A proposal for a distributed virtual reality system for architectural simulations

Citation for published version (APA):

Document status and date:
Published: 01/01/1995

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 06. Oct. 2019
A PROPOSAL FOR A DISTRIBUTED VIRTUAL REALITY SYSTEM FOR ARCHITECTURAL SIMULATIONS

A.H.A. Brouwer
W.A.H. Roelen
J.A.G. Jess

BIT Notes 1995 / 1
Other issues in the series of BIT Notes are:

94/1 Future Organisation of the Building Process Information Technology in the Building Industry.
Geert T.A. Smeltzer.

Geert T.A. Smeltzer, Jan Dijkstra, Rob H.M. van Zutphen.

Geert T.A. Smeltzer, Jo M.M. Mantelers, Walther A.H. Roelen.

94/4 NOBI / BBB Project Bouwkosten en bestek Rapport prototype

94/5 NOBI / BM Project Het Intechno Bouwkundig Model Eindrapport
Erik P. Deiman, J.C. Hubers, Jos P. van Leeuwen, Geert T.A. Smeltzer,
Harmen C.A. Tacke, Bauke de Vries, Rob H.M. van Zutphen.

94/6 Architectural Product Modelling
A Case Study.
Jos P. van Leeuwen, Rob H.M. van Zutphen.

A Proposal for a distributed Virtual Reality system for Architectural simulations
A.H.A. Brouwer
W.A.H. Roelen
J.A.G. Jess

Keywords: CAD.

© 1995 Technische Universiteit Eindhoven,
faculteit Bouwkunde, sectie Bouwinformatica.
Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd en/of openbaar gemaakt door middel van druk,
fotocopie, microfilm, electronische media of op welke andere wijze ook, zonder voorafgaande schriftelijke toestemming van de uitgever.
All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, transmitted or utilized in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher.
Contents

Contents .................................................................................................................................... i

1. Introduction .......................................................................................................................... 1

2. Other Virtual Reality Research .......................................................................................... 3
  2.1. Veridical User Environments ....................................................................................... 3
  2.2. MR Toolkit ................................................................................................................... 4
  2.3. SIMNET/DIS ............................................................................................................... 5
  2.4. DIVE ........................................................................................................................... 6
  2.5. WAVES ....................................................................................................................... 6
  2.6. NVR ............................................................................................................................. 8

3. Specification ......................................................................................................................... 11
  3.1. Definitions ................................................................................................................... 11
  3.2. Background .................................................................................................................. 11
  3.3. General description ..................................................................................................... 12
  3.4. The innards of CALView ........................................................................................... 13
  3.5. Event Manager ........................................................................................................... 14
    3.5.1. Event classification .............................................................................................. 15
    3.5.2. Event description ............................................................................................... 16
    3.5.3. Event processing ............................................................................................... 16
  3.6. Communication process ............................................................................................... 19
    3.6.1. Overview of the CCM ....................................................................................... 20
    3.6.2. CCM interface ................................................................................................. 21
    3.6.3. Testing the CCM ............................................................................................... 21

4. CALView Communication ................................................................................................. 25
  4.1. Internal communication ............................................................................................... 25
    4.1.1. ToolTalk ............................................................................................................. 25
    4.1.2. Distributed objects .............................................................................................. 27
  4.2. External communication ............................................................................................... 28
    4.2.1. ISIS ..................................................................................................................... 28
    4.2.2. Horus ................................................................................................................. 30
    4.2.3. DOME ............................................................................................................... 32
  4.3. Implementation of the CCM ....................................................................................... 33
    4.3.1. Functional description ......................................................................................... 33
    4.3.2. Application programmers interface .................................................................... 34

5. Future Developments ......................................................................................................... 37
  5.1. Behaviour with multiple participants .......................................................................... 37
  5.2. The user interface ........................................................................................................ 38
  5.3. Implementation issues ................................................................................................. 38

6. Conclusions ........................................................................................................................ 39

7. Bibliography ........................................................................................................................ 41

A. Event Description ............................................................................................................. 45
  A.1. Lexical scanner ........................................................................................................... 45
  A.2. Header files ................................................................................................................. 46
    A.2.1. ScanEvent.h ...................................................................................................... 46
A.2.2. String.h ................................................................. 47

B. CCM Header Files ............................................................ 49
  B.1. config.h ................................................................. 49
  B.2. Error.h ................................................................. 51
  B.3. ccm.h ................................................................. 51
1. Introduction

The Computer Applications Laboratory in Building Research and Education (Calibre) is an institute conducting research for the university, the building industry and the government. Some areas of interest are CAD, visualisation, light/colour simulations and planning systems. It is closely connected to the faculty of Building Research and Education of the Eindhoven University of Technology.

Currently the institute is involved in a pilot project called Asterisk: a collaboration between the Calibre Institute, Sun Microsystems Nederland B.V. and Autodesk B.V. Finding the importance of multi-media technology within the architectural design process is the question that is to be solved by this project. The research goal was defined as follows: create prototypes of virtual reality applications for demonstrations. These demonstrations generate data that is going to be used to obtain the necessary information in order to be able to introduce commercial virtual reality systems [Smeltzer93b].

It was necessary to develop software to design, visualise and interact with 3D models. The current toolset consists of three tools. In figure 1.1 the relation between these tools can be seen. The design can be done in DynaCAD which is a tool written for AutoCAD. It relieves the task of designing and describing a 3D model. Rayder is a set of programs which calculate the lighting properties of this model. It is based on radiosity techniques. CALView can be used to interact with the final outcome of the first two steps [Smeltzer93a].

At first CALView was written for stand alone usage only and in a later stage it had to become a distributed system. That would allow several people to participate in the design process. These people are part of a building team and are geographically dispersed over one or more countries. A communication layer had to be developed to enable multi-user, multi-site usage of the program. This was the goal of my master's research. In order to achieve this goal I had to divide my research into three parts:

1. Investigate current distributed virtual reality solutions using literature research.
2. Write an initial specification of the main parts of CALView.
3. Develop the software to enable distributed virtual reality in CALView.

At the time of writing a simple communications layer based on TCP socket streams is developed for the prototype of CALView. Now the internet can be used to connect two members of a building team. The current prototype will be used to test the 3D user interface between

\[ \text{Figure 1.1. Calibre VR toolset overview} \]
the Calibre Institute at the Eindhoven University of Technology and the Kantoor van de Toekomst\textsuperscript{2} in Rosmalen, see figure 1.2. Both sites use the same peripherals: stereo glasses and a 3D mouse. In a later stage the communication layer could be changed to use a more sophisticated commercial package to enable more than two persons to interact within CALView. There are three solutions which seem to be fit for this task: ISIS, Horus and Dome. The choice depends on the programming language used for the implementation. This report describes the results of the literature research which have formed the basis for the specification. The specification of the event processing and the communication layer in CALView is presented. Then the commercially available alternatives for communication with several instances of CALView are examined. This is followed by the implementation details of the communication layer. Before coming to the conclusions, future developments are discussed.

