New Folder Name Data Acquisition & Analysis

Work in progress on Data Acquisition and Analysis

Tim R. Barnett, Bernard F. Schutz, Justin R. Shuttleworth,
David Nicholson
Department of Physics and Astronomy
University of Wales College of Cardiff, Cardiff, U.K.

March 1992

1 Introduction

This document is a successor to an internal GEO report "Data Acquisition and Analysis" that many of you will have read already. It describes our current thinking about the data acquisition and analysis systems that will need to surround a full-size laser interferometer. We describe the analysis prototyping system `Grid` that we are developing and data formats for archival, exchange and processing with a case study of the storage of data taken from the Glasgow prototype detector.

Early discussions within the GEO group revealed that the data acquisition and analysis systems on such a complex scientific instrument as a large-scale laser interferometer were likely to be non-trivial to implement. The efficient and reliable operation of such systems is vital not only when the instrument is observing but also during the comissioning phase. The expected data rates and the perceived need for *flexibility* during comissioning led us to propose that the large interferometers be surrounded by high-speed computer networks, which would connect several computers that cooperate to capture, analyse and archive data. This document describes the software tools and techniques which we are developing for such a distributed computing system.

There are many advantages to using such an approach. Data acquisition peripherals may be hosted in computers which are attached to the network. Data may thus be acquired anywhere around the detector and sent to any other computer system attached to the network. Data acquisition peripherals may easily be added or reconfigured as needs change. Other computer systems on the network may be dedicated to archival of data, or computationally-intensive analysis. Portable workstations allow experimenters to view and analyse data taken from anywhere on the network wherever they may be. Analysis may be performed remotely on more powerful computer systems and the results viewed on some local workstation. Local control functions may be remotely instigated and monitored. Many experimenters may work on different aspects of the detector at the same time. The network may easily be expanded — adding more computing power if needed.

Such flexibility can only be guaranteed by adopting various software and hardware standards. This document describes the various standards that we have chosen to embrace, and our reasons for doing so. The computing systems and network hardware that support these standards are available "off the shelf" now and are not prohibitively expensive.

2 Our development environment

It is worthwhile to briefly detail the software tools that we are using, and to explain why we chose these particular tools. A major concern is that the software that we develop be portable to a wide variety of target computer systems. We cannot predict (or dictate!) which computer systems will be purchased by the different research teams around the world. Therefore we must ensure that our software relies upon various computing standards. Perhaps the most important decision that we have made is to base *all* our development on the Unix operating system [Miller, Boyle and Stewart 1984]. Unix provides us with a mature and stable development environment. Unix has become the most popular operating system within the academic world since its creation over twenty years ago. Many of Unix's features have become enshrined in standards such as POSIX[1] which defines a portable operating system standard. By relying upon these standard features we can guarantee that our software will work on *many* different architectures — from a Cray down to an IBM PC compatible. Unix provides built-in support for networking and multitasking, and our development systems are connected by an Ethernet-based local area network [Digital 1980].

To try to enforce portability we have acquired two of the most successful Unix variants — the AT&T System V derivative SCO[2] OpenDesktop and the BSD[3] derivative SunOS[4]. Efforts are currently being made by the industry to bring the clutch of available Unix variants closer together. However, these two variants differ quite widely and if our software will run without alteration on these platforms we have already achieved some degree of portability. We have also recently acquired an AT&T System V Release 4 version of Unix on we which can test and develop our software and intend to obtain the next BSD release as soon as it is available. These four versions of Unix are the two pure variants (AT&T and BSD) and their commercial derivatives which together represent the vast majority of current Unix versions.

Unix has one main defect as far as the programs described in this document are concerned, which is its lack of explicit real-time programming support. That is, Unix does not provide support for programming *hard* real-time applications where requested operations can be guaranteed to occur at some designated time. However, several manufacturers are now offering POSIX-compliant Unix variants with real-time support. We anticipate that these specialised versions of Unix can be used if such "mission-critical" features are required. For instance, on a computer which has to perform some local control as well as data acquisition.

A vital component of our software is its graphical interface. The most widely used graphical interface on Unix-based computers is the X-Window system from MIT [Quercia and O'Reilly 1988]. X provides a hardware-independent abstraction of a raster graphics device. X was designed to be as hardware (and even operating system) independent as possible. A major part of the work performed by the X-consortium at MIT and elsewhere has been to port X to many different varieties of Unix. Therefore X is particularly suitable for us to adopt as a standard because we can guarantee that our software will run on a large number of different architectures. X consists of two elements — the server and a collection of programming libraries. The server contains device-dependent code to drive the local graphics hardware. The server is a separate process which accepts graphics commands and transmits user-interaction events to client processes through a network-independent protocol (the X-protocol). Programs (clients) written to use X, call functions in the supplied libraries which communicate with the server process using this protocol. This

[1] See the IEEE Portable Operating System Standard 1003.1-1988
[2] Santa Cruz Organisation
[3] Berkeley Software Design
[4] Sun Microsystem's Operating System

communication can take place across a network. That is, the client process and the display server do not have to coexist on the same machine. This powerful abstraction sets X apart from most other graphics systems, and provides us with exactly the functionality that we wish to exploit. That is, powerful analysis can be performed on some remote compute-server and the results of the analysis can be displayed graphically on a local workstation. An X server can communicate with many different (possibly remote) client processes at the same time. The current release of X is considered a stable development platform and is distributed free of charge.

