TCP/IP CONNECTION MANAGEMENT USING A REAL-TIME DEVELOPMENT TOOL

ANN GRAY*, R. WHITELOCK*, E. PEYTCHEV** and D. AL-DABASS**

** Siemens Communications*
Technology Drive, Beeston,
Nottingham, NG9 1LA.
ann.gray@siemens.com

*** School of Computing & Mathematics*
The Nottingham Trent University
Nottingham, NG1 4BU.
evtim.peytchev@ntu.ac.uk

Abstract:The Central Integration Unit (CIU) is a major component of a system being developed to provide voice and data communications between mobile radio users and fixed terminal users. The CIU has several different TCP/IP interfaces, both external and internal to the CIU, with varying characteristics. Some use permanent connections whilst others use transaction-based connections; some are clients or servers, others incorporate both client and server operation. This paper looks at the issues behind the design of the connection management aspects of the CIU and then describes the implementation of the connection management software within the CIU.

Keywords: TCP/IP, real time tools

1. INTRODUCTION

The aim of this project is to demonstrate a method of managing several different types of TCP/IP interface within a single application.

The Voice Radio System provides a means of intelligently routing calls between mobile users and fixed terminals based upon current location of the mobile user. This system uses a GSM network to provide the mobile user switching capabilities, in conjunction with a Central Integration Unit (CIU) to provide the voice and data routing capabilities. Figure 1.1 illustrates the basic functionality of the Voice Radio System.

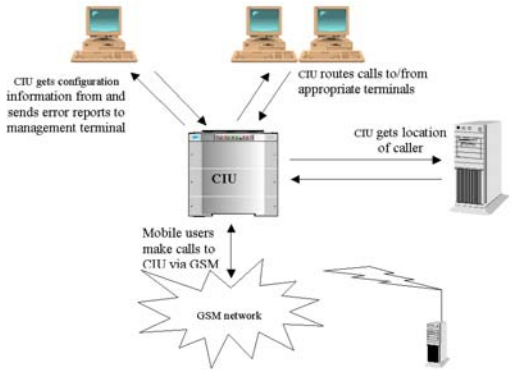


Figure 1-1 Overview of Voice Radio System

The CIU is a combined hardware and software development consisting of many Commercial off the Shelf (COTS) sub-components, including a Realitis PABX, along with bespoke CIU application software. The application's primary function is to control the routing of calls to and from the fixed terminals, which it does using the CTI (computer telephony integration) capabilities of the PABX. In order to perform this role it uses application level communications to the other external components of the system. These communications use a proprietary system messaging protocol, which has been specified by the customer, via TCP/IP socket connections.

All of these interfaces (five in total) use uni-directional, transaction-based connections. This means that the connection is established whenever a single message is required to be sent on that interface and closed again immediately after the transmission. Three of the links are one-to-one, the others having multiple remote end-points.

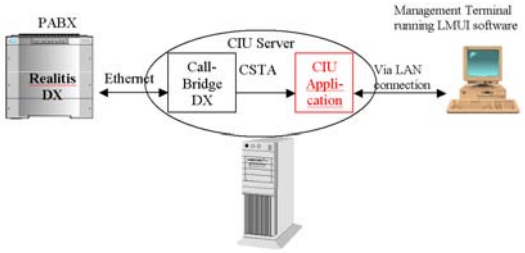


Figure 1-2 CIU Internal Interfaces

In addition to the external interfaces, the CIU application also has two further internal TCP/IP interfaces to manage, as illustrated in Figure 1.2 The first of these concerns the CTI interface to the PABX (via CallBridge DX software, an existing Siemens Communications product). This utilises CSTA, the European standard CTI API, as defined by the standards ECMA-179 and ECMA-180 (ECMA, 1992). The underlying TCP/IP connection

(c) SCS Europe BVBA, 2003 Copyright (c) Siemens AG 2003

is permanent, one-to-one and bi-directional, with the CIU application acting as the client.