\textsuperscript{1} Internet without capitals means the global internet [Comer88].
\textsuperscript{2} Office of the Future
2. Other Virtual Reality Research

The projects described in this chapter use different methods for communication depending on the goal of their research. Therefore it is important to define the goal of the distributed virtual reality application before an implementation decision regarding the network communication can be taken.

In [Kazman93b] topologies of virtual world architectures are described and divided into two broad categories: tightly connected and loosely connected. Communication for these categories relies on a client-server architecture (loosely connected) or upon a static allocation of processes to processors communicating with each other along predetermined paths (tightly connected). See [Tanenbaum89] for a description of these communication techniques.

The architectures that have been investigated are:

1. Veridical User Environments
2. MR Toolkit
3. SIMNET/DIS
4. DIVE
5. WAVES
6. NVR.

These architectures found during my literature research are discussed in the following sections. The acronym VR is used to denote "virtual reality."

2.1. Veridical User Environments

In the Veridical User Environments a hybrid of client-server and broadcasting techniques (called a distributed client-server paradigm) is used for communication with two users of the VR world, see [Appino91] and [Codella92]. A dialogue manager is associated with each user, it broadcasts messages to other users in the VR world. For example, the position of the hand of one user has to be sent to other users The hand can then be mapped onto their representation of the world.

![Diagram of VUE architecture](image)

*Figure 2.1. VUE architecture*

The dialogue manager is a client process It receives events from input servers and applica-
tions and sends information to output device servers and applications, see figure 2.1. It serves as an intermediary between the servers, although large quantities of data necessary for the renderer are not mediated through the dialogue manager but are sent directly to the renderer for performance reasons.

[Kazman93b] notes two potential problems when enlarging the world:

1. Low-level events (e.g. hand movements) are communicated to all dialogue managers, thus large amounts of interaction could swamp the communication channels.
2. Because events are broadcast, not all information sent to the dialogue managers might be necessary in a large virtual space (e.g. a user walking in a room without being able to look into other rooms is not interested in events happening in those other rooms).

2.2. MR Toolkit

The Minimal Reality Toolkit [Shaw93] structures VR worlds as strict hierarchies. One process is designated the master and all other processes are either slave processes (typically private output devices such as a head-mounted display), server processes (shared I/O devices such as trackers or sound generators) or computation processes (simulations). [Kazman93b] notes that because of this rigid hierarchical structure, communication between processes is limited to the parent-child relation. Thus servers, slaves and computation processes can only communicate through the auspices of the master process. In a complex VR world the master process would quickly become a bottleneck, since all control eventually resides at the server.

The peer package is an extension of the MR Toolkit that allows a master process to communicate with other master processes on other machines. Any running MR Toolkit program may start up the peer package at any time. It may initiate and quit communications with other processes at will. Each MR Toolkit master program, or peer, maintains a list of other peers that it is connected to. Peers are connected pairwise and a peer may send a message to any or all peers at once.

Communication with the peer package is based on the User Datagram Protocol (UDP), a connectionless transport service from the TCP/IP suite of services [Comer88]. The advantage of using UDP is that it is connectionless thus connection management is simple. Packet communication is one-way and communication errors do not result in error returns from the recipient. However the disadvantages are that UDP only guarantees that when a packet arrives it is error-free but it does not guarantee delivery. It is also possible with UDP to receive packets out of order. These can be severe limitations for wide area networks (WANs). Another disadvantage of the peer package is that it maintains a complete graph connection topology. Although the peer package does not have to concern itself with routing, the

---

1. Modelled after the postal system. Each message carries the full destination address, and each one is routed through the network independent of all the others [Tanenbaum89]. Compare this to the connection-oriented service TCP which is modelled after the telephone system.
number of packets transmitted is in the order of the square of the number of participants. With more than five participants traffic growth becomes unmanageable resulting in growing network delays and potential packet loss [Tanenbaum89]. The latter disadvantage could be solved by using multicasts, but this is not available on all workstations or not usable without having specific privileges on the workstations where it is available.

2.3. SIMNET/DIS

The SIMulation NETworking system [Calvin93] has been designed to allow large numbers of entities from disparate organisations to interact in a VR world, in real time, and over large geographic distances. It is primarily designed for military simulation purposes.

The requirements for this architecture are as follows:

• 100s to 100,000s of geographically distributed entities are supported
• simulations are heterogeneous and computations are distributed (no central site)
• the system is low cost (for military budgets)
• the system operates in real time.

To meet these requirements, the following key design decisions were made:

• The model is object and event-based. Objects in the VR world interact with each other through a series of events.
• Objects are autonomous. All events are broadcast and are available to all interested objects. An object initiating an event does not calculate which other objects might be interested or how the receiving objects can be affected by it. One event may generate other events. A side benefit of using autonomous objects is that objects may join or leave a VR world at any time.
• Each object transmits the absolute truth about its state. The receiving objects are responsible for transforming absolute truth information into a model of the real world.
• Objects transmit information only about changes in their states to minimise requirements imposed on communications processing and bandwidth. This reduces the transmission and reception of redundant information.

Although these design decisions are necessary to support large and geographically dispersed entities, the SIMNET architecture protocol is heavily based on military simulation. It requires fast local area networks and WANs with real-time allocation gateways. The local area network (LAN) is based on ethernet but uses a non-standard way of multicast which makes it difficult to port to some workstations available today.

Distributed Interactive Simulation (DIS) is a newer simulation standard currently in development, it shares the goals and purposes of SIMNET but allows for greater complexity and realism. Unlike SIMNET, DIS does not require any non-standard network coding for ethernet. But it shares, together with the goals and purposes, the same main disadvantages of SIMNET.
2.4. DIVE

The Distributed Interactive Virtual Environment (DIVE) is a heterogeneous distributed VR system based on UNIX and Internet networking protocols. Each participating process has a copy of a replicated database and changes are propagated to the other processes with reliable multicast protocols, see [Carlsson93a], [Carlsson93b] and [Carlsson93c].

The model can be described as a database shared over a network. Processes interact by making concurrent accesses to the shared memory and by sending messages to each other. The database is partitioned into worlds. Each world represents a specific set of objects and parameters completely distinct from other worlds. A DIVE process is a member of exactly one such world at a time, although it may change worlds dynamically, resulting in a complete switch of context. Such a process can be either a representation of a human user or an application. A process acts on a world by modifying objects and parameters and by sending messages to the other processes of that world. Internally, each process runs multiple light weight processes called threads. Each thread is responsible for a certain task such as rendering, controlling I/O devices or updating the database.

Worlds are implemented using ISIS process groups [Birman93]. A process group is a set of processes which may be addressed as one entity: messages addressed to the group are relayed by multicast protocols. The prime advantage with the process group abstraction in DIVE is the ability to manage replicated data with atomic multicast protocols and achieve high availability by fault-tolerant computing algorithms. A world is a process group where each participating process manages its own replica; a complete copy of the database is kept in primary memory. From an application point of view the replication is transparent, a process only modifies an object in what it sees as a shared database.

The concurrency control mechanism uses mutually exclusive locks, reliable source ordered multicast and distributed object locks to ensure consistency between the copies of the replicated database. The current implementation has a limit of about 11 systems to achieve real-time distributed VR. With more than 11 systems the latency is becoming worse and makes the system unusable.

