Requirements Document #1

August 12, 2003

Dave Gilbert

1 Introduction

This document presents an overview of a general software tool for use at the McMaster Nuclear Reactor called MNRsdq (sdq=SimulationandDataaQuisition - we are open to suggestions here). The purpose of the tool is to provide remote access to a variety of software tools. At the same time, this collection of related tools will be organized under a relatively uniform interface. The scope of the project is illustrated in Figure 1.

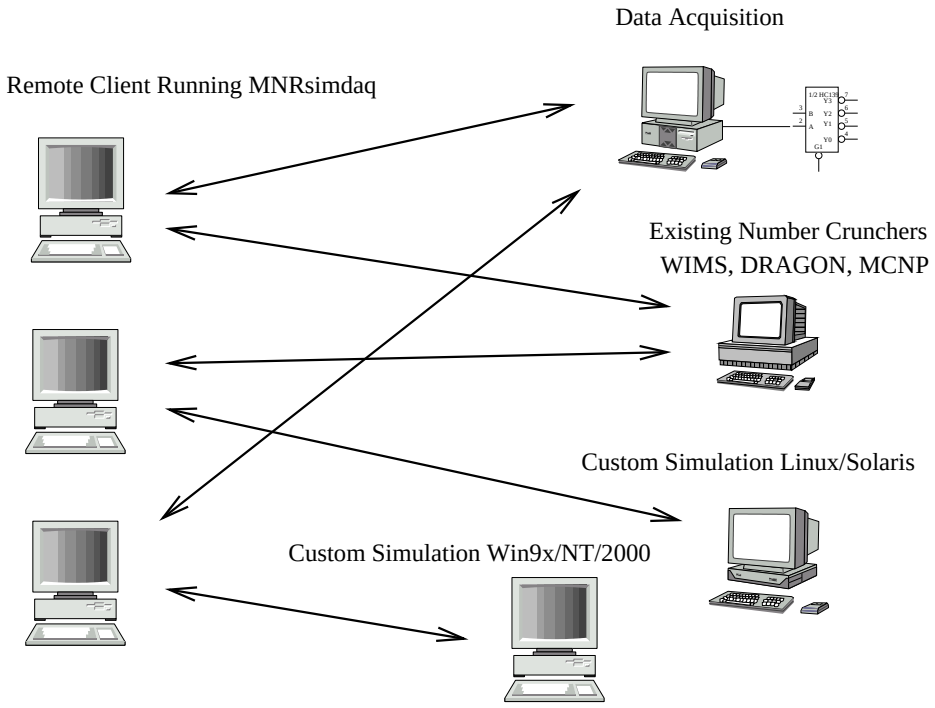

Figure 1: MNRsdq Overview

This implementation has several goals, the first of which is promoting access to the tools used by and developed for the MNR by a wider group of people. Communications take place over the university Internet, and are implemented by a simplified set of TCP calling libraries. A single display client program will be maintained that provides accesses to multiple data and simulation servers.

This document is intended to be a short introduction to the design of the client and server interface. This document does not describe the separate modules embedded in the client software, but rather defines how new client modules ought to be incorporated. No academic references are

supplied, rather the emphasis is on essential implementation details. This document should be a focal point for persons interested in the project, input and modifications are welcome.

1.1 Design Ideals

The principle concept in use here is a separation of function and display. This provides several distinct advantages:

- Stability: If a client program crashes, this should not impair the function of any server.
- Redundancy: Multiple client programs may be used to monitor the same server, so in the event of the failure of a client, another online client can access the same server.
- Remote access: Data acquisition equipment can be monitored from anywhere on campus
- Collaboration: This display tool should become a virtual meeting place for the scientists and engineers of the MNR.

2 Installation

The Windows display component of MNRsdq is implemented within Measurement Studio LabWindows/CVI. This document will assume that the programmer is working under Windows9x,NT, or 2000. LabWindows is a ANSI C programming environment, and so some familiarity with C programming will also be assumed. After running the installation CD for measurements studio you should find that all related software is installed at:

```
C:\Program Files\National Instruments\MeasurementStudio
```

Several instructive tutorials are included with Measurement Studio. They can be found in the following subdirectory:

```
C:\Program Files\National Instruments\MeasurementStudio\cvi\tutorial
```

For consistency it is recommended that the MNRsdq display development tree be built under:

```
C:\Program Files\MNRsdq
```

If you download the .zip file from the website, and unzip the file you will see several subdirectories.

```
C:\Program Files\MNRsdq\V0.1
C:\Program Files\MNRsdq\V0.11
```

Version numbers should help to identify various strains of the interface. The version number consists of a major part (i.e. the integer of the fraction), and a minor, or decimal part.