The final interface allows the CIU Local Management User Interface (LMUI), another piece of software specifically designed for the CIU development, to connect to the application for administration and configuration purposes. This is also a permanent, one-to-one, bi-directional TCP/IP connection, but in this case the CIU application acts as the server.

In order to provide the call routing capabilities required for the Voice Radio System, the CIU must be developed in such a way as to manage its multiple TCP/IP connections in the most effective manner in order to support the level of performance required by the customer. The aim of this project is to demonstrate a way of managing these different types of TCP/IP interface.

2. THE CURENT SYSTEM

For a prototype CIU development, some of the TCP/IP interfaces were developed, namely the two internal interfaces plus one of the external interfaces. The design of all three interface components was based upon the CSTA interface component (known as the CSTAIC) from the CallBridge DX software, with the following adaptations:-

1. The CIU CSTA interface component was modified to be a client, rather than a server.

2. The CIU LMUI interface component needed to be a server, therefore no modification was required.

3. The external interface component was considerably altered to handle transaction-based operations for both client (for outgoing messages) and server (for incoming messages).

Therefore the prototype software contained three different interface components, all with the same basic structure. This increased the amount of testing required, and also led to maintainability issues. Any fault that was found in one interface component had to be checked and potentially fixed in two other places. In the final CIU development this situation would be further compounded with the addition of four further external interfaces. It is not practical to maintain seven different interface components. Therefore it is desirable to have as few interface components as possible, with the ideal being a single, generic connection component.

The second concern regarding the design of the connection interfaces is whether to re-use any part of the prototype software as the basis for the phase 5 interface components. The prototype software, including the interface components, utilises a proprietary framework, which was developed by a third party, for which Siemens Communications owned the libraries, but not the original source code. This framework provides the following capabilities: - threads, task scheduling, inter-process communication and diagnostic logging, as well as socket handling classes.

The framework was found to have several limitations. Firstly, link errors were encountered when building the CIU application using Microsoft Visual C++ 6.0. It is thought that the framework library was built using an earlier version of Visual C++ and includes versions of the Microsoft C++ libraries that are incompatible with those linked in using version 6.0. During the prototype development, reverting back to Visual C++ 5.0 solved this problem, but this is not an ideal long-term solution, since support for earlier releases of Microsoft products is not guaranteed indefinitely. Although it may be possible to acquire a version of the framework library that is compatible with Microsoft Visual C++ 6.0, this is also limiting since there is no guarantee that it would be compatible with future versions. Furthermore this may take too long to acquire and would generate an unwanted dependency on external developers.

Secondly, there were doubts within the engineering department regarding the performance of this framework under heavy traffic, although this has not been proven one way or the other. Although the prototype CIU met its required load, the traffic requirements for the final product would be much greater. The project time constraints mean that the risk is too high to invest time in using this framework only to find later that it is not effective.

The final limitation is that, since Siemens Communications do not own the source code, there is no control over its operation. Any modifications to the framework behaviour have to be made by adapting the custom software in the application to achieve the right results. This is very restrictive and, in some cases, impossible.

For all these reasons, it was decided that the framework used for the prototype would not be re-used for the main CIU development.

3. THE NEW SYSTEM

In order to determine a strategy for the design of the connection management software, several options were considered. These options had to be examined in light of a customer requirement mandating the use of Rational Rose RealTime (RRRT) as a development tool.

Rational Rose RealTime is a modelling tool which uses an extended form of UML to enable the modelling, implementation, building and debugging of complex real-time systems. The RRRT modelling language includes support for concurrent objects, allowing communication between them via ports using user-defined protocols. The dynamic behaviour of these capsules can also be modelled using state diagrams. Further information about these concepts can be found in the Rational Rose RealTime Modeling Language Guide (Rational 2002).

3.1 Pre-Existing CallBridge Framework

The majority of the CallBridge software utilises a different framework to that used for the CSTA interface. It was originally developed for CallBridge's predecessor and has been enhanced over many years into a robust and reliable platform on which to build an application. This framework has also been used in other in-house developments and is an obvious first-choice candidate for use in the CIU. It provides a round-robin task-based scheduler, including inter-task communication, plus a message transport layer that allows non-blocking TCP/IP socket management within a single-threaded application.