2.5. WAVES

The WAterloo Virtual Environment System tries to take a flexible approach towards managing, integrating and communicating between processes in a VR world, see [Kazman93a] and [Kazman93b].

In this system, a VR world is comprised of a number of message managers which mediate communication between a number of hosts. Hosts are processes which simulate a number of objoids in the VR world. The term objoids is chosen to stress the difference between the object from object-oriented programming languages and an object in a VR world. The hosts also provide some services to these objoids: simulation, data logging, interaction detection.
and resolution, switchboard routing and rendering. A host does not have to control an I/O device, see the following figure 2.2. The VR world is composed of nothing but objoids.

![Figure 2.2. The WAVES architecture](image)

In a distributed VR world, each host only simulates a subset of the objoids in the virtual world, so a collection of hosts is necessary to create an entire simulated world. Some of these objoids will be native to a particular host and potentially under the control of a human user associated with that host. Objoids can also be clones of objoids which are native to other hosts, and which have been communicated to the user’s host in order to provide a realistic shared world.

The message manager mediates communication between hosts and relieves the responsibility of knowing who to communicate with from individual hosts. This allows for flexible worlds, were hosts can enter and leave dynamically, and allows for a smooth scaling of world complexity.

Another important function of the message manager is to filter, upon request, messages which a particular host receives. This is crucial to minimise levels of communication in a complex world, where most events will not be of interest to any given host. Simple message broadcasting strategies would not be appropriate (e.g. in a large virtual world any given user will only view a small subset of objoids in that world). Because of the sparseness of information which any user needs to see at any given time (compared with all the information contained in a virtual world) intelligent filtering can dramatically reduce the message volume in a system.

The message manager can not only transmit messages from one host to others, but can, in special cases where data volumes are high and minimising latency is crucial (e.g. a direct video feed), delegate point-to-point communication between hosts. The message manager will actually be a family of distributed, interconnected managers for large virtual worlds.

Objoids are autonomous entities which encapsulate their own state and possess an explicit model of behaviour. Using this model, objoids can update their state in real time. These models can even be used to predict an objoid’s state in order to accommodate for system latencies. Objoids do not control their rendering in the world, in this way the distinction between
functionality and presentation is maintained. Hosts can (but need not) control some input or output device. However, hosts can serve other purposes: e.g. interaction detection. This is such a complex task in any complex world it must be distributed as well.

Hosts and the message manager will maintain a cache of objoids they are interested in and can update this cache at any time. Without such a cache, the WAVES architecture devolves to a distributed server architecture with broadcasting as the only mechanism of keeping the world consistent.

The overall result of all the measures described above is that the amount of inter-objoid communication is greatly reduced as compared with the client-server or point-to-point paradigms. When an objoid’s behaviour changes, only hosts which are interested in that objoid are notified due to the filtering technique in the message manager. A virtual world is thus a set of cooperating processes which can communicate by virtue of subscribing to a communication service. The structure is completely determined by the dynamics of the simulated world and can be easily scaled by adding more hosts, objoids and message managers.

2.6. NVR

The Networked Virtual Reality system has low cost as its main asset [Berger94]. It uses the TCP client-server approach, were the server is a UNIX system and the clients are PC's. Communication between clients takes place through the server. This approach has been chosen because of the lower software complexity. The World Toolkit\(^1\) is chosen as the underlying VR platform.

Each client maintains its own copy of the virtual world and only changes are transmitted to other clients. These changes are absolute and not relative to previous values. Since any client can change any object in the world a locking mechanism is available, however the responsibility for avoiding deadlock lies with the world builder. While obtaining a lock the client will not react to the user until the lock has been granted or denied.

To overcome the possible overcomplexity of a virtual world, a world can be broken up into smaller ones connected by portals. In the clients only the data of the current world is available. When the user changes to another portal, a copy of the new world has to be transferred to the client. This also implies that the server keeps track of which clients are in which worlds and sends updates only to the relevant clients. The set of worlds connected with portals is called a universe. This way of breaking up a complex virtual world ensures a high level of performance since clients only have to control part of this universe.

Another interesting feature is called thresholding. Clients send their changes once per frame to the server which can lead to network congestion or a slower client may run behind in processing the incoming changes. Thus a client can limit the number of transmissions every

---

\(^1\) World Toolkit is a commercial rendering package created by Sense8 Corporation.
nth frame. Note that because NVR transmits absolute changes that this can happen without any problems for the other clients.

Users experienced variable station-to-station lag because of the usage of internet. Two factors have been identified as the cause:

1. queuing delay at intermediate nodes because of congestion
2. lock acquisition delay.

When travelling through a portal users also experience lag because the new world has to be transmitted to client and the local software has to update its database and graphics.

Some future improvements which are going to be researched are:

- using the hierarchies which occur naturally in 3D modelling in the network
- callbacks, which means that other clients will be notified that an event occurred at another client without continually querying the database
- support of fault-tolerant transmissions.

Other network architectures than the client server approach might also be investigated.
3. Specification

It is important to create a specification because it will define the goals which have to be satisfied by a program written according to that specification. Although several approaches are feasible we have used the trial-and-error approach, i.e. we specify part of the CALView program and try to implement it and adapt the specification according to the results of the implemented program. Because of time limits and change of scope in the course of the research only event processing and the communication process have been described. A prototype of CALView was already created whilst this specification was being devised and this has had some influence on the design.

3.1. Definitions

This section defines some important notions which are being used in the following sections.

\[
\begin{align*}
\text{world} & := \text{scene}^+ \\
\text{scene} & := \text{properties}^+ \\
\text{properties} :&= \text{object}^+ \\
\text{object} & := \text{user} \mid \text{block}^+ \mid \text{light} \\
\text{user} & := \text{block} \& \text{"a user can manipulate properties"} \\
\text{block} & := \text{polygon}^+
\end{align*}
\]

3.2. Background

In the following query and answer dialogue some aspects of CALView are discussed. This is necessary before the specification can be written.

- What is the scope of the VR program?
  
  *To show distributed architectural simulations is the primary use.*

- How big do the VR-worlds have to be? This question reflects upon the number of scenes and the properties within such a scene.

  *There is only one world, the number of scenes: 45, number of properties: 255 per scene, the individual properties are limited by the AutoCAD limits and performance limits.*

- How many people can walk in a world simultaneously?

  *A building team consists of about 6 people and this is the maximum number of users allowed.*

- What amount of lag is acceptable?

  *Depending on the way of interaction this can range from several hundred milliseconds for interaction to several seconds for file transfers.*

- What about future expansion? Some possibilities are enlarging the number of properties within a scene, adding new kinds of behaviour to the properties, etc.

---

1. These numbers are taken from experience with the ESPEC project.
New kinds of behaviour should be added in a later stage to the properties. It should be possible to present non-geometrical attributes within the program, for example: price information for parts, mechanical stress analysis, temperature distribution. But it should also be possible to add architectural legislation, planning and design rules.