Increases in major version numbers should indicate major increases in functionality for the user, at the possible risk of newly introduced incompatibilities between servers and clients. Ideally servers should be backward compatible with older interfaces, although generally this is a fairly strict requirement. Clients and servers can at least communicate their version numbers, and issue warnings or errors in case of poor matches.

Increases in minor version numbers should indicate bug fixes, and small enhancements in the interface. Modules which are actively being modified should progressively increment the minor version number until the code is stable enough.

3 Interface Design

The top level interface design acts as an index into the collection of available modules. A typical menu design is shown in Figure 2, along with the menu path that users follow.

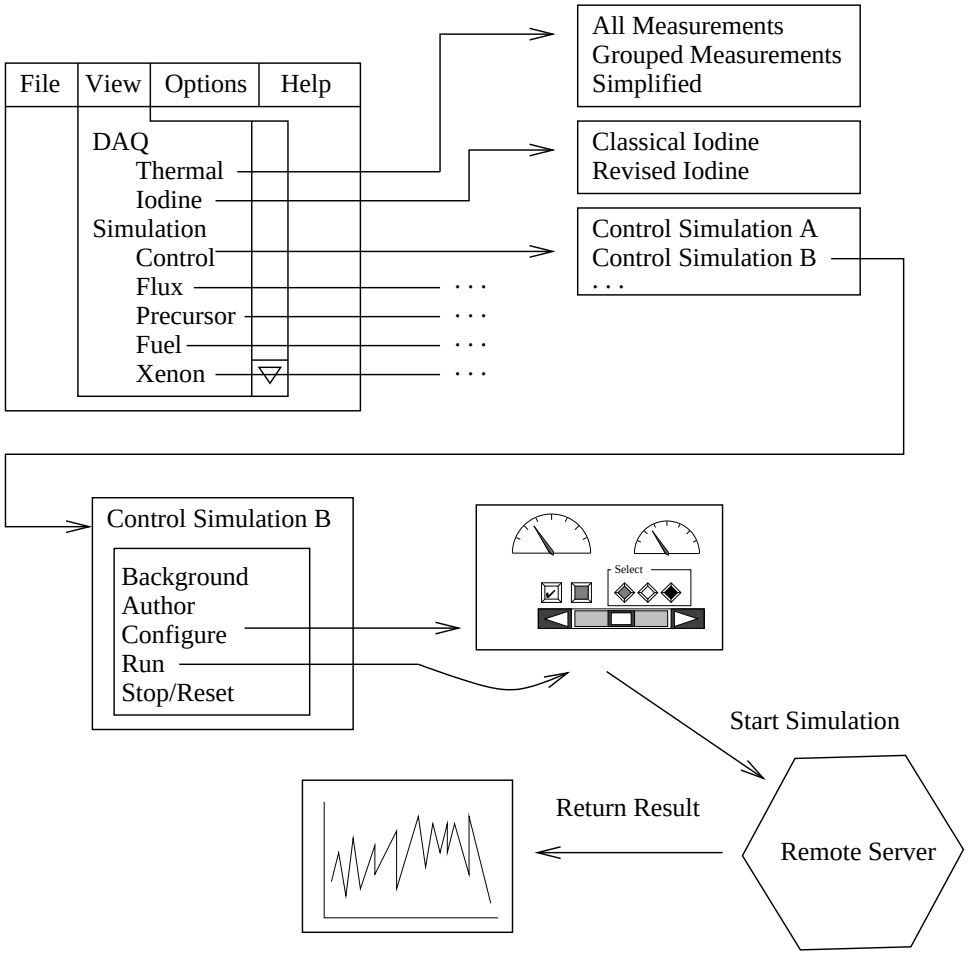

Figure 2: Interface design

In this example the top level of the view menu hierarchy shows a basic distinction, simulation or data acquisition. The second level menus, further sub-categorize the problems. A good rule of thumb for defining trees is that they should neither be too deep, nor too wide. Exactly what menu items are available will evolve over time.

Each simulation, or data acquisition monitor should have the same basic buttons for consistency. Here we suggest:

1. Background: 100-500 words describing the purpose of the module, and the theoretical basis behind its operations. An experimental, or non functional module should have its background button flashing red.

2. Author: Author contact information may be important for a variety of reasons. Suppose the simulation module predicts a reactor failure, in such a case it would be appropriate to contact the author and try to determine how accurate the prediction is. Authorship is also appropriate to acknowledge up-front to engender collaboration.