During the last decade "object-oriented" programming has emerged as the programming style of choice for large projects, and is regarded as especially suitable for the programming of graphical user interfaces. Academic and industrial software developers both agree that object-oriented programming techniques are as important a step forward as the advent of structured programming in the sixties. A detailed overview of object-oriented techniques can be found in [Booch 1991]. `Grid`'s approach suits the object-oriented paradigm very strongly indeed and therefore we have adopted an object-oriented language for all our software development. This language is a superset of the ubiquitous programming language C. The language is called "C++" and was developed by Bjarne Stroustrup at AT&T [Ellis and Stroustrup 1990]. C++ provides most of the features that typify object-oriented languages in an efficient as possible manner. Thus C++ represents a good engineering solution, providing needed features without the performance problems which tend to plague *purer* object-oriented languages.

The final software tool that we have settled on is the InterViews graphics library developed at Stanford [Linton et al. 1991]. Like X, InterViews is distributed free of charge. InterViews provides a C++ interface to the X-window system, providing an object-oriented view of the various X concepts. InterViews defines several light-weight graphical objects which may be assembled to form the user-interface of an X-based program. The InterViews project is part of the MIT consortium and is going to become part of the X standard. The current InterViews libraries are well-suited to our purposes and we are using them to develop the visual interface for `Grid` as described in section 6.1. InterViews is also under development, and we expect to make use of its increasingly sophisticated features as and when they become available.

We were much encouraged to learn that the NRAO-led project to update the well-known astronomical software package AIPS has also independently decided on the development environment of Unix, X-windows and C++. Having chosen, installed and familiarised ourselves with our development environment (itself a process that took many months) we began to program in earnest towards the end of 1991. So far, we feel that our decisions have been validated and already have a (minimal) prototype implementation of `Grid` up and going.

3 Networking and distributed systems

3.1 The Fibre Distributed Data Interface standard (FDDI)

FDDI[5] is a recent networking protocol designed by ANSI to operate over high-speed fibre-optic cables. Surrounding a large-scale interferometer by such a network is becoming increasingly attractive. FDDI interfaces have continued to drop in price and seem likely to become even cheaper. Several companies have also announced interfaces that use the same protocol over unshielded twisted-pair cabling offering the same performance for much reduced cabling costs[6]. The only potential drawback with twisted-pair is that electrical interference from such cabling may introduce

[5] See ANSI standards X3.139-1987, X3.148 Draft and X3.166 Draft for full details of the current specification of FDDI.

[6] This system is known as CDDI, for Copper Distributed Data Interface.

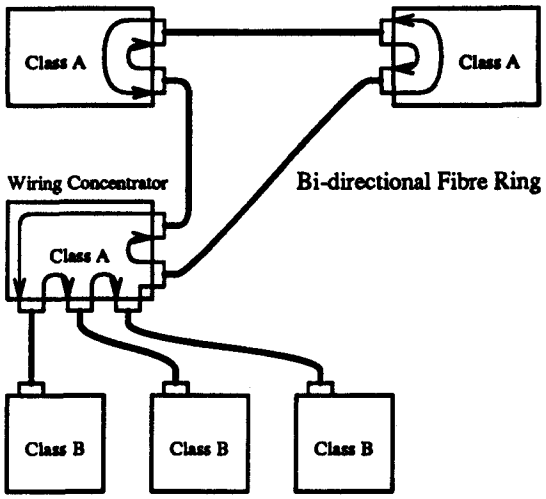

Figure 1: An example FDDI network

noise into nearby sensitive detector systems. A costing for the fibre required is given below and we would suggest that the noise-immunity offered by fibre is probably worth paying for. FDDI can support the TCP/IP protocol suite (described in section 3.2) available to us on our development machines and therefore we can guarantee that our software will operate in an identical (albeit much faster) fashion over such a network. FDDI system components are available *now* for a wide range of computer architectures. We intend to purchase FDDI hardware for testing. We will then be able to report on realistic data rates, and what network delay might be experienced by control loops under different network loads.

Figure 1 shows an example FDDI network. Note that the network consists of twin fibre rings (known as the primary and secondary rings) with data running in opposite directions on the rings. The fibre is interrupted by nodes which can be one of two classes. Class A nodes connect to both primary and secondary rings and can re-configure the ring structure should a fault develop. Class B nodes trade lower cost against the fault-tolerance of Class A nodes. However the secondary ring can also be used to carry data. This gives a fully configured FDDI system 200 Mbit s^{-1} of possible throughput. Once a protocol suite such as TCP/IP (see section 3.2) is used, the effective *data* transport rate with full fault-tolerance may be around 50 Mbit s^{-1}, which is equivalent to over 150 16-bit data streams each sampled at 20 kHz. Discussions with the research teams have suggested that this maximum data acquisition rate should be adequate for our purposes. Assuming a 3km arm-length, we would need 24km (2 loops) of fibre — which would currently cost £80,000. Away from the sensitive detector areas (for instance in the main computer room) we can use twisted-pair and/or Ethernet for any additional network domains.