3.3. General description

The metaphor of a meeting with a speaker is used for CALView. The speaker, or rather information manager, opens the meeting. The speaker regulates all communications. This implies that the machine of the speaker starts the session and notifies the other machines of the meeting by establishing connections. The speaker's machine also transmits the necessary data with which the other databases will be initialized. See figure 3.1 on page 12 for an input/output model of CALView. The file input and output consists of two kinds of files:

- CAS, which means CASe as well as Computer Aided Simulation and contains data for a special case (status of properties, lighting).
- CAL is an abbreviation for Computer Aided Lighting and contains the geometry information for the simulation.

![Diagram of meeting metaphor and input/output model of CALView](image)

*Figure 3.1. Meeting metaphor and input/output model of CALView*

Each building team member, called participant henceforth, can be identified by a unique symbol.

In a meeting only the speaker, or rather information manager, has permission to change the geometry of a scene by changing objects in that scene. Participants may change these properties in their local databases: the colour and placement of lights.

Participants follow the speaker as he visits scenes, they cannot change to another scene without him changing it for them. However they can move around the scene individually, change
the lights on their local copy of the scene. As noted before they cannot change the geometry of the scene.

This ‘meeting’ metaphor implies that all CALView communication must be consistent. This means that the connections to the other participants should be error-free and that data should arrive synchronously and in an order consistent with causal dependencies [Birman93].

3.4. The innards of CALView
Let user imply the local participant of the meeting. The blocks in figure 3.2 are going to be discussed in this section.

- Error Manager: this block handles all unusual situations which can occur in the other blocks. The Error Manager keeps track of all generated errors and tries to take action to alleviate the problem. Each block of CALView must register itself and inform the Error Manager of the errors it can generate along with a means of measuring the severeness of the error. The Error Manager is now able to take appropriate measures (by taking into account the severeness of the error) by submitting an event to inform the user (and possibly the participants) of the problem.

- Event Manager: several processes can be seen which generate events. These events are handled by the Event Manager. The Event Manager reacts upon events from the user, the modules and the events generated by other participants. It keeps track of changes to the properties in a scene and is responsible for the computation for all participants if the user changes a property or the behaviour of a property. E.g. the user picks up an object and drops it, the local machine computes the changes and sends events to the other participants. Note that this might cause network load, but it is necessary to keep the world consistent with the different copies of the world.

- The model database which contains all relevant data for the model being shown in
CALView. At start-up the data of the model is read from a file into this database and is then distributed to the participants. The database is a repository of all data which is necessary for rendering and sound generation. However when keeping in mind some of the extensions proposed earlier, it might be necessary to expand the capabilities of the database by adding other data which has to do with modelling.

- Representation: generates the data for the perceptive output, e.g. generating sound and rendering images.

The ovals in figure 3.2 represent capabilities of the system which may or may not be enabled. Each capability contains processes responsible for some properties of the system:

- modelling: processes for the creation and editing of the model (e.g. DynaCAD)
- simulation: processes responsible for gravity, collision detection and possible other dynamic processes such as animation or Rayder
- communication: the networking process which keeps in contact with the other participants.

3.5. Event Manager

The Event Manager is responsible for routing the events to the processes contained in the modelling, simulation and communication capabilities. The processes then compute the new situation and will send events to the Event Manager. Figure 3.3 shows the processes which are necessary for a version of CALView with basic functionality.

![Functional overview of CALView](image)

*Figure 3.3. Functional overview of CALView*
Because of the modular structure, new processes generated by the modeller can be attached to the Event Manager.

When an event is generated, it will be sent to the Event Manager which then performs the following in one atomic action:

1. The event gets a unique number $n \in N$.
2. It will be classified and put in a queue for processing.

The Event Manager maintains a queue wherein the events are placed and from which they will be sent in the order of arrival to the process which can handle the event.

The next section defines and classifies all events. The speed of the Event Manager has an enormous influence on the total speed of CALView. It is therefore necessary to make it very efficient.

3.5.1. Event classification

The events are divided in five classes: user, modelling, simulation, communication and database events. Each of these classes contain special events bearing importance to that class only. Table 3.1 gives some examples of events and their classification.

Table 3.2 shows the relations between event classes.

<table>
<thead>
<tr>
<th>user</th>
<th>simulation</th>
<th>modelling</th>
<th>database</th>
<th>communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>3DEvent</td>
<td>objectMoves</td>
<td>modelChanged</td>
<td>dump</td>
<td>incoming</td>
</tr>
<tr>
<td>voiceEvent</td>
<td>objectCollides</td>
<td>loadModel</td>
<td>change</td>
<td>outgoing</td>
</tr>
<tr>
<td>mouseEvent</td>
<td>lightOn</td>
<td>saveModel</td>
<td>add</td>
<td>start</td>
</tr>
<tr>
<td>keyEvent</td>
<td>lightOff</td>
<td></td>
<td>delete</td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>objectColour</td>
<td></td>
<td>load</td>
<td></td>
</tr>
<tr>
<td></td>
<td>changeChairman</td>
<td></td>
<td>save</td>
<td></td>
</tr>
<tr>
<td></td>
<td>personMoves</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>personHand-Moves</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.1. Event classification**

<table>
<thead>
<tr>
<th>action event</th>
<th>resulting events</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>simulation OR database OR modelling</td>
</tr>
<tr>
<td>simulation</td>
<td>simulation AND communication</td>
</tr>
<tr>
<td>modelling</td>
<td>database</td>
</tr>
</tbody>
</table>

15
All possible events which can be generated by the processes which interact with the Event Manager have to be described and classified.

3.5.2. Event description

How can these events be described? The following idea was taken from [Hilderink94] and [Heijligers94]: a text format with a Lisp-like syntax is used, where each keyword is preceded by an open bracket, followed by its arguments and a close bracket. See figure 3.4 on page 16 for an example. Refer to appendix A see page 45 for the definition of this structure.

Advantages of this method are:

- The ASCII format makes debugging easier.
- This format can be extended without disturbing the existing data structure: unknown keywords can be skipped by processes which don’t understand them.
- The history is always a part of the event, because different processes can add their specific information at the end of the data structure of the event.
- If the Event Manager is implemented properly, new processes can be added easily without having to recompile the complete system (i.e. at run time).

How to use this structure in the processing of the events is the subject of the next subsection.

```plaintext
(event
  (version 1)
  (id 14687)
  (source "3Dmouse"
    (class "user"
      (type "3Devent"
        (x 1.9) (y 2.4) (z 3.8) (dx -3.6) (dy -2.3) (dz 0)
        (roll 2.1) (pitch 5.67) (yaw 9.6) (d_roll -1.0) (d_pitch 1.4) (d_yaw -2.6)
        (buttonDown "YES") (buttonUp "NO") (buttonNumber 1)
      )))
)
```

Figure 3.4. Example of the textual representation of an event

3.5.3. Event processing

Only the speaker can move and transform properties. Therefore a linear event structure from which events are processed sequentially is sufficient. The user will generate events by mov-
ing around and handling properties. These events have to be processed in a linear order to maintain the causality of the interaction. However once we will allow independent animation processes or if more than one user may handle properties, the linear event structure will not be sufficient.