Although, at first, the CallBridge framework appears to be a good option, there are some disadvantages to weigh up.

1. The message transport layer currently only supports permanent connections, and would therefore require substantial modifications to allow its use for transaction-based connections.

2. Since RRRT has its own built-in mechanisms for task-scheduling and inter-task communication amongst other things, it will provide the framework for the CIU application. The only part of the CallBridge framework that is required is the message transport layer. However, the CallBridge framework is not structured in such a way as to easily extract the message transport software on its own. Effort would be required to repackage it for use within RRRT.

3. Finally, the CallBridge framework is written in 'C' and is likely to be difficult to maintain by a team whose main skills are in C++ and Java. This would not have been a major issue if the framework was suitable in its present form, but since points 1 and 2 imply a substantial amount of rework, this would add a greater risk to the project.

3.2 Single Generic Connection Manager

This option involves the development, from scratch, of a generic connection manager component, using the RRRT framework. This component would be able to handle both permanent and transaction-based connections; client or server operation, or both; and either single or multiple remote end-points.

The advantage of this approach is that there is common code for all interfaces, so modifications only need to be made in one place. This would also reduce the time required for testing the connection management software and it carries less risk, both to the project timescales and the quality of the resulting software.

Preliminary investigations into a possible design, however, indicate that this is not straightforward, and that the resulting code could be very complex and difficult to follow.

3.3. Common External Connection Manager Template

The final option is to have a single connection manager template for all external interfaces, i.e. those requiring transaction-based connections. The two internal interfaces would then have their own customised connection managers The customer had already used Rational Rose RealTime during the prototype development of one of the external components and made available the connection package (known as the TCPIP Key Mechanism) for possible re-use by Siemens Communications.

As can be seen in Figure 3.1, the top-level TCPIPHandler object is responsible for initialising the Winsock library ready for use as well as managing the incoming and outgoing connections. Outgoing messages are passed to the TCPClient, which creates the client connection, sends the message, then closes the connection down.

In order to handle incoming messages to a particular port, the TCPIPHandler initialises the TCPServerController, informing it of the server port number. The TCPServerController then creates the listening socket and kicks off the first server thread to listen for a connection. The listening socket is a global variable so that it can be accessed by each server thread. As soon as a server thread receives a connection, the controller is informed and the next thread is set listening whilst the first receives the message.

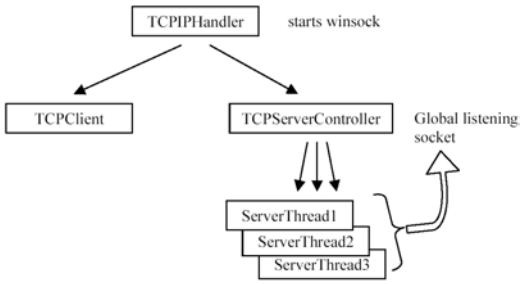


Figure 3-1 Overview of TCP/IP Key Mechanism

Whilst this connection package contains the basic classes for transmission and receipt of messages using the transaction-based sockets method, it was designed for use in an application that only requires a single TCP/IP interface whereas the CIU requires multiple such interfaces. It would therefore require some small modification in order to re-use the TCP/IP Key Mechanism in the CIU.

It was proposed that the two internal connection managers should be designed from scratch in RRRT using non-blocking sockets. The reason for use of non-blocking sockets is that for permanent bi-directional connections, the socket code needs to be able to deal with incoming and outgoing stimuli asynchronously. Using a blocking socket to wait for input would delay outgoing message transmission.

Some of the low-level socket handling code from the prototype CIU connection classes could be incorporated in the transitions and operations of the new capsules where appropriate in order to reduce the implementation time and reduce the risk.