3.2 The TCP/IP protocol suite

We have assumed implicitly that the local-area network will be based upon some common, standard communications protocol. The particular standard communications protocol we are currently

using is the TCP/IP[7] protocol suite which was developed in a DARPA project and is used by the Internet. The operation of TCP is detailed in [DARPA 1981a], and IP in [DARPA 1981b]. The Internet is the most successful (and largest) network in the world, currently linking hundreds of thousands of machines world-wide, and growing in size at an amazing rate. It is safe to assume that emergent computing systems will continue to support TCP/IP. Adoption of TCP/IP as a network standard will also ease connection of a detector's computer system to the Internet providing *global* access (if needed) to the features described in this document. Research teams and other collaborators would potentially be able to monitor the activities of the detector remotely, in real-time, from any computer attached to the Internet. This may be achieved in the simplest manner by remotely logging into one of the detector's computers perhaps to initiate an X-windows session, although more sophisticated connectivity can be supported by the Internet. Indeed a (less ambitious, but conceptually similar) system is already in operation at the Maryland and Rome cryogenic bar detectors. As detector sites are unlikely to be situated where all the research teams are located, remote access may prove very useful. As with any sophisticated multi-user computing system, access can be restricted to authorised users. User status may be controlled to range from just low-priority searching of the archive database access to god-like absolute control of the detector's computers.

Certainly, remote querying of the archive database (described in section 5) will be a powerful feature when the detector is operational. We anticipate that sophisticated analysis programs based on `Grid` will be able to perform on-line, automated, real-time searches for candidate gravitational wave events. The work described in [Watkins 1991] has shown the feasability of such an approach. Using relatively modest computing power (and a reduced number of coalescing binary filters) Watkin's system performs automatic analysis of data taken from the Glasgow prototype at around eight times real-time. That is, an hour of collected data takes approximately eight hours to process. With more powerful computers and software we are confident that real-time analysis is attainable. It is not unreasonable to imagine the following scenario once detectors begin full-time observing. A designated detector network correlation site may receive "interesting" events over the Internet from each currently operational detector within seconds. Such a site could then remotely extract the relevant information from each detector's database in order to perform an event correlation, possibly (assuming sufficient computing power) producing strong gravitational wave candidates and triangulated source positions within minutes of the actual event. Adoption of the various proposals in this document makes such an enticing development possible.

Moreover, connection to a global network is important even if only for the more day-to-day tasks of mail, news and file-transfer. Many sites (and hopefully all) involved in our endeavour are or will be connected to the Internet. The Internet offers a powerful resource for international collaboration that we are well placed to exploit.

4 File format

Exchange of data between computers on the FDDI network and between different detectors is essential. This document proposes a simple data standard designed to facilitate data exchange. This standard should allow easy interchange of binary data through networks and on various portable media. The standard proposed arose from encouraging informal discussions with members of the European, Japanese and American initiatives. The standard does not intend to dictate *what* data should be stored, merely *how* data should be stored. This document should be treated as a request for comments — hopefully after one or two iterations we will have a standard that

[7] Transmission Control Protocol/Internet Protocol

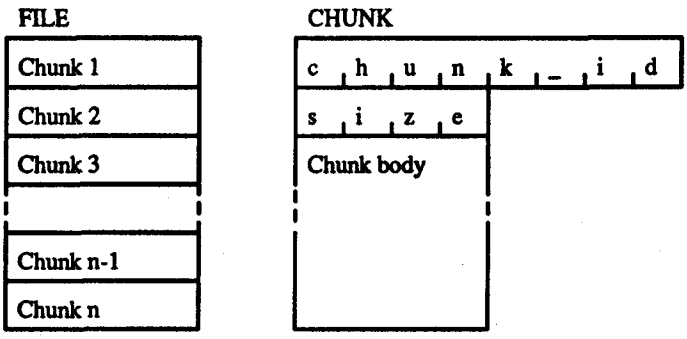

Figure 2: File and Chunk structure

everyone is happy with. The sooner that we can agree on a standard the better! Data are already being produced from prototypes (and from simulation programs) that we might wish to share. The same format is also used internally by `Grid`. If we arrange all data in this simple format then a single set of software tools can be used to read, manipulate, and create such data. In Cardiff data taken from the 100hr Munich/Glasgow prototype run is being massaged into this format to simplify our analysis work.

4.1 Basic Format

1. Files consist of a sequence of fundamental units of data called "chunks".
2. Chunks have a very simple structure as shown in Figure 2. A chunk consists of an eight byte ASCII identifier which signifies the chunk type, followed by a four byte unsigned integer chunk size, followed by the actual data — the chunk body. The size is defined to be equal to the chunk body length in bytes plus 12 which is the size in bytes of the header.
3. Chunks may contain chunks.
4. Chunks are padded to the nearest 4-byte boundary.
5. All integers are stored most significant byte first in 2's complement form[8], and all floating point numbers use the ANSI/IEEE 754-1985 standard.

For the moment (until someone suggest a better name) let us refer to this file format as **BIFF** for Binary Interchange File Format. The ASCII identifier denotes the "type" of the data stored in the chunk, rather than being a "name" associated with the particular data stored in an individual chunk. The chunk identifier tells the reading software what routines to use when reading the data. Thus an identifier **FFT**___ might denote a fourier transform, but it might be a subchunk of a larger chunk also containing a text chunk that describes the FFT — where it came from, etc. — in a unique way.

This format has been arrived at for the following reasons. Firstly, the format must be capable of representing complex data structures without significant overhead. Chunks as described are

[8] A standard way of denoting sign — various other methods have been used in the past but *nobody* does it differently nowadays.

sufficiently general to store any data and demand a minimum of overhead in terms of size and complexity. Chunks associate a data type with some raw data. The size is stored in the header to enable skipping when reading a file. That is, inappropriate chunks may be speedily ignored by moving the current file pointer to the end of the chunk. Nesting (to any depth) of chunks within chunks permits representation of collections of data with constituent aggregate values. A top-level chunk may therefore contain several (maybe optional) subchunks.