Before going into details, we will present an example of the processing of an event. When the user uses the 3D mouse to move a selected object, a 3DEvent will be generated. This event is sent to the command interpreter which in turn generates an objectMoves event. The latter will be sent to the collision detection process, then to the Representation Manager and the gravity, and from the gravity to the communication process. The event can now be discarded, see figure 3.5 on page 17.

![Event Processing Diagram](image)

**Figure 3.5. Example of event processing**

An event can be handled by several processes simultaneously but only one process will alter it. Take for instance the processing of the 3DEvent mentioned earlier. After the objectMoves event has been processed by the collision detection process, it will be sent to the Representation Manager. This results in an update of the users' displays and the gravity process will add data to the history of the event.

But when are events removed? Factors of importance are: the history and the age of the event. By the age of an event we mean the time it has been idle in the event queue.

The history is of the utmost importance in determining the deletion of an event. This also
depends on the class of events.

*Event graphs* representing the input and output of events will help to determine when an event can be removed.

Definitions of the edges of an event graph:

- \(\rightarrow\) : an event is submitted into the event queue
- \(\rightarrow\) : an event is forwarded, processes might have added data
- \(\rightarrow\) : the event can be deleted after it has been received by the process
- \(\rightarrow\) : the process gives confirmation of the reception of the event.

The event graph in figure 3.6 describes the input and output for the following user processes: 3D input, mouse/keyboard input and speech input. The event will be submitted by one of these processes and sent to the command interpreter process. If the submission is successful the event must be removed from the event queue.

![Figure 3.6. User input processes event graph](image.png)

The command interpreter process will submit an event after interpretation of an incoming event. Depending on the event class, the submitted event can be intended for the simulation processes. The graph in figure 3.7 describes the behaviour of the collision detection and gravity processes. The grey box denotes the submission of this event as a direct result from an event generated by another process, be it on the local host or from another instance of CALView. The Representation Manager might have to request data from the database, see figure 3.8, but this has no relation with further processing of the original event.

Going back to events coming from the simulation processes, it can be seen that after the gravity process the event is sent to the communication process. The latter can also receive events from other instances of CALView. These received events have to be submitted into the event queue, see figure 3.9.

During the processing of events, database queries are also done with events. The graph representing these operations can be seen in figure 3.10.

After having described these graphs, the following strategy can be used to determine the age of an event: depending on the event class we will use a different *event time-to-live* (ETTL). This ETTL is based on the total number of events processed by the Event Manager. When an
event is given an unique id, a timestamp is added. Each process adds a timestamp to the
event. With this event history the ETTL can be computed. The Event Manager must check
the queue to remove old entries on a regular basis.

3.6. Communication process
The communication module in figure 3.3 on page 14 is responsible for the initialization,
maintenance and termination of the communication with other instances of CALView. In this section this process is defined. Henceforth the communication module in CALView is called the CCM (CALView Communication Module).

### 3.6.1. Overview of the CCM

In the following figure 3.11 several distinct parts of the CCM can be seen. The Error Manager is already known from CALView and is enhanced to take into account errors occurring in this process.

There is an abstraction layer between the basic communication interface and the Event Manager. The communication events are high level commands and have no notice of the underlying communication protocol. Therefore it is possible to change the communication protocol without requiring a rewrite of the Event Manager. This is necessary when the speaker concept of CALView changes (e.g. several participants may change and move properties at the same time, or the maximum number of users is raised).
3.6.2. CCM interface

The interface of the CCM defines how programmers should let their code interact with the CCM. Figure 3.12 shows the front-end for the communication interface. In this chapter the communication interface is left unspecified, all possible alternatives are discussed in the next chapter together with the one chosen for the implementation.

We can see that there are two categories of events: communication and other events. The first are used to request specific services from the CCM:

- initialization/termination
- file sending/receiving.

Other events are encapsulated and transmitted. The encapsulation process is transparent, thereby making the communication invisible to the caller and recipient.

3.6.3. Testing the CCM

Before the CCM is employed in CALView it needs to be tested to find out if it meets the specifications. The following figures have to be determined (depending on the number of connected stations):

- delays during set-up and normal transmissions
- total number of events that can be transmitted per second
- delays during transmission of files.

These data can help in determining the expected lag. There are two situations these measurements have to be determined: in a LAN and in a WAN. The first situation is called phase 1 and the second phase 2. An average size for the events has to be defined. Another possibility would be to create a standard interaction script that can be played over and over again in order to take these measurements. Note that the numbers defined in section 3.2 can be used to create a such an interaction script.

Testing phase 1

Figure 3.13 shows an overview of the situation for a LAN. The machine attached to the net
The following measurements have to be taken for 2, 4 and 6 machines:

1. set-up delay
2. average transmission delay per event
3. speed of file transmission
4. number of events that can be transmitted per second (measured by flooding the network with messages and measuring the delay for the reception of each of these messages)
5. behaviour of the system when stations are dropping out.

Test phase 2
To find out what happens using the CCM on a WAN the same measurements used in test phase 1 can be taken. However, the network situation is considerably more complex, see figure 3.14 below. It can be seen that the private Sun network capabilities over a satellite link might be used.

When the measurements have been taken in this situation, it is possible to find out if there are elements of the CCM that need to be optimised. The CCM can then be employed within CALView.
Figure 3.14. Overview of test phase 2
4. CALView Communication

This chapter discusses the last part of my research: the implementation of a communication layer in CALView. The specification is used as a guideline to help create this communication layer. It also became apparent that local communication within a single CALView instance is important. The possible tools for the internal communication are discussed first. In a later section this is extended to communication with other CALView instances. These tools are obtainable for current and future computing platforms available at the Calibre Institute. The last section presents the communication software for the prototype of CALView.

4.1. Internal communication

Depending on the usage of the programming language (C, C++ or Objective-C), several tools have come forward as interesting to use for the implementation of the communication within CalView:

1. ToolTalk for usage with C and C++
2. the GNU\(^\text{1}\) Objective-C library with distributed objects support.

The possible advantages and disadvantages of these two alternatives are considered in this section.

4.1.1. ToolTalk

The ToolTalk service is a concept of inter-operability: it enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipients, and then delivers the messages to the appropriate applications (figure 4.1). It is not only possible to send messages to different processes on the same machine but also to processes running on other machines, see [SunSoft92a], [SunSoft92b], [SunSoft92c].

Developers of inter-operating applications agree upon a set of messages and then use the ToolTalk service to deliver the messages. The set of messages exchanged by a group of inter-operating applications is called a protocol: a protocol forms a small well-defined interface that maximises application autonomy.

The message protocol specification includes the set of messages and how applications should behave when they receive the messages. If protocols are observed, cooperating applications can be modified, or even replaced, without affecting one another.

The ToolTalk service supports several message styles. A sender can address a ToolTalk message to a particular process, to any interested process, to an object, or to an object type. ToolTalk is a product of SunSoft and it is included on Sun, HP, Silicon Graphics, IBM and DEC workstations. An example of its usage is in the OpenWindows environments were it

---

1. GNU is Not UNIX

25
manages communication between different desktop programs: e.g. double-clicking a file in an e-mail message attachment opens the appropriate application for that file type as registered by the ToolTalk service.

