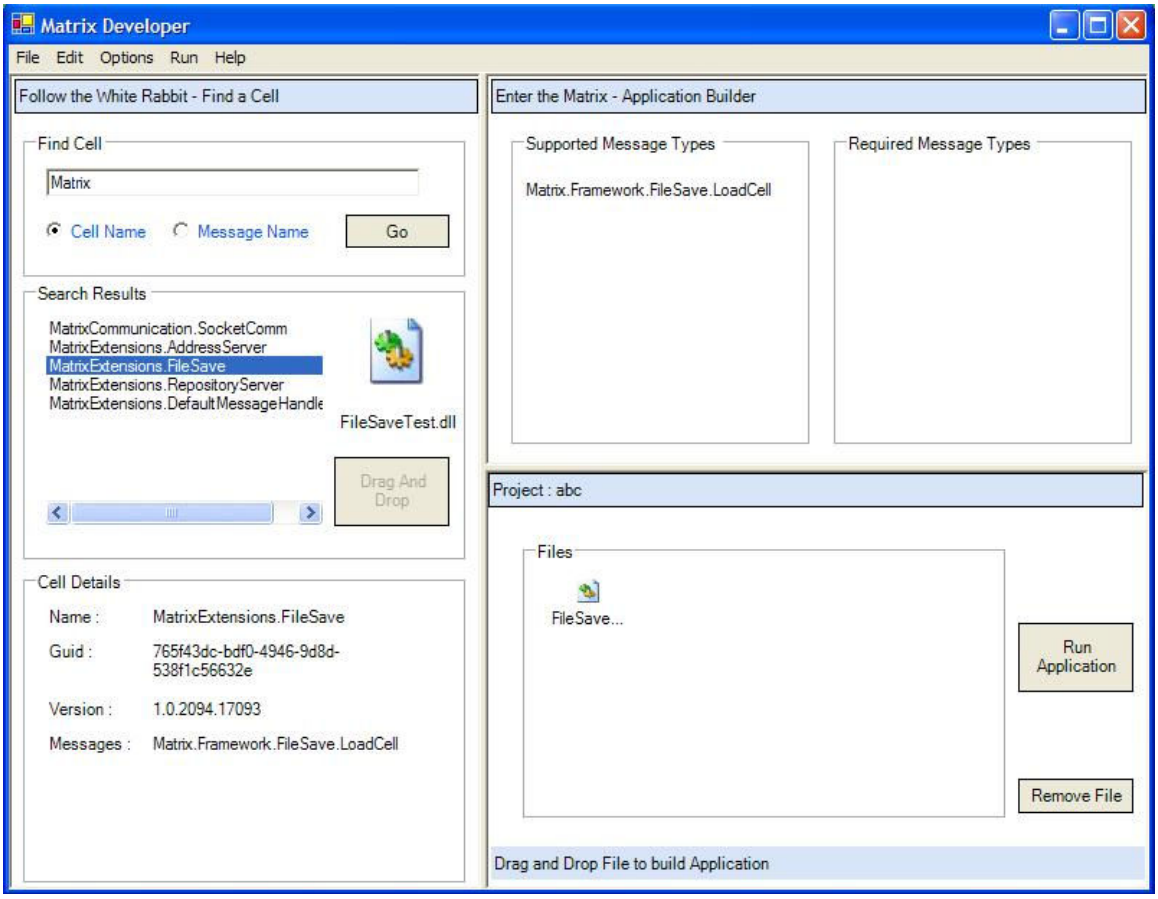


5.1. Matrix Developer Class Diagram

The Developer consists of three UI objects and two support classes –

Developer

The developer is the main User interface Class and also the executive for all operations.