Secondly, most workstations and minicomputers currently have 32-bit CPUs and 64-bit CPUs look set to be the next step. The data sizes of the chunk identifier and chunk size (at 1 or 2 CPU words) were chosen so that they could be efficiently manipulated by such CPUs. Chunks are padded to a 4-byte boundary for the same reason. Such padding will, in general, only be necessary for a few "odd-sized" chunks. I/O buses tend also to be 2^n $(n > 4)$ bits wide and thus I/O bandwidth to memory and peripherals is optimised by this arrangement.

Thirdly, the format defines a byte-ordering for integers. The most significant byte first or "big-endian" standard was chosen to be compatible with the byte ordering inside a standard TCP/IP network packet. It makes sense that any data that we wish to transfer should not require massaging before being sent through our main communication network. Many CPUs are big-endian (e.g. the Sun workstation CPUs), with the notable exception of Intel's range. Intel's later processors have a speedy "byte-twiddle" instruction precisely so they can efficiently communicate with the big-endian world. Very recent DSP[9] chip design research seems to be indicating that most-significant-byte first representation of numbers may offer profound speedup in a new generation of DSP designs. In line with the issues discussed in section 3.2, `BIFF` thus facilitates efficient use of de-facto standard software tools for TCP/IP-based networks, such as Sun's Network File System and Remote Procedure Call. It is of note that if a computer can communicate with such a TCP/IP network (and most can) then it has, if necessary, built-in software to transform any local integer representation into the standard big-endian form.

Finally, the chosen floating point representation is the standard representation for almost all recent computers and workstations and defines a binary format for 4- and 8-byte floating point numbers. This standard provides representation of numbers in the range $\pm 10^{\pm 38}$ for 4-byte floats and $\pm 10^{\pm 308}$ for 8-byte floats. Most mathematical coprocessors and accelerators currently on the market manipulate numbers in this format.

As an early part of our software development we have written code which performs reading and writing of `BIFF` files, and plan to distribute this utility library to all of our collaborators.

4.2 BIFF in practice

The `BIFF` definition puts the onus on the creators of files to provide an unambiguous definition of the layout of chunks that they have designed. Someone wishing to read such files must have a complete description of each chunk, detailing whether the chunk is optional, what data is stored in the chunk and so on. Some informal criteria for choosing chunk identifiers should be agreed by the various research teams in order to avoid identifier clashes. This could be achieved by allotting each research group a standard prefix to use in headers. Identifier duplication could cause considerable problems when trying to unravel someone else's data that describe their own specific types of chunks. Obviously the length of the prefix should be kept to a minimum so that relatively meaningful chunk identifiers can be invented. A prefix of two or three characters should prove ample. The examples in the Section 3 employ the prefix `GW_` to indicate that the chunks contain data taken from the Glasgow prototype.

[9] Digital Signal Processor.

Some chunk types may emerge as a standard (e.g. FFTs), and some chunks may just wrap a previously existing standard. For instance, we suggest that bitmaps (an array of bit values commonly representing an icon) are stored in the standard X-window manner inside a **BITMAP_** chunk. Formats for images are well-developed and we would wish not to duplicate the effort in that field.

4.3 A BIFF example

Figure 3 shows an extract of the **BIFF** format which we have developed for the data taken from the Glasgow prototype detector. The format is described using a pseudo Backus-Naur grammar. The top-level chunk which we have called **GlasgowFileWrapper** is defined as having a chunk identifier **GW_FILE** followed by a size in bytes of the chunk, followed by four sub-chunks. This is consistent with the general **BIFF** definition of a chunk — an 8-byte chunk identifier followed by a size followed by the chunk contents. So each file of data in this format is a single chunk of type **GW_FILE**. Notice that for chunk items that are not sub-chunks a comment on the right hand side makes the type of the item explicit.

Definitions for the format of the four sub-chunks is also given. For instance the **DESCRIPT** chunk which just contains ASCII text of arbitrary length. Such a chunk is used to attach comments or textual descriptions to an enclosing chunk. For reasons of space the complete format definition is not given here — some of the sub-chunk formats are left undefined. However, this format gives an idea of how real data may be arranged in a **BIFF** file. The definition also serves to demonstrate how an unambiguous description of a **BIFF** file may be given. However, the meaning of the chunk contents is not described and this would also need to be well documented if data were to be exchanged in this format.

The chunk **PowerSpectrumSampleSet** provides a good example of the kind of chunk that might be passed between units inside **Grid**. The definition given here describes precisely the format in memory, or on disk that such chunks would have.

5 Data Archival

We anticipate storing various data streams taken from the detector in a database. Such a database would have to be capable of storing the required quantity of data in real-time. This needs investigation, as many commercial databases have comparatively slow transaction rates. However, storing data from the detector in a database would allow sophisticated investigation of previous detector operation, and comparison with current behaviour. Section 3.2 briefly discussed the fact that remote querying of such a database would be possible.

During the commissioning of the detector we expect to discover what data needs to be archived, and what can be discarded. Obviously, some housekeeping signals should be archived so that correct functioning of the detector can be verified when analysing previously taken data. However, much of the available data may prove irrelevant when the detector is fully operational. It is certain that the quantity of data to be archived is likely to be large (maybe 1Tbyte/year) even for today's storage media. Some hierarchy of memory devices may prove useful, regularly accessed data being kept on fast disk, and less regularly accessed data migrating automatically to slower but cheaper devices. Systems offering 0.5Tbyte store based on such a principle with disks and high-density magnetic tape jukeboxes are being advertised now, so it seems safe to assume that a system with a suitable capacity may be available by the time the detectors become operational.