At the moment ToolTalk is based on RPC (Remote Procedure Call [Cypser91]), a relatively reliable service, but when a failure does occur, the sender is unable to distinguish among many possible outcomes [Birman93]. However, in the case of local communication on a LAN the chances of the occurrence of errors are relatively small.

Message sequence is preserved between a given sender and receiver, if process $A$ sends message $m_1$ and later $m_2$ and process $B$ receives both $m_1$ and $m_2$, process $B$ will get $m_1$ before $m_2$.

There are two exceptions, both fairly specialised, see [Lamer93].

Two other points of concern are the following:

1. The transfer of massive amounts of data might better be handled by another means such as sockets and shared memory.

2. If maximum speed is necessary and flexibility is not mandatory (an example of another means could be RPC).

It is not yet clear if the current version of ToolTalk is adequate for usage in CalView on a LAN basis. Questions which have arisen during the research are the following:

1. How many messages per second can be handled by the ToolTalk service?

   With about 100 small messages per second for Solaris 2.0, performance mainly depends on how many recipients each message has (according to [Larner93]). The less matches for potential recipients are found the faster the service.

   Sending directly to a process is faster than procedural messages that only match one receiver, object messages are the slowest.

2. What is the correlation between the answer on 1. and the size of messages?

   Again according to [Larner93] there is no limit on the size of the messages or the number of arguments. However, since the service copies the data several times while processing the message this could seriously impact performance especially when

Figure 4.1. Applications using the ToolTalk service

manages communication between different desktop programs: e.g. double-clicking a file in an e-mail message attachment opens the appropriate application for that file type as registered by the ToolTalk service.

At the moment ToolTalk is based on RPC (Remote Procedure Call [Cypser91]), a relatively reliable service, but when a failure does occur, the sender is unable to distinguish among many possible outcomes [Birman93]. However, in the case of local communication on a LAN the chances of the occurrence of errors are relatively small.

Message sequence is preserved between a given sender and receiver, if process $A$ sends message $m_1$ and later $m_2$ and process $B$ receives both $m_1$ and $m_2$, process $B$ will get $m_1$ before $m_2$.

There are two exceptions, both fairly specialised, see [Lamer93].

Two other points of concern are the following:

1. The transfer of massive amounts of data might better be handled by another means such as sockets and shared memory.

2. If maximum speed is necessary and flexibility is not mandatory (an example of another means could be RPC).

It is not yet clear if the current version of ToolTalk is adequate for usage in CalView on a LAN basis. Questions which have arisen during the research are the following:

1. How many messages per second can be handled by the ToolTalk service?

   With about 100 small messages per second for Solaris 2.0, performance mainly depends on how many recipients each message has (according to [Larner93]). The less matches for potential recipients are found the faster the service.

   Sending directly to a process is faster than procedural messages that only match one receiver, object messages are the slowest.

2. What is the correlation between the answer on 1. and the size of messages?

   Again according to [Larner93] there is no limit on the size of the messages or the number of arguments. However, since the service copies the data several times while processing the message this could seriously impact performance especially when
large amounts of data are involved together with many recipients. But since the increased load of large messages is just the time spent to copy that data, there is no edge to the drop-off: the time required to send drops off linearly with the size of the message. This implies that if large amounts of data have to be transferred regularly some way to bypass ToolTalk has to be used (e.g. a socket address could be delivered by a ToolTalk message after which this address is used by the recipient to transfer the data).

3. What is the complexity involved by using ToolTalk instead of for instance sockets or shared memory?
   One of the reasons Sun promotes the usage of ToolTalk is that it is rather simple to integrate networking inter-operability in applications compared to older solutions (RPC, sockets, shared memory). The demonstrations found during the research seem to confirm this [SunSoft92d].

   ToolTalk doesn’t use broadcasting or multicast so if hundreds of processes register interest in the same message ToolTalk will have to send the message hundreds of times. ToolTalk is designed on the premise that each message is only of interest to relatively few processes. There is no load-balancing distribution mechanism, if two processes have registered identical patterns ToolTalk will choose one of them and deliver all messages to it.

   With the goals that have been set for CalView ToolTalk could be used on a LAN for communication between several processes within one instance of CalView. It cannot be used for WAN communication: failures are not handled gracefully and the transfer of large amounts of data might require all kinds of bypasses which increase the complexity and thus diminish the advantage of the simplicity of ToolTalk.

4.1.2. Distributed objects

When Objective-C is used for the implementation, messaging is part of the run-time library which is required with the language [NeXT93], [Lozinski91]. Like most other programming languages, Objective-C was initially designed for programs that are executed as a single process in a single address space. However, the object-oriented model is ideally-suited for interprocess communication as well. Remote messaging in Objective-C requires a run-time system that can establish connections between objects in different address spaces, recognise when a message is intended for a remote address, and transfer data from one address space to another. It must also mediate between the separate schedules of the two tasks. At the time of writing, the GNU Objective-C library supports distributed objects and although the software is in its first release it looks like a promising alternative. Since source-code is available and free, it would be possible to enhance the library although this might be outside the scope of the institute’s research programs. This approach would have the benefit that internal and external communication would be indistinguishable to the programmer.
4.2. External communication

This section presents some of the available packages which could be used for the communication between several instances of CALView. Each tool is briefly described and the availability and the suitability of the software for the project is discussed.

4.2.1. ISIS

The ISIS software is developed at Cornell University and is now being maintained and enhanced by ISIS Distributed Systems, Inc. It runs on all major UNIX flavours.

ISIS is a system that provides tools to support the construction of reliable distributed software. The thesis underlying ISIS is that the development of reliable distributed software can be simplified using process groups and group programming tools [Birman93].

There are four styles of process group usage with ISIS, see figure 4.2 on page 29. They can be classified as follows:

1. Anonymous groups: An application publishes data under some topic and other processes subscribe to that topic. Properties that are needed for an application to operate reliably in this situation are:
   - Messages should be sent to a group address
   - Messages should be delivered exactly once, if the sender fails, a message should be delivered to all or none of the subscribers
   - Messages should be delivered in some sensible order, e.g. an order consistent with causal dependencies, but stronger ordering properties could also be desired.
   - It should be possible to obtain a history of the group, a list of key events and the order in which they were received.

2. Explicit groups: a group is explicit when its members cooperate directly, each member knows itself to be member of the group and employs algorithms that incorporate the list of members, relative ranking within that list, or in which responsibility for responding to requests is shared. Compared to anonymous groups, explicit groups have additional needs like for instance the notification of group membership changes.

All these properties are not independent and their integration within a single framework is not trivial. The solution proposed by ISIS is based on the virtual synchrony execution model. The communication layers of ISIS are discussed first. Contemporary operating systems offer three classes of communication services:

- Unreliable datagrams (e.g. UDP): corrupted messages are automatically discarded but do little additional processing. Most messages get through but under some conditions messages might be lost in transmission, duplicated or delivered out of order.
- Remote Procedure Call (RPC): communication results from a procedure that returns
a result. It is a relatively reliable service, but when a failure does occur, the sender is unable to distinguish among many possible outcomes: the destination may have failed before or after the receiving request, or the network may have prevented or delayed delivery of the request or reply.