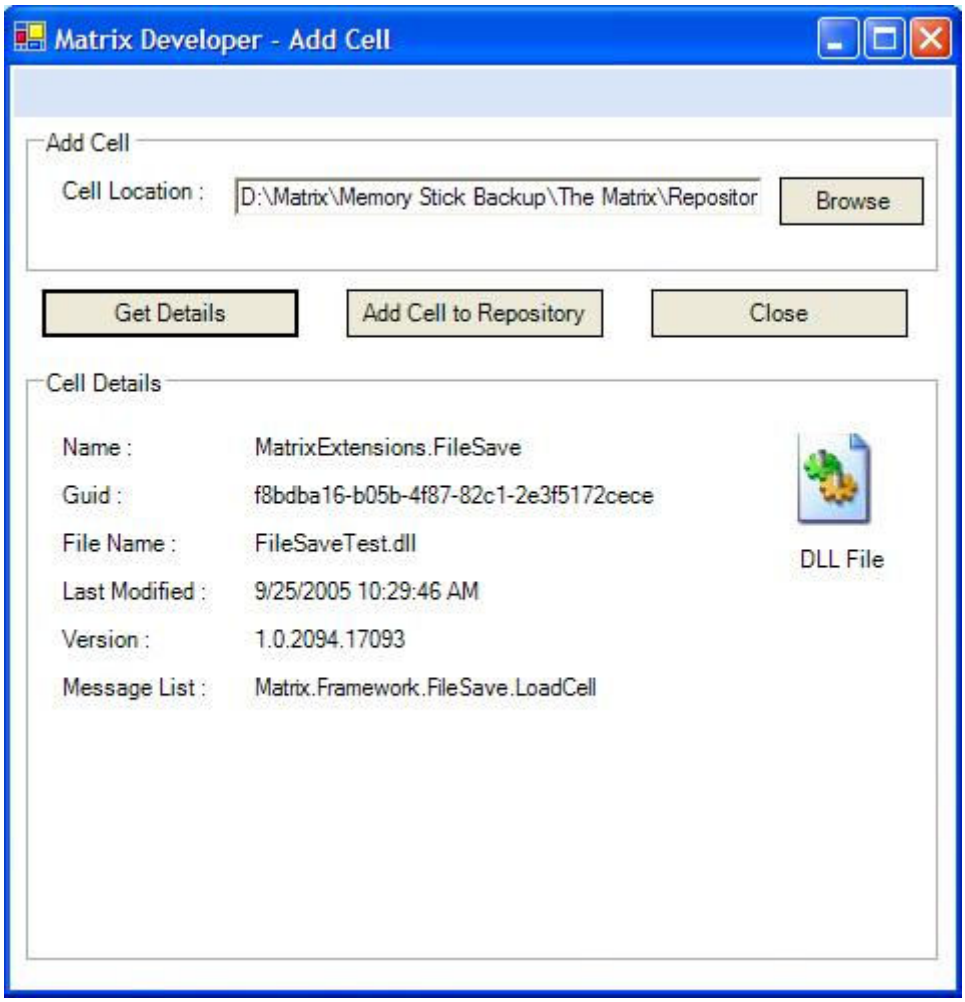
5.2. Matrix Developer User Interface

It allows the User to create or open Projects, search for Cells from a database, and add or remove Cells from the project. As each Cell or project is selected, information describing that particular element is displayed. The Developer allows the user to add new Cells into the database by invoking the AddCell UI form.

AddCell

The AddCell form contains a browser button to help locating the Cell to be added. Once a Cell has been chosen, the Get Details button displays the file name, the Cell name, the version number and the Message Types that the Cell can handle. The Add Cell

button uses the connection class to store the collected information and the selected file into the database.



5.3. Adding new Cells

NewProject

The NewProject Class is a simple UI that collects the project name and storage path from the user and creates an environment for the Matrix application. This consists of copying the Matrix.dll file, the Matrix Executive and the Config file. An empty Plugin folder is created and is filled as the project is built. Information about the project is stored

in XML format in a file with the project name. This XML file can be used later to load the created project and make changes as required.

Connection

The connection class is the link between the Matrix Developer and the back end database. It uses the .Net ADO classes to create a XML schema and storage files and uses these to back up the generated data. A class named CellData is used to represent the data contained in each Cell and transfer this information between other classes.

Loader

The Loader class operates in a similar fashion to the Matrix Loader. It retrieves the specified file and uses reflection classes to create a Cell instance. The instance is queried to generate the required information about the capabilities.

5.1.2. Conclusions

The goal of this experiment was to create a Development Environment where Applications can be built with little effort. As stated before, we would like to build a framework of Matrix Cells that support all basic functions required by any application. With this in place, only User interfaces need to be customized to build specific applications. The required Cells can be put together using this Developer tool and new applications created quickly. Applications can be customized at will, adding and removing Cells as required.

The Developer is a tool that can be used by any person, even one with little knowledge of programming or software development. With a list of well documented

Cells, each containing a description of their functions, building software applications becomes an extremely simple task requiring a very short learning curve.

Chapter 6

Conclusion

In this chapter we try to summarize the concepts and implementations that have been discussed so far.