Data in the archive would again be stored in the format discussed in section 4. Recent database advances allow such techniques as versioning and temporal querying to be performed, and therefore

```
GlasgowFileWrapper  :=
GW__FILE                8 character string, not nul terminated
<size>                  4-byte integer
[Description]
[GlasgowDateChunk]
[GlasgowSpanWrapper]
[PowerSpectrumSampleSet]


GlasgowDateChunk    :=
GW__DATE                8 character string, not nul terminated
<size>                  4-byte integer
<year>                  4-byte integer
<month>                 4-byte integer
<day>                   4-byte integer
<hour>                  4-byte integer
<minute>                4-byte integer
<second>                8-byte real
[Description]

GlasgowSpanWrapper  :=
GW__SPAN                8 character string, not nul terminated
<size>                  4-byte integer
[GlasgowBlockWrapper].. (repeated as often as necessary)

GlasgowBlockWrapper :=
GW_BLOCK                8 character string, not nul terminated
<size>                  4-byte integer
[SecondaryFeedbackSampleSet]
[DigitalByteSampleSet]
[MicrophoneSampleSet]
[SeismicSampleSet]
[PrimaryErrorPointSampleSet]
[PrimaryVisibilityStatisticSet]
[SecondaryVisibilityStatisticSampleSet]
[WirePushStatisticSet]
[SecondaryErrorPointSampleSet]
[FourierTransformSampleSet]
[AmplitudeStatisticSampleSet]

PowerSpectrumSampleSet :=
PWR_SPEC                8 character string, not nul terminated
<size>                  4-byte integer
<transform count>       2-byte integer
<frequency>             4-byte real
<elements>              4-byte integer
<array>                 4-byte real array [elements]
[Description]           (optional)

Description :=
DESCRIPT                8 character string, not nul terminated
<size>                  4-byte integer
<text>                  N-character string, not nul terminated
```

Figure 3: File format for Glasgow Data

we intend to investigate the potential impact of such techniques on our analysis systems. Object-oriented databases are breaking new ground with a move from the standard relational techniques in line with the advent of object-oriented programming. An object-oriented database would allow us to store and retrieve the data-objects from inside the system described in section 6 directly and transparently. We hope to provide facilities within our software to make use of such a database and have therefore recently installed a copy of "Postgres", which is a freely available object-oriented database under development at Berkeley. The capabilities of this new generation of databases are discussed, in one of the first overviews of the subject, in [Kim 1990].

6 Grid

The time-series analysis prototyping system that we are developing, currently known as `Grid`, attempts to provide a visual metaphor of an experimenter's laboratory environment. In essence, a lab can be thought of as a collection of *independent* information processing units (hardware FFTs, control-devices, electronic filters) connected by information paths (e.g. BNC cables). An experiment will also probably make use of input- and output-only devices such as transducers and chart-recorders. `Grid` provides a model of this environment. The advantages that `Grid` can offer are almost instant reconfigurability, no equipment shortages (e.g. software can emulate as many FFT boxes as required) plus many functions which are not easily performed outside of a computer. For instance, data may be archived in a database and successive experimental runs correlated with existing data, or powerful statistical analyses may be applied to the archived data. However, some operations are not likely to be efficient in the digital domain (e.g. high-bandwidth control-loops, certain analogue filters) and therefore we are suggesting that `Grid` be used as part of a hybrid system. Put simply, where flexibility and reconfigurability are important, the computer wins, and where efficiency and speed of operation are important pure hardware wins.

`Grid` presents a conceptually simple extension of the laboratory scenario, helping experimenters to concentrate on the behaviour of the detector rather than the computing systems. We hope that the advantages gained by the use of `Grid` may speed both the commissioning of the detectors and the development of analyses for various gravitational wave sources. Indeed, the group at Cardiff are eager to begin using `Grid` on data taken during the 100hr Munich/Glasgow prototype run. We have around 6 man-years committed to the development of `Grid`. Although this is not a great deal of time, when compared to the manpower investment of other large software projects, we are using more sophisticated software development techniques than were available previously, and we only have to develop `Grid`'s framework. As will be described later, the system has been designed from the outset to be open. Users may easily expand it by adding programs written in a language with which they are familiar.

6.1 `Grid`'s visual interface

`Grid` provides the user with a "software construction kit", adopting some of the successful concepts from current graphical user interfaces. Independent information processing units are rendered as icons which can be dragged and dropped on an infinite mesh. These icons may have input and output connections, and these are displayed as drawings of sockets which decorate the exterior of the unit icon. The user can connect an input socket to an output socket by dragging a line from one to the other. This is the analogue of using a cable to connect two pieces of equipment. In fact, it is natural to speak of "wiring-up" processing units. The graph of connected units can be in one of two states — running or stopped. Such wiring-up can only be performed whilst

the network is at rest (stopped). Buttons which switch between the two states are displayed as part of `Grid`'s overall control panel. When running, data percolates through the analysis network, being transformed and processed by the units it passes through. Some units are dedicated to obtaining data from some external (to `Grid`) source, and some units archive data to some external place (such as disk). Naturally, some units are dedicated to the display of data (graph-drawing, histograms, oscilliscope-imitation). All data handled by `Grid` is represented in the standard file format described in section 4. Import and export units are available to convert data to or from formats (such as ASCII) expected by other software. Signal generator units are also available to design and inject test signals.