3. Configuration: Each server should be able to report its default or current configuration. For some servers it may also be possible for the configuration to be altered, especially in the case of supplying new configuration values.

4. Run: Forward the configuration values to the appropriate server, and start a new simulation, or data acquisition series. Probably there will be a few different categories of tool execution behavior, so the precise meaning of run will be situation dependent

   (a) Tools which run non-stop- for example data acquisition tools

   (b) Tools which have long running times- for example a detailed simulation

   (c) Tools which have short running times- short calculations which can be done in under a minute

5. Stop: Certain simulations may not always be guaranteed to converge, so in some cases it is important to be able to stop the simulation.

At some point the server will communicate its results to the client, after the run key is pressed the client must be ready to accept results and display them.

4 Communication Implementation

TCP sockets are included as a part of LabWindows and they provide a platform independent way for machines to communicate. Socket programming can be rather difficult especially if both communication errors and memory leaks are to be avoided.

You should find in the current implementation of MNRsdq the files named:

```
TCP_wrap.c
TCP_wrap.h
```

These short libraries simplify the socket message calls. They supply the following functions:

```
int TCP_open(char * remote_IP);
int TCP_read(char * string);
int TCP_write(char * string);
int TCP_fsend(char * filename);
int TCP_close();
```

This collection of functions is meant to be very simple and to handle several of the more complicated issues related to socket programming. The functions are meant to behave in a way similar to file handling functions. They are coded for robustness, and ease of use rather than efficiency. Two modes of communication are supplied, short strings (under 100 bytes) and arbitrary files.

Strings are easier to deal with than binary structures, strings are also platform independent and they simplify debugging. Strings are limited to 100 characters in length, to avoid bounded buffer overflow problems. To facilitate the transmission of more complex data, files may also be sent from the working directory of one peer to another. It is expected that some simulations may generate graphic files (.gif, .jpg, .bmp etc.) as part of their output, and it is therefore helpful to have a programmers level ftp style mechanism.

TCP_open() Establish a connection to the remote machine. The version number of the interface is sent to the server, and if there is a mismatch an error is reported. Servers wait for connections, they pass in NULL as their remote_IP address since they don't know where the connection may come from. Returns 0 on success, -1 on failure.

TCP_read() Assuming that a network connection has been opened, wait for a string to be received. By default TCP_read() will time out after 5 seconds has gone by. Returns 0 on a timeout, -1 on a failure, 1 indicates a string was received and stored in the users buffer, 2 indicates that a file was received, the name of the file is stored in the users buffer.

TCP_write() Send a string. Data must be a string of less than 100 characters. A server which receives a connection will automatically send replies via TCP_write() until either the server or the client closes the connection. Writing to a closed connection causes an error. Returns 0 for success, -1 for failure.

TCP_fsend() Send a file to the peer. The file should exist in the current working directory, it is presumed that the peer has sufficient disk space to store the file. The file name is passed as the argument. Returns 0 on success.

TCP_close() Since only one connection at a time can be opened, TCP_close() must be called in between sessions. Returns 0 on success.

4.1 Client Server Example

There are a wide variety of communication paradigms that one might choose to follow. The example shown in Figure 3 illustrates how a client and server might communicate with one another using the suggested communication primitives.