6.1. Reducing complexity in systems

As applications become more and more complex, it becomes necessary to deal with this complexity to simplify it in some fashion or it becomes incomprehensible. As complexity of Applications grows, new technologies evolve to enable them. The problem is that some of these technologies are in themselves involved and difficult. It requires highly skilled people to understand them. Distributed object technologies like COM can be considered one of the great enablers of Microsoft's original vision of "a computer on every desk and in every home" as described by its chairman Bill Gates [14]. They allowed applications to be modified without requiring recompilation. This was something that could be practically automated, as updates install themselves without requiring users to know the details of the internal operations. But there is one major disadvantage to this technology, developing applications is an elaborate process, beyond the ability of all except experienced developers. What is going to happen when programs become larger and more distributed? Now, when managing and maintaining applications is becoming more and more difficult, IBM's Autonomic systems and Microsoft's Dynamic Systems Initiative plan to follow the same path by automating certain processes. But while automation certainly works, designing programs to enable automation is not as simple as working on ordinary programs. Here again, special skills and training in these

technologies are required. Functionality expected of software will only continue to grow. And if technologies grow more complex, design and developing new systems will progressively become more difficult.

In the course of this thesis research, we explored different ways to reduce effort required to developing new applications. Two concepts were discussed, reducing complexity of tools and automation of basic services. Neither of these are new ideas but in both we explore improving existing technology by using the Software Matrix.

To reduce complexity, we propose the Software Matrix Framework. There are three things to recommend the Matrix in this context. The first is the original concept behind the Matrix itself - reuse. When it becomes geasy to reuse code from one project to another, it reduces time and effort required to develop new applications. The second idea comes out of the first – raising the level of abstraction. As existing Cells can be combined with new applications so easily, applications can be built just by putting Cells together. If a sufficient number of Cells providing all basic services can be developed, developers can work at the component Cell level instead of using object libraries which are much smaller in scope. In this context, we built some generic Cells like the Network Cell discussed in Chapter 2 and a generic File Handling Cell that can be used in any application. While no formal usability study was conducted during development, we constantly evaluated what effort is required to work with existing Cells. Integrating Cells was found to be extremely simple as services can be requested just by naming the messages and supplying arguments. The only drawback was that there is no way of knowing message types,

arguments and return types each Cell provides unless suitable documentation is provided. Since this project was well documented, it should not be a problem for anyone to use the Cells generated as a result. The third reason to endorse the Matrix is the configurability that the framework provides. In the introduction we had discussed simplifying applications by stripping them down to the absolute essentials. A simple application should not have to deal with a framework that enforces a multithreaded, distributed environment. In this project, we put a lot of effort into making the Matrix as configurable as possible to allow it to adapt precisely to the needs of the application being developed. The design of the Network Cell is an example of this work. The Cell is designed to be replaced with any communication technology as required or even removed altogether if network access is not needed. The configurability of the self healing framework was discussed in detail in section 3.5.

We therefore feel that the Matrix is an ideal technology to help reduce complexity. To further enhance its repertoire, we looked at the second concept – automation. Self Healing is an important part of the automation concept. We implement a Self Healing technology for the Matrix and at the same time evaluate the effort required in creating the service. The details of the Framework that was developed, form the main focus of Chapter 3 and some tests conducted on the Framework are described in Chapter 4. The ease of building this framework is evident in the simplicity of the designs that are described. A byproduct is the simplicity of adding the Self Healing property to any Matrix application. All that is required is to add the Address Server, Repository, Default and File Handler Cells into an application, even at runtime, and it becomes Self Healing

enabled software. It is not a perfect solution as there are some drawbacks like lack of support for preservation of state but it is ideal for any service for which state recovery is not an issue. The second advantage, the new framework brings to the matrix is the dynamic construction property. Although the ability to add new Cells into an existing application already existed, Dynamic Construction automates the process by adding a discover mechanism. When any new service is required, this property allows it to be installed and accessed in a way that is completely transparent to the requesting component. The tests conducted in section 4.2 show that the time taken to access a dynamically loaded Cell are almost exactly identical to the time required by a pre loaded one.

In addition to these developments, some changes were made to the Software Matrix developed previously, to make it more efficient and enhance its features. Using a directory watcher to monitor Cells is discussed in section 2.2.1. The Mediator's Message Handling component was optimized and the synchronous message passing mechanism was completed. These are reviewed in section 2.1 that examines the Software Matrix.

6.2. Conclusion

In the thesis, we showed that

1. Simple Self Healing is possible. The combination of the Software Matrix and the Self Healing Framework allow applications to add the self healing property with little effort.

2. Dynamic Construction works effectively. Applications can be built dynamically even ones that are distributed over a Network.