3.4 Decision Analysis

In order to determine which of the three options is most appropriate, a Kepner Tregoe Decision Analysis was performed. This is a technique for arriving at a decision based on an analysis of the alternatives against the key objectives of the decision. The full analysis is shown in Appendix A.

There are two mandatory objectives – the solution must be capable of supporting both permanent and transaction-based connections, and it must be compatible with the RRRT and C++ development environment.

The CallBridge Framework option fails both of these criteria, and was therefore rejected outright.

The remaining objectives were weighted according to importance and the remaining two alternatives (having met the mandatory objectives) were judged on their performance against these criteria.

The Common External Connection Manager Template alternative performed better on most of the criteria and finished top overall. Therefore this option was chosen.

Despite the design goal of minimising the number of different connection managers, the decision analysis process showed that this objective was of relatively low importance compared to the issues of timescale, maintainability and risk.

4. SOFTWARE IMPLEMENTATION

In this section, the design and implementation of the software for the management of the internal connections will be described.

Rational Rose RealTime was used throughout the development for the design and implementation of the application, as well as the building and unit testing. The design is described in the following subsections, using terminology and diagrams from RRRT. The reader is referred to Rational (2002) for a detailed explanation of these concepts.

4.1 External Connection Manager Template

The design of the connection managers for the external interfaces is based upon the TCPIP Key Mechanism with only a few modifications to enable its use in the CIU.

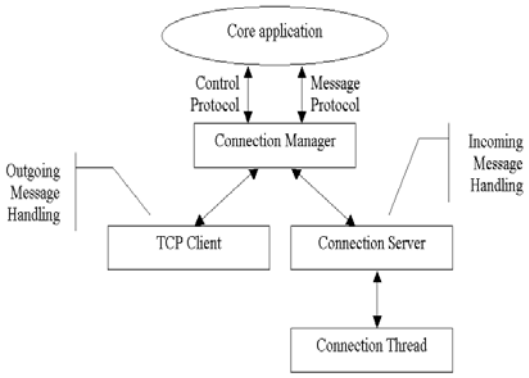


Figure 4.1External Connection Manager Architecture

Since the modifications were minor the design will not be described in detail here, however the basic architecture is shown in Figure 4.1 in order to provide a reference for the discussion on the CSTA interface design.

4.2 CSTA Connection Management

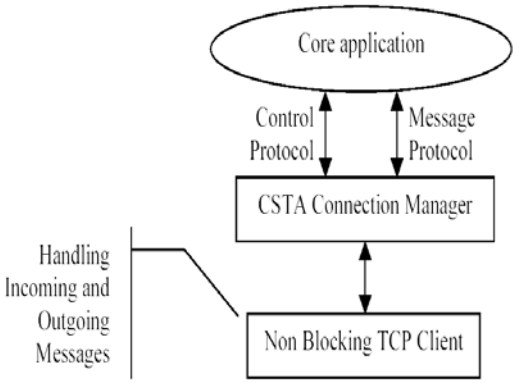
4.2.1Architecture

When looking at the design for the management of the CSTA connection, it is pertinent to investigate the possibility of re-using any parts of the TCPIP

Key Mechanism. To recap, the CSTA interface uses a permanent bi-directional connection acting as a client.

The overall structure of the External Connection Manager Template, whereby the core application communicates with the manager using one protocol for connection control and another protocol for messaging, and the manager controls the capsules that deal with the low-level connection processing, could be re-used. Indeed, using the same protocols would help to create an integrated connection layer with a common interface to the core application.

The ***TCPClient*** capsule from the Key Mechanism cannot be used here since it only handles outgoing messages. Similarly, the ***ConnectionThread*** capsule only handles incoming messages. Moreover, it uses a blocking socket to wait for input. This means that all other operations on that thread are blocked until a message is received. Whilst this is viable in the case of the external interfaces (since that thread is only performing a single operation, either transmitting one message or receiving one message) it is not appropriate for permanent connections where the relative ordering of incoming and outgoing messages on the single thread is indeterminate. In the latter case, it is possible that an outgoing message may need to be sent before the blocking call has returned. A non-blocking socket must therefore be used, so that stimuli from either the socket or the core application can be acted upon immediately.