Units are obtained from a `ToolBox`. `ToolBoxes` display directories of units stored on disk and allow the user to traverse that disk's directory hierarchy. As many `ToolBoxes` as are required may be opened concurrently. Units are dragged by depressing a mouse-button when the cursor points to the required unit, and then moving the mouse while keeping the mouse-button depressed. The unit may be dropped where required by releasing the mouse-button. Units which have already been dropped on the mesh may be repositioned in precisely the same manner, the difference being that a copy is taken of the `ToolBox` unit, rather than actually moving it. As many copies of a unit as are required may be dropped on the mesh. The files representing different units are stored in the standard file format discussed in section 4. These files contain the source code for the unit as well as other information such as the unit's size and appearance on the grid.

The easiest way for a user to create a new unit is to copy one that most closely imitates the required functionality and then edit the source code appropriately. A menu selection from the `ToolBox` edit menu will automatically extract the source into the editor of the user's choice, and then recompile the unit when the user's changes are complete. All instances of the user's new unit on the mesh will then take on its new functionality. Put briefly, software development is possible from inside `Grid` permitting rapid testing of new units with real data. Note that we intend to explicitly support unit development in languages such as Fortran and C in addition to C++. This is achieved by supplying template units that store incoming data in a standard place and then call a user-supplied function name to perform the unit's processing. The user's function should store its processed data in another standard place and return. The template unit then outputs the data as normal.

As well as source-code unit construction described in the previous paragraph, units can be constructed from other units. Sub-meshes may be created by selecting a menu option from a `ToolBox`'s control panel. Sub-meshes are identical in operation to the main mesh except that they have input and output connectors on their mesh edges. The user adds input and output connectors through menu options available on the sub-mesh. The connectors may be dragged and dropped to arrange their positions anywhere on the sub-mesh's edges. A sub-mesh appears as a normal unit on the main mesh. The sub-mesh icon is decorated with connectors corresponding to those arranged on the edges of the sub-mesh itself. On the main mesh, units can be wired-up to the sub-mesh unit exactly as they would be to other units. There is no restriction on the number of sub-meshes that may be under development at any one time. Closing a sub-mesh window automatically updates a `ToolBox` entry for that sub-mesh. Copies of the new unit can then be dragged and dropped from the `ToolBox` as with any other unit. Double-clicking on a sub-mesh icon in the `ToolBox` opens a sub-mesh window for editing, if it was not open previously. Finally, sub-meshes can themselves contain further sub-meshes. Thus reusable new analysis units may be constructed without recourse to source-code programming.

Although we expect `Grid` to be mainly operated in colour, it can be started in monochrome and grey-scale modes also. Like most X-based programs `Grid` is very configurable and the user can redefine most supplied color defaults, and even the size of the meshes to suit the display in

question.

Selection of a unit which is already on the mesh is performed by pressing and releasing the mouse-button while the cursor points to the required unit. Selected units may be deleted by choosing the appropriate menu item from the mesh's edit menu. Double-clicking (that is rapidly selecting a unit twice) will cause the unit to open a window displaying more detail about its operation and/or allowing manipulation of parameters. Any number of the units may be opened in this manner at the same time. Windows may be opened while the analysis network is running. The parameters that are changed are local to the specific unit. For instance, many instances of a general-purpose filter may be on the mesh, each with particular parameters, and therefore, filter characteristics. Similarly many instances of an FFT unit may be on the mesh performing FFT's of different sizes as requested by the user.

The current analysis network may be stored at any time, and previously developed networks can be restored from disk. Once again, the network description files conform to the format described later. Conformance to the interchange file format will allow the different groups around the world to easily exchange not just data, but analysis programs as well.

6.2 Local concurrency and real-time issues

As hinted at in the previous section, each unit is an independent processor of data. The unit need not be concerned with any details of the network other than data arriving on its input connections, and sending data to its output connections. In practice, this means that each unit must be a separate *task*. Each unit is executed concurrently with all other units.

Under the operating system Unix (which is the target operating system for initial versions of `Grid`) this could be achieved in one of two ways.

1. Each unit is a Unix process.

2. Each unit is a separate thread of one Unix process, using some task-switching library.

Option 1 has the advantages of true concurrency as supported by the Unix system — no attention has to be paid to explicitly handing over the flow of control when a unit is not busy and the unit is woken automatically when data appears for it to process. However execution of such "blocking" system calls for true Unix processes incurs a fairly large performance penalty, and for this reason we have chosen Option 2. We are currently using a derivation of the AT&T C++ task library with extensions for real-time operation. This popular library has been in use since the inception of C++ and has performed admirably in many other projects. Although programming using such a task library is slightly tricky, only the implementors of `Grid` need to get involved in the grubby details, and the performance improvements can be dramatic[10].

Automatic wake-up and shutdown of tasks can be achieved by imbuing the connections between units with special properties. Connections inside `Grid` are actually queues — a data structure where items of data emerge in the same order as they were inserted. Actions on a queue can alter a task's state (idle or running) according to whether a queue is empty or full. Attempting to insert a data item onto a queue which is full will cause the inserting task to sleep, only to be woken when the queue drops below its maximum size. Attempting to extract an item when the queue is empty will cause the extracting task to sleep, only to be woken when a data item enters the queue. Queue sizes have to be chosen to be large enough to avoid deadlock (where all tasks are sleeping, waiting on empty or full queues) without causing inefficient delays. For example, a queue size of

[10]The C++ task library documentation compares true Unix processes with tasks and finds that for most operations the task library is *at least* an order of magnitude faster.