3. Reuse can be extended to the level of writing only customized UI's to build new software. The experiment in Chapter 5 showed that complete applications can be built just by putting Cells together.

These conclusions support our statement that software development processes can be simplified and yet allow us to build complex applications. We focused on two methods, making Framework tools easy to use automation of important support systems. The Matrix was identified as a suitable technology to support these mechanisms and we modified and built on the framework to explore its usefulness. We found that Matrix and the Self Healing Framework have the potential to be powerful tools to aid developers. Combining the Matrix technology with simple and elegant design allows many tasks to be programmed efficiently and effortlessly.

6.2. Future work

Some ideas for future work deal with modifications that will affect both the Self Healing Framework and the Software Matrix itself.

Multi-threaded Server Cells

The design of each Matrix Cell includes a built in Message queue which is accessed by a Process function. Since the function runs on a single thread, each new

message must wait until the previous operation is completed. For high volume cells like the Address Service, this mode of operation may be too slow. To help increase processing speed, a thread pool can be used instead of a single thread.

In addition to the speed issue, an observation during development indicated that when cells call each other inside the Process function; it is possible to put them into deadlock since the message will keep waiting in the queue. Since synchronous message passing has a timeout mechanism, deadlock may be prevented but messages will be lost without any obvious reason. The thread pooling mechanism can help in this situation too.

In addition to speed of the cells operations, the Mediator operations can themselves be enhanced to operate at greater speed. The Mediator follows the same design with an input queue and a single processing thread.

However, it is important to note that this additional speed may not be required for most applications and should only be provided as an easily configurable option.

Sophisticated Network Cells

The Communication framework built into the Matrix can be enhanced in many ways. Since new implementations only need to follow the same Message format, the entire Network infrastructure can easily be updated by replacing the Network Cell. Other schemes like Remoting or Web Services can be used in place of socket communication. Encryption and authentication schemes can be added to allow greater security.

Multiple Service Providers

The design of the Self Healing Framework assumes that only one instance of a service (or Cell) exists in the application at any given time. If more than one instance of a Cell exists, all its messages are redirected to the Cells that still survive. Also the delivery of messages that do not contain explicit addressing information can be ambiguous in a multiple cell application. The framework was implemented this way since it was assumed that services that rely on location would no longer be available if the node failed. Physical failure of the machine cannot be restored. However it may be required in some cases that multiple service providers exist in the system. Designs for implementing this concept may be explored. The Matrix Framework supports multiple Cells through a cloning mechanism which can be extended into a distributed environment.

References

1. Software metrics and reliability, NASA SATC, http://satc.gsfc.nasa.gov/support/ISSRE_NOV98/software_metrics_and_reliability.html

2. Ghosh, Riddhiman, 2004, The Software Matrix: An Architecture for Software Salvage.

3. Self Healing definition, Whatis.com, http://whatis.techtarget.com/definition/0,,sid9_gci858972,00.html

4. Autonomic Computing, Webopedia.com, http://www.webopedia.com/TERM/a/autonomic_computing.html

5. Frye, Colleen, 2003, Self Healing Systems, ADTMag.com, http://www.adtmag.com/article.asp?id=8132

6. About IBM Autonomic Computing, IBM Website, http://www-03.ibm.com/autonomic/selfmanage.shtml

7. Worden, Daniel and Chase, Nicholas, 2004, A quick tour of Autonomic Computing, http://www-128.ibm.com/developerworks/edu/ac-dw-ac-intro-i.html

8. New to Autonomic Computing, IBM Website, http://www-128.ibm.com/developerworks/autonomic/newto/

9. Cutlip, Rob, 2005, Self Managing Systems, http://www-128.ibm.com/developerworks/autonomic/library/ac-selfo/index.html

10. Towards an Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications, 2004, Intel Technology Journal, Vol. 8, Issue 4, http://developer.intel.com/technology/itj/2004/volume08issue04/art03_autonomic/p07_ibm_toolkit.htm

11. Dynamic Systems Initiative Overview, 2005, Microsoft Website, http://www.microsoft.com/windowsserversystem/dsi/dsioverview.mspx

12. Newmarch, Jan, A Programmers guide to Jini Technology, 2000, Apress publishers, 1st edition.

13. Garlan, David, Cheng, Shang-Wen and Schmerl, Bradley, 2003, Increasing System Dependability through Architecture based Self Repair, appears in Architecting Dependable Systems, 2003

14. Gates, Bill, During a talk titled ‘Unleashing the power of creativity’, 2005, part of the transcript is available at http://www.npr.org/templates/story/story.php?storyId=4853839