- Reliable data streams: communication is performed over channels that provide flow control and reliable, sequenced message delivery (e.g. TCP, ISO protocols [Tanenbaum89]). Because of pipelining, these streams outperform RPC when an application sends large amounts of data. However, there are conditions under which a stream will be broken (excessive retransmission, timeouts).

Keeping the problems of these classes in mind, ISIS makes the following assumptions about the network and message communication: the network is structured as a WAN composed of LANs interconnected by long-haul communication networks. Each LAN is assumed to have a collection of machines connected by high-speed, low-latency communication devices such as ethernet. If shared memory is employed it is assumed that it is not used over the network and clocks are not assumed to be closely synchronised. To overcome errors which can occur in all these networks, ISIS uses a windowed acknowledgement protocol integrated with a failure-detection subsystem. By using this non-standard approach, a consistent system-wide view of the state of components in the system and of the state of communication channels can be presented to higher layers of software. For instance, the transport layer will only break a communication channel to a process in situations in which it would also report to any application monitoring that process that the process has failed. Moreover, if one channel to a proc-
ess is broken, all channels are broken.

A thorough discussion of virtual synchrony can be found in [Birman93], here is a summary of benefits of this model:

- It allows code to be developed assuming a simplified, closely synchronous execution model.
- It supports a meaningful notion of group state and state transfer, both when groups manage replicated data, and when computation is dynamically partitioned among members of the group.
- Asynchronous, pipelined communication.
- Treatment of communication, process group membership changes and failures through a single, event-oriented execution model.
- Failure handling through a consistently presented system membership list integrated with the communication subsystem. This is in contrast to the usual approach of sensing failures through timeouts and broken channels, which does not guarantee consistency.

The limitations of this approach are:

- Reduced availability during LAN partition failures: it only allows progress in a single partition and hence tolerates at most \( \lfloor n/2 \rfloor - 1 \) simultaneous failures, where \( n \) is the number of sites currently operational.
- Risks incorrectly classifying an operational site or process as faulty.

Note that theoretical arguments exist that no system that provides consistent distributed behaviour can completely evade these limitations.

It has been shown with DIVE (see section 2.4) that ISIS is a feasible approach for implementing distributed virtual reality. At the moment there is an educational version available\(^1\).

### 4.2.2. Horus

In the previous sub-section group communication and group membership protocols which form a very important concept of ISIS were introduced. [Renesse93] states that these protocols are poorly integrated and complex to use in modern operating system structures. Horus tries to use microkernel design techniques to tackle this problem. It has proven to be lightweight and fast and is easy to embed in operating systems like Chorus and Mach. At the same time it is very flexible to use for real-time communication, to control high speed communication devices and for high availability security technologies.

In comparison with ISIS, which tries to support all kinds of features and thereby becomes very complex for casual developers, Horus tries to become a framework in which modules can be placed once they are needed. With this approach, excessive overhead from unnecessary features is avoided. One can compare this to microkernels of modern operating systems:

---

1. Version 2.1 is from 1991 which has no support and a commercial version 3.1 is available with an academic discount.
there is a core collection of mechanisms which forms a skeleton for a generic group communication system.

Horus is built on several layers, see figure 4.3 on page 31:

- the Multicast Transport Service (MUTS)
- the Virtual Synchrony layer
- user-level services.

![Horus overview](image)

**Figure 4.3. Horus overview**

MUTS provides an asynchronous, reliable, one-to-many message-passing model with preemptive threads and support for timers and multiple address spaces. It can support multiple transport protocols, reliable or unreliable, point-to-point or multicast. It is designed to run either as part of the kernel or as a user process and is highly portable. Other parts of the Horus design make as much use as possible of this layer to provide complete portability.

The Virtual Synchrony technique has already been mentioned in the previous sub-section. In Horus it actually consists of several layers which present the same interface but perform different features.

The user-level services provide the modules that make Horus versatile, the services make use of the mechanisms provided by the other layers. Some examples are: failure analysis, message logging, a time service or an authentication service. All these services are fault-tolerant.

It seems that Horus is sufficiently powerful so that it can support the full range of functionality of the ISIS system. When looking at the performance results presented in [Renesse93] it looks as if Horus is suited for use in a distributed virtual reality system. It not only supports the aspects which made ISIS attractive but on top of that it is a set of lightweight processes (thus having a small impact on the overall machine performance) and it has a small amount of lag which would help in the interactive virtual reality environment.

Horus is available for universities at no cost and is available for Solaris, however at the time of writing there was no port available for IRIX\(^1\).

---

1. IRIX is the UNIX operating system of Silicon Graphics, Inc.
4.2.3. DOME

The Distributed Object Management Environment is an object management environment that organizes network objects into a hierarchy called Directory Services. It runs on most UNIX platforms (although not IRIX), VMS and MS-Windows. Because it sits in the middle between programs to enable them to communicate, it is also known as middleware.

In the DOME environment, any process can send a message to any other process. The software supports client-server, peer-to-peer, and broadcast messaging between any C-compatible applications, anywhere in the world. Messages are guaranteed to be delivered even in the presence of arbitrary failures. Thus a DOME application consists of multiple applications sending each other messages, just as a Smalltalk or Objective-C application consists of multiple objects sending each other messages. This makes it ideal for a distributed virtual reality application like CALView especially when it will be implemented using Objective-C.

Messaging is a very dynamic and powerful paradigm for building not only application programs but also distributed systems. With message-based systems, the location of the receiver can be changed at will. Messages can be broadcast to multiple receivers. They can be stored and forwarded when the network connection is re-established. They can be traced, or logged for future reference. Compared to RPCs, messaging middleware is much more flexible.

There are two types of messages: synchronous and asynchronous processing. Naturally, synchronous processing is more suited for CALView.

The DOME messaging services are based on queues. Messages are stored, forwarded, and held for the receiving application. The queuing approach leads to powerful properties. Hardware failures pose no problem: the messages just queue up until they can go through. Priorities are arranged by sorting the queue. Since the message servers run on every node in the network, any computer can go down, but the other computers can keep sending messages.

DOME supports registration of data messages, so that C data types are automatically translated between different operating systems. Other external communications can be merged in as well, including EDI, RPCs and X.400 messages.

When a program needs to send a message, Directory Services make it easy to find the receiving process. Directory Services manage the name and location of network resources so that even when a network resource moves, the name stays the same, and applications do not have to be modified. Directory Services can take a hierarchical name, much like a path name, and return the physical address of the process. Of course on a single computer this would be easy to do, but once there are thousands of computers the problem gets more difficult. First of all, the data has to be duplicated across the network, so that reliability and performance are achieved. Secondly, not every computer should store every address, or else the system would overload. Finally, whenever something changes, all copies of the information have to be updated. To solve these problems, DOME assigns each node in the Directory Services tree to multiple computers, and then manages that data correctly. At run-time the Directory Services
tree can be dynamically reconfigured to optimise network performance.

Locking, Notification, and Time Services are also supported by DOME. Locking Services allow locks to be placed on any network resource. Notification Services allow applications to request notification whenever any network object is changed. Time Services synchronise the time and date across multiple computers and time-zones.