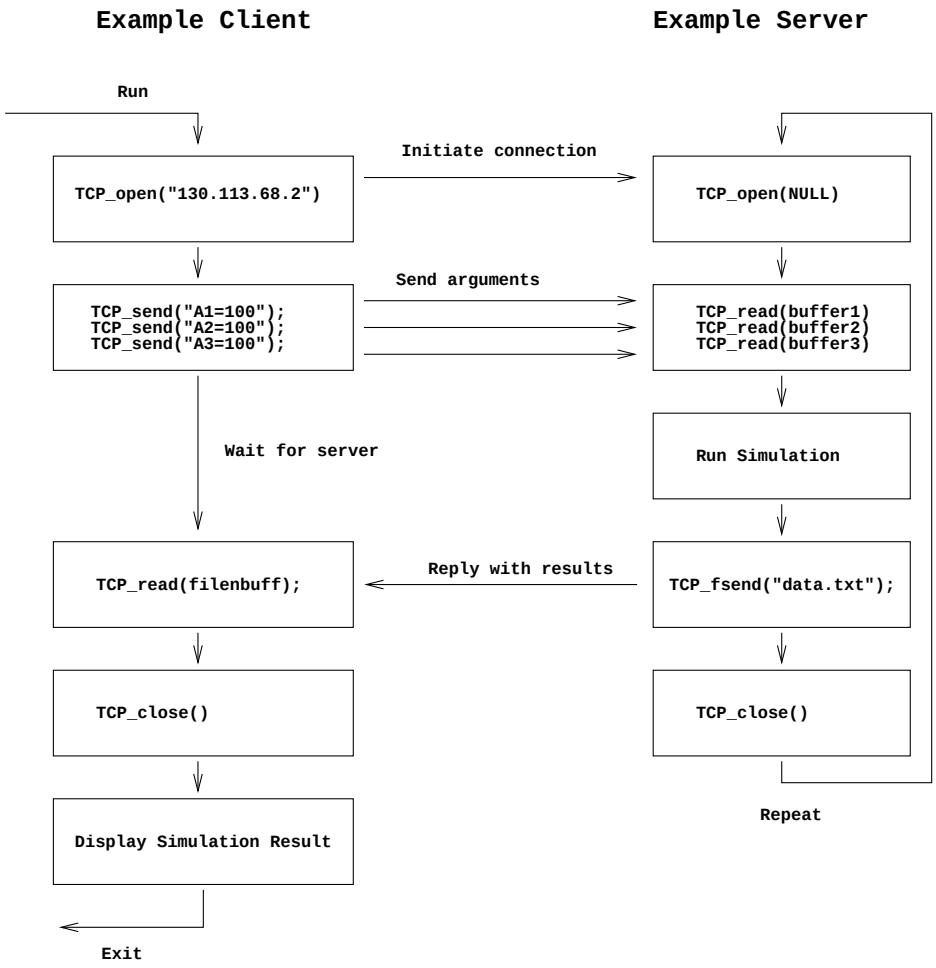

Figure 3: Example Client Server Design

The communication subroutine of the client is initiated when the run button is pressed in the GUI window. The client must know the IP address of the server that it is to communication with. The server has called TCP_open with a NULL argument, meaning that it is willing to accept a connection from any client. Once the client has initiated the connection the server will probably expect the client to send several parameters. The client will be forced to wait for the server to complete its calculation. After the server finishes transmitting its results it should close the connection and return to the start of its communication loop. The client should display the results transmitted from the server after it has closed its connection.

The previous example should be recognized as a hand coded implementation of something like either Java's remote method invocation (RMI) or the older Unix remote procedure (rexec) calls.

While it is believed that the example shown in Figure 3 is likely to be a typical communication pattern it is not the only possible pattern. Other common scenarios to consider:

Slow Server vs. Fast Server

Some servers, like a data acquisition machine, may always be able to instantly supply results when clients connect. Simulation servers which take a long time to run may not have this luxury. Programmers must make some sort of decision about what should be done in the case when a slow server is already handling a remote client. One possibility is that the server can simply reply with a busy message if more than one client seeks service. Slow servers could try to handle multiple concurrent clients, or Slow servers could queue up client requests and handle them in sequence.

Breaking Connections

Clients who talk to slow servers should be able to break the connection some how. This is important in the case of iterative solvers which may not be able to converge in a reasonable amount of time, or which cannot converge under any condition.

Progress Reporting

Slow servers should report progress to their clients in some way. A time estimate would be ideal, although in many cases this may not be practical.

4.2 Applicability of Client Server

In some cases the use of network communications may be overkill. In the case of data acquisition, the client server mechanism is necessary if the collected data is to be displayed anywhere other than on the machine which is directly connected to the data acquisition hardware. Distribution in this case is forced by the physical location of the equipment. Simulation software may need to be located on a specific type of machine, or may only run well on a very fast machine, and so in some cases separating the simulation execution server, and simulation display client may be helpful. However in the case where the simulation server is small enough that it can be easily embedded into the client, it is suggested that it should be.

Distributed programming is a solution to a problem only when the distributed problem exists in a kind of forced way. The data acquisition problem is the best example of this. When the server is simple enough that it can be implemented as simply a subroutine of the interface then many of the aforementioned communication problems can be avoided. It was pointed out that the client server model in use here tries to emulate a remote function call, so it should be clear that if there is no need for the call to be remote, then it should be handled locally.

5 Typically Programming Difficulties

To be written. It has been my experience working with students that typical things tend to trip them up, and it is often helpful to supply a FAQ style section to help them along with their programming. Some possible sections follow... to be filled in as needed.

5.1 C programming

5.1.1 reading & writing files

5.1.2 handling strings

5.1.3 dealing with pointers

5.2 Unix/Linux/Windows Differences

5.2.1 LabWindows under Linux and Windows - Compatibility issues

5.2.2 Unix Sockets versus LabWindows Sockets

5.2.3 Linux/Unix command line basics