1000 when the items are computationally-expensive 32k point FFTs is far too large. Such a queue could slowly be filled when the network has already deadlocked elsewhere. In general therefore, queues should be as small as possible.

Collecting data from local real-time data acquisition peripherals is achieved by permitting certain units to be woken on receipt of a particular software interrupt. The data acquisition peripheral signals that a set of data has been acquired by causing an interrupt. This interrupt wakes the appropriate unit which then injects the data into the analysis network. In practice, this scenario is a little more complicated but once again the gritty details of such data acquisition units will only be of concern to `Grid`'s implementors. Real interrupt servers may well be provided which buffer data before sending a software interrupt to `Grid`.

As computing speed is finite, it will always be possible to overload a computer by attaching too many data acquisition peripherals. If the host computer does not have enough spare cpu cycles, interrupts may arrive before preceding interrupts have been serviced. `Grid` will inform the user if it fails to keep up with real-time in this way. The solution to such bottlenecks is to move some of the data acquisition units to another computer and connect the computers by a network. The higher the acquisiton rate, the more host computers you add, until the network itself becomes saturated. This distributed approach is discussed in section 6.3.

As accurate time-stamping of data is important, a clock signal will have to be sent to each of the computers from some master system clock. It is unclear currently whether an FDDI network could accurately distribute such a clock signal, and we suggest that we should assume that cables carrying such a master-clock signal will have to be run out to each computer separately. Such cables would have to be the same length to avoid transmission delay effects. Similar techniques are used in other large scientific instruments such as radio telescopes.

`Grid` maintains a real-time clock based upon this master clock signal. The clock value may be read at any time by a unit. Predicates such as **`before`** and **`after`** are defined for use by units. Data sent from a unit may be scheduled to arrive at a specified time (strictly, to arrive at the connected unit no earlier than the specified time). The connection queues can maintain a temporally-ordered sub-queue of such data which is dispatched as the appropriate time arrives. As described above, units waiting to receive data would automatically be woken when such "timed" data arrives. Units may also request that they be woken at specified times. In this way, units may be triggered (repetitively, if necessary) to begin processing at distinct times. For instance, units which perform various local control functions may be triggered in sequence to correctly "lock-up" the interferometer. As mentioned previously, the response of a traditional Unix-based system may be too sluggish for some functions and we would therefore make use of one of the Unix variants which have real-time extensions. However, these features do permit simple real-time programming to be performed *even across a network of computers*. In many cases we expect that the local control will actually be performed by some dedicated hardware hosted in the Unix computer. Units would communicate with such hardware, initiating various control functions when requested and reporting on the hardware's operation. Control hardware would communicate with units, in the same manner as data acquisition hardware, through interrupts.

6.3 Networking and `Grid`

So far we have considered the operation of a single copy of `Grid` on an isolated workstation. Whilst this is indeed a possible scenario, and may be adequate for some small-scale experiments, `Grid` takes a more general *distributed* approach, in line with today's technology. In most institutions computers are attached to a high-speed local-area network which permits fast and easy sharing of data between connected computers. `Grid` can exploit this connectivity by executing different

units on different computers.

This is achieved through a software technique known as remote procedure call (RPC). A description of a popular implementation of RPC can be found in [Sun 1986]. RPC permits procedures to be executed remotely. A call to a remote procedure is trapped by the local computer, the parameters are transmitted to the appropriate remote computer where the procedure parameters are unpacked and the procedure executed. Results from the remote procedure are then re-packaged and returned to the local computer. This technique is well-understood and RPC services exist as standard on all modern implementations of Unix. Remote procedures register with the RPC service on the computer on which they wish to execute. The RPC service can advertise the remote procedure to all the other computers on the network. When a program attempts to call a remote procedure, the local RPC services are queried to find the computer on which the remote procedure resides[11].

A unit which is to be executed remotely is first installed on the remote computer. The queues associated with the input connections to the unit are also moved to the remote computer, while the queues associated with the output connections of the unit remain on the local computer. RPC routines are inserted to transport data to and from the remote unit.

Consider a single connection from a local unit to a remote unit. The connection's queue resides on the remote computer and the queue's output is attached to the remote unit's input as normal. An RPC routine which resides on the remote computer is attached to the input of the remote queue. This routine registers with the remote RPC services indicating that it is willing to accept data from another computer. When data arrives from another computer this routine inserts the data into the connection's queue. On the local computer, the local unit's output is attached to another RPC routine. This routine sends data received from the local unit to the remote RPC routine. In this way, data is *transparently* forwarded to the remote unit. As far as the local unit is concerned, it is just sending data to a connection as normal. As far as the remote unit is concerned, it is receiving data from a connection as normal. All communication between distributed units is performed in the same way. The behaviour of units and connections is thus completely unaffected by distribution of units across a network. Neither units, nor connection queues need be concerned about their location. Units may be moved to another computer without changing any code in the rest of the analysis network. Instructing `Grid` that a unit is to be executed remotely will cause appropriate RPC routines to be inserted into all affected connections.

The remote unit is also sent pertinent user-interaction events such as double-clicking on the local unit icon. The X-window system permits such remote units to open windows on, display data on and receive user-interaction events from the local display.