4.2 CSTA Connection Manager Architecture

NonBlockingTCPClient capsule is created that handles both incoming and outgoing messages asynchronously. In order to make the ***NonBlockingTCPClient*** capsule re-useable, a new ***ConnProfile*** data class is created to hold interface specific information such as knowledge about the format of the message header and position of the message length field. This class is initialised by the ***CSTAConnectionManager*** and passed to the ***NonBlockingTCPClient*** on connection startup. Thus the operation of the connection is tailored to the CSTA interface.

The CSTA Connection Manager has responsibility for the startup and shutdown of the CSTA connection, as well as the configuration of the connection profile. The complexity lies in the NonBlockingTCPClient.

4.2.2 NonBlockingTCPClient Design

The ***NonBlockingTCPClient*** capsule has the following responsibilities:-

- Maintenance of the link to CallBridge DX, if the connection is in a started state.
- Reading and writing of messages at the socket level.
- The ability to handle asynchronous bi-directional communication with the ***CSTAConnectionManager***.

The implementation of the first two of these was already understood since this type of interface was implemented in the prototype. The implementation of the last one, however, is specific to the way in which Rational Rose RealTime handles its internal protocols and requires some investigation.

In RRRT, when one capsule transmits a message to another capsule, the message is placed onto the receiving capsule's queue and the capsule is signalled to wake up and process the message. This processing is all done within the RRRT capsule framework and normally the developer does not need to be aware of the mechanism by which this is achieved.

In the ***NonBlockingTCPClient***, however, the capsule needs to be woken up either by a message from the ***CSTAConnectionManager*** (i.e. the normal wakeup mechanism) or by activity on the socket. In order to do this it is necessary to customise the capsule's behaviour to allow it to check for both types of event. RRRT permits customisation of a capsule running on its own thread by enabling the developer to change the type of thread controller from ***RTPeerController*** (the default setting) to ***RTCustomController***. The developer can then override the default *waitForEvents*() operation with a customised version.

To enable the capsule to detect events on both the local interface (to the ***CSTAConnectionManager***) and the remote one (to CallBridge DX) use is made of the TCP/IP "select" function through which the

software can specify a number of sockets to be monitored for activity. RRRT implements this functionality through its ***RTIOMonitor*** class. For this to work it is necessary for a socket to be used for the local interface as well as the remote one, however a UDP socket will suffice in this case since reliability of the link will not be an issue.

On connection startup the following actions are performed by the ***NonBlockingTCPClient*** :-

1. create a UDP socket (attribute *internalFd*) and connect to the local port
2. register the UDP socket with the ***RTIOMonitor*** (attribute *ioMonitor*)
3. register the remote socket (attribute *c_socket*) with the *ioMonitor*, so that its status can be monitored
4. attempt to connect to CallBridge DX
5. if connection succeeds, the client is now ready to process messages
6. if connection fails, a retry timer is started, upon expiry of which connection will be attempted again

The *waitForEvents*() operation monitors activity on *internalFd* and *c_socket* as follows:-

1. it checks the status of the sockets by calling *ioMonitor.wait*() with a parameter of 0 to indicate 'no blocking'.
2. if there is something to read on *internalFd*, i.e. *ioMonitor.status*(*internalFd*) returns a non-zero value, then it wakes up the state machine by calling *recv*() on the *internalFd* socket. The internal message will then be processed through the state machine.
3. if the remote connection is in the established state and there is some data to read, i.e. *c_socket.hasData*() returns a non-zero value, the *readSocket*() operation is called to read the data in from the socket and process it.

4.3 LMUI Connection Manager

4.3.1 Architecture

Since the LMUI connection is, like the CSTA connection, a permanent bi-directional link, it also cannot re-use any of the classes from the TCPIP Key Mechanism. It should use the same control and messaging protocols to communicate with the core application, and will also have the same basic structure consisting of a connection manager (***LMUIConnectionManager***) to handle the startup and shutdown and to initialise the connection profile, and a low-level capsule to handle the socket processing.

However, as the LMUI connection acts as a server rather than a client, it cannot re-use the ***NonBlockingTCPClient*** capsule. Instead, a ***NonBlockingTCPServer*** capsule will be created.

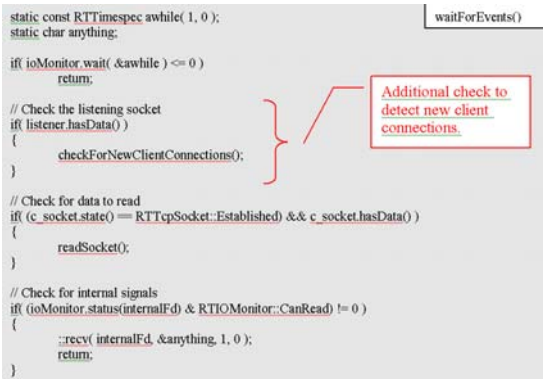


Figure 4.3 waitForEvents() operation in NonBlockingTCPServer

4.3.2 NonBlockingTCPServer Design

The design for the ***NonBlockingTCPServer*** is similar to that for the ***NonBlockingTCPClient*** in that it needs to create a UDP connection to the connection manager and use an RTIOMonitor to check for activity on both this and the client connection. The key differences are that on startup, a listening socket is created and the server then waits for the client to connect; the listening socket has to be registered with the *ioMonitor*, so that any new connection activity can be detected by *waitForEvents()* (see Figure 4-8); if a new client connection is received when the *c_socket* is already active, then the new connection is refused by closing it down; and if an active connection goes away, the server just waits for a subsequent reconnection – no retry state is required.

5. RESULTS AND DISCUSSION

5.1 Test Objectives

Since the connection management software is very low level it is crucial to the operation of all communication between the application and the other system components. As such it is important to verify the connection software thoroughly before testing the higher level functionality.

The testing must prove that the appropriate connections can be established, that data can be transmitted in either direction, and that the software can recover from a connection failure.

5.2 Test Strategy

The initial unit testing of each connection manager in isolation was to be performed using Rational Quality Architect (RQA-RT), a desktop test tool that is integrated into RRRT. RQA-RT provides the

developer with the capability to specify, and automatically verify, a test sequence. The aim is to test as near to 100% of the code as possible. In order to demonstrate the RQA-RT test procedure that was used the testing of only one of the connection managers (the CSTA Connection Manager) will be described in detail, in section 0. The same procedure was followed for all connection components.

Once unit testing of all components has been completed, the application would undergo pre-integration testing on a test system in the laboratory followed by a period of thorough functional and performance testing.

5.3 Unit Testing the CSTA Connection Manager

In order to test any of the connection managers it is essential first to have an application to simulate the remote end of the connection. In the case of the CSTA Connection Manager, a server that can send and receive CSTA type messages is required. Note that the full CSTA encoding and decoding is not required for the testing of the low-level connection software since the connection management code is only concerned with the transport of the messages, not their content. Thus, the phrase "CSTA type message" means that the header part of the message has the correct length and format for CSTA messages.

A simple TestServer application was created in RRRT with the help of the ***NonBlockingTCPServer*** capsule. The parent ***TestServer*** capsule creates the

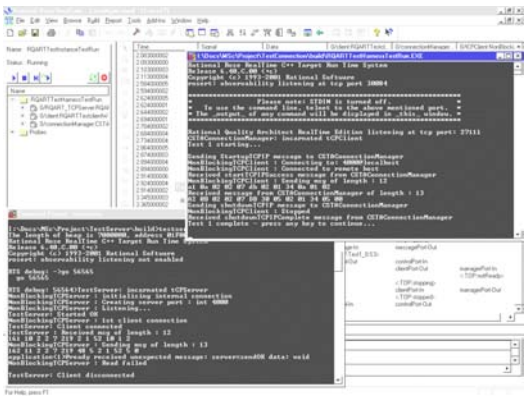


Fig. 5.1 Screenshot of RQA-RT Test Run

NonBlockingTCPServer, and then just prints out connection information and received messages, sending a response message in reply. The TestServer application was built in RRRT and the executable was run from the DOS command line when required.