Groups of connected units may also be executed remotely. Only the input and output connections of the group (the group's edges) need be transformed in the manner described in the previous paragraph. Intra-group connections are implemented in the normal way on the remote computer.

This relatively simple process distribution can be very powerful. Units can be executed on computers that have specific hardware to support their function. For example, FFTs can be sent to computers that have dedicated DSP chips, and more obviously data acquisition can be performed on the computers that actually contain the data acquisition hardware. Improvements in speed can also be gained by noting that a pipeline of processing units can be set up on separate processors. In one time-step, n units can be processing where n is the number of units in the pipeline.

We hope to provide load-balancing performance monitoring tools to allow successful analysis

[11] This is a little simplistic but provides us with an idea of what the RPC services do for the purposes of discussion.

networks to be distributed for maximum speed. Obviously, the execution of a unit remotely incurs some performance penalties due to the packaging and transmission of data across the network. These tools will help to judge whether the penalty/payoff ratio is advantageous. Whole analysis networks may be executed remotely if needs be. Thus, an experimenter with a cheap, portable workstation may perform sophisticated analysis of data from all over the detector from wherever he or she may be. This may be of significant benefit during the commissioning of the detector before we fully understand the behaviour of large-scale interferometers.

7 Conclusions

Surrounding the large-scale detectors by a high-speed computer network appears to offer considerable rewards. Use of readily available hardware such as FDDI and software standards such as TCP/IP seems to be the most cost-effective way of gaining those rewards. Adoption of TCP/IP as a network protocol standard allows standard software development environments to be used. Unix is a mature and popular operating system that has built in support for networking using TCP/IP. Many commercial software packages are available to run on networked Unix platforms. However, should custom software need to be designed, Unix offers an environment that most commercial software developers would be happy to work in. Therefore, the combination of FDDI, TCP/IP and Unix as the base for the detector's computer network looks most promising and we would propose that this combination be adopted by the different research teams. Note that this suggested hardware/software platform will not preclude different software solutions in the unlikely event that the `Grid` effort does not achieve its potential.

Some form of data format standard is needed for transmission of data around the network, archival of data and exchange of data between research groups. We are proposing `BIFF` as such a standard. `BIFF` is lightweight and pragmatic without imposing restrictions on what may be stored. `BIFF` was designed to dovetail neatly with current CPU and network design. The software that is being developed at Cardiff stores data exclusively in `BIFF` form. Exclusive use of `BIFF` may not be so attractive to other research teams and we would like feedback about file formats. However, we propose that `BIFF` be recognised as a data format standard by the interferometer community. Agreement on a data format standard soon could save us all a lot of time and effort later.

We intend to investigate the capabilities of the new generation of object-oriented databases further. Archival of detector data in such a database could offer considerable benefits. Powerful searching and querying of a database could ease both commissioning of the detector and in-depth studying of observation runs. We intend to provide access to such a database from within `Grid`. This would permit more sophisticated analyses to be quickly developed. For instance, on discovering unwanted noise in a particular signal, the database could be queried to discover when the noise began and to search for correlations in other signals in an attempt to discover the noise source. Several periods of operation with different control parameters could be compared. Integration of database functions within `Grid` is one of our development goals.

`Grid` is principally an analysis prototyping system. Interactive development and testing of sophisticated time-series analysis algorithms is possible from within `Grid`. `Grid` will be an important system just from this point of view. However, `Grid`'s distributed approach and real-time features mean that *much of the operation of the detector* could conceivably be controlled and analysed from within copies of `Grid`. We are intending to vigorously develop `Grid`'s features over this coming year. We hope to collaborate closely with the various interferometer research teams so that needed features are incorporated and redundant effort is avoided. `Grid` could have a significant impact on the speed of detector commissioning as well as aiding the rapid development of new

analysis methods. If the various standards proposed in this document are adopted then `Grid` will be available as a powerful tool to research teams with a minimum of porting effort. We intend to release successive versions of `Grid` to the interferometer community as they are developed.

References

[Booch 1991] Booch, G., "Object oriented design with applications.", USA : The Benjamin/Cummings Publishing Company, Inc.

[DARPA 1981a] Defense Advanced Research Projects Agency, "RFC 793: Transmission Control Protocol.", USA : DARPA

[DARPA 1981b] Defense Advanced Research Projects Agency, "RFC 791: Internet Protocol.", USA : DARPA

[Digital 1980] Digital, "Introduction to Local Area Networks.", USA : Digital Equipment Corporation.

[Ellis and Stroustrup 1990] Ellis, M.A. and Stroustrup, B., "The Annotated C++ Reference Manual.", Addison Wesley

[Kim 1990] Kim, W., "Introduction to Object Oriented Databases.", USA : MIT Press

[Linton et al. 1991] Linton, M.A. et al., "InterViews Reference Manual Version 3.0.1.", USA : Stanford University

[Miller, Boyle and Stewart 1984] Miller, C., Boyle, R. and Stewart, A., "Unix for Users.", UK : Blackwell Scientific Publications.

[Quercia and O'Reilly 1988] Quercia, V. and O'Reilly, T., "X Window System User's Guide.", USA : O'Reilly and Associates

[Sun 1986] Sun Microsystems, "Remote Procedure Call Programming Guide.", USA : Sun Microsystems, Mountain View, California.

[Watkins 1991] Watkins, W.J., "Analysis of Gravitational Wave Signals.", University of Wales : PhD Thesis