The next stage of test preparation is to create a test capsule in RRRT, this was named ***ConnectionTest***. This will contain the capsule under test (CUT), in this case ***CSTAConnectionManager***, as well as a dummy client capsule, ***ConnClient***, that is connected to the test capsule to simulate the core application.

Each test is now specified by creating a sequence diagram showing the expected sequence of events for the particular scenario. Input data must be specified for signals sent into the CUT and, similarly, the content of output signals must be checked. Once the test specification is complete and a build component has been created for the test capsule, RQA-RT is then invoked to verify the sequence. RQA-RT automatically generates the test code for the dummy ***ConnClient*** capsule, then builds the test capsule and runs the sequence. Finally, it verifies the test output against the expected sequence and reports any differences.

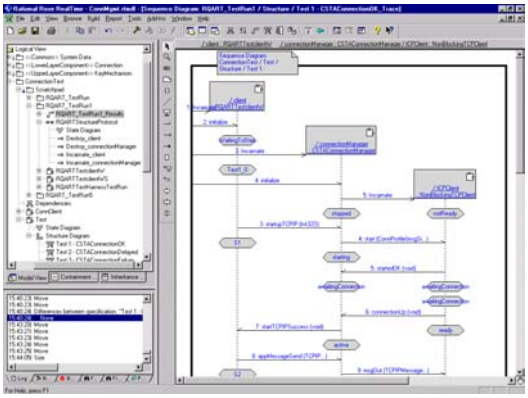


Figure 5.2 Screenshot of RQA-RT Test Trace

Figure 5.1 contains a screen shot of a running RQA-RT test. It shows the ***TestServer*** output window as well as the console window for the test. In the background is the RRRT window that is controlling the test run.

Figure 5.2 shows a screen shot of the test trace that is generated at the end of the test run. The Log window in the bottom left of the screen details any differences between the test specification and the trace. In this case there were no differences.

6. FUTURE DEVELOPMENT

The report shows that it is possible to design a consistent approach to managing to diverse types of connection within a single application and that this can be done within the Rational Rose RealTime framework by use of the RTCustomController designation.

Furthermore, the use of RQA-RT as a testing tool enabled rapid development of a test environment, which would have been very difficult to achieve using traditional methods.

The combination of a robust framework with the already proven interface specific code from the prototype development meant that very few problems were found relating to the connection management software, and only one fault was raised in this area of software during formal system testing. This fault, in the external connection manager software meant that a failure to connect to the remote server in order to transmit a message caused the application to block for around 20 seconds.

Further work has been done more recently to enhance the performance and maintainability of the external connection managers. Use of the dynamic thread creation and deletion capabilities of RRRT has enabled the production of a single common connection manager which dynamically incarnates TCPClient capsules on separate threads. This has solved the bug that caused the application to block.

REFERENCES

RATIONAL SOFTWARE CORPORATION 2002. **Rational Rose RealTime Modeling Language Guide**

KEPNER, C.H. & TREGOE, B.B. 1997. **The New Rational Manager.**
Princeton, New Jersey: Princeton Research Press

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION 1992. **Standard ECMA-179, Services for Computer-Supported Telecommunications Applications (CSTA)**
Available at: http://www.ecma.ch/

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION 1992. **Standard ECMA-180, Protocol for Computer-Supported Telecommunications Applications (CSTA)**
Available at: http://www.ecma.ch/

BIBLIOGRAPHY

COMER, DOUGLAS & STEVENS, DAVID 1991. **Internetworking with TCP/IP**
Englewood Cliffs, New Jersey: Prentice-Hall International, Inc.