It could be argued that because of the distributed character of DOME compared to the server approach of ISIS the first would have no potential bottleneck [Lozinski94]. At the moment this is no big issue since the number of participants is restricted to 6 but it could become important at a later stage.

This program does not have an academic discount although it can be arranged that the price will be the price of ISIS.

4.3. Implementation of the CCM

In the previous section several solutions have been presented that are available for external communication in CALView, in this section the implementation of the solution that has been chosen is discussed. Because of time constraints a prototype of CALView was adapted to include external communication. A simple albeit temporary solution was chosen, it is based on the NVR system in section 2.6: the TCP socket approach. The concepts of programming TCP sockets are explained in [Comer88].

4.3.1. Functional description

A peer-to-peer connection between two participants is used, because of the abstraction between the communication interface and the rest of the system the software can easily be replaced by one of the more advanced methods. In figure 4.4 a graphical representation of what has been created can be found.

![Figure 4.4. CCM functional overview](image)

The CCM software is a front-end for the TCP sockets which are used in the implementation. The Error Manager reports possible errors and terminates the program when a severe error has been encountered. The testcal software has been written to test the CCM without CALView and was later adapted to form a kind of remote control that can play or record
scripts or allow the user to type events which can be sent to CALView. A script is a collection of events generated in CALView which can be played over and over again, for example it could be possible to show demos remotely.

The CCM uses a simple protocol to format the events, see figure 4.5. Each event that is transmitted has a header containing the length of the event. This header is needed because of the stream concept. It is not necessary to create a more complicated protocol since the TCP protocol takes care of delivering the messages in the proper order and without errors.

Two different types of sockets are used:

- event socket: tuned for fast transmission (large buffers, no queuing delay)
- file: tuned for optimum transmission of large amounts of data.

![Figure 4.5. Event transmission format](image)

This distinction between events and bulk transfers can also be used in other implementations. Most communication protocols have special facilities for bursty traffic and bulk transfers.

### 4.3.2. Application programmers interface

The interface a programmer has to use to be able to use the Error Manager and the CCM in CALView is defined in this sub-section. The C language has been used to implement the software [Kernighan88], the important header files are to be found in appendix B see page 49.

**Error Manager:**

All errors are of type Error. The Error Manager defines two macros:

- **PrintError**(`E`, `S`)
  
  This macro prints a message on the error stream `stderr` with the string `S` for error `E`. Depending on the error number it will call the appropriate UNIX error function or print only the message string `S`.

- **FatalError**(`E`, `S`)
  
  This macro uses `PrintError(E, S)` to print the error message but will also terminate the process. When the `DEBUG` macro is defined, the process will dump core.

**CCM:**

We use two sockets: one which is tuned for short messages like events and one for the transmission of large amounts of data such as files. See [Sechrest86] and [Leffler86] for a description of BSD inter-process communication on which the CCM is based.

```c
Error_InitConnections(const char *hostName, int *eventPort,
                     int *filePort)
```
Initializes the connections to the given machine. Two file descriptors are returned which can
be used using for instance select() to check for incoming messages. Apart from a time­
out error CONN_TIME_OUT_WAITING no other errors are returned in the current imple­
mentation, they are handled by the Error Manager instead.

Error CloseConnections()
Closes all open connections.

Error SendEvent(void *data, int length)
Sends an event to all connections. Data should be a machine-independent description of the event (encapsulated by using for instance xdr() or something based on the type codes used by the +descriptionForMethod method of the Object class in NEXTSTEP [NeXT93]). The last argument is the length of the data.

void *ReceiveEvent(void *data, int *length)
Receives an event from a connection. This function should be called once activity on the
event socket has been noticed. It returns the received event. In cases of error a NULL pointer
is returned and the global variable errno indicates what went wrong. There is one excep­
tion: if a CONN_BUFFER_TOO_SMALL error occurs when using static buffers, the buffer
will contain the maximum amount of data it could contain and all other event data is dis­
carded. If the buffer is allocated dynamically, enter the length of the buffer. In all other cases
enter a NULL pointer and a zero length and this routine will allocate the data. It is the respon­
sibility of the programmer to free the data eventually.

Error SendFile(int fileDesc)
Sends the information given by the file descriptor to all connections. Note that the program­
mer has to open the file first and send an event to inform the peer that a file will be sent. When this call has finished the file can be closed.

Error ReceiveFile(int fileDesc)
Receives the information given by the file descriptor from a connection. The file descriptor
must be an open file (an event informing the program of the imminent arrival must be
received first). Close the file once this function has finished.
5. Future Developments

Although a current prototype of CalView can be regarded as a distributed virtual reality program, a final implementation will have to address specific issues. Some of these issues have been discussed in the previous two chapters but there are also topics such as user-interfaces, behaviour of the CALView program when more than two participants are connected, etc. This chapter tries to address some of these issues in order to help the future development of the virtual reality research at the institute.

5.1. Behaviour with multiple participants

Here are two ways in which the behaviour of CALView can be extended with more participants:

1. Replace the communication layer but leave the functionality in tact. This is of course the simplest possibility which requires no adaptation of all other aspects of CALView since the speaker can only address more participants. Note that the behaviour of the participants does not change.

2. Change the behaviour of the participants from inactive viewers to users who can interact with the virtual world. This will have a major impact on the whole structure of CALView and this requires a complete rewrite of the current prototype. For instance, a linear event structure is no longer sufficient in the Event Manager. Depending on the maximum number of participants and the way they can interact, an appropriate solution has to be chosen.

The ideas from WAVES (see section 2.5) seem very promising for large virtual worlds dispersed over a wide geographic area. Although the simulations intended for CALView are not that large, the ideas concerning user interaction in WAVES are of importance: synchronisation of the interaction between objects becomes a necessity. For instance it could be possible that user A picks up an object which was already picked by user B but because of transmission delays the host machine for user A was not yet notified. What to do? This situation should never occur, several papers on this subject ([Codella92], [Kazman93b], [Locke93], [Shaw93]) propose locking mechanisms but they always seem to have a large impact on the performance experienced by the users. WAVES proposes behavioural models where they may be used to predict the behaviour in the near future. The previous example could possibly be solved on host A by predicting the course of the hand of user B. It could place an initial lock on the object and user A cannot pick it up immediately.

The previous example directly raises another point of concern: user A could also have predicted the behaviour of user B when the representation of the last user shows her intent. Thus the imaging of users should reflect their (intended) actions and direction of movement.
5.2. The user interface

In the previous section the subject of the user interface has already been mentioned. DIVE is an excellent example for learning how to use a 3D user interface. Diverse applications have been created for that environment each using a different approach for a 3D user interface depending on the kind of application [Carlsson93a]. Other interesting directions for research in this area can be found in [UKWorkshop93]. The concerns discussed above make it clear that a lot of research has yet to be done in this direction.

5.3. Implementation issues

At the time of writing the prototype for CALView has been developed for the Sun Solaris operating system using XGL for rendering and the OpenWindows environment for the user interface. It is written in the C language.

If a major rewrite of CALView is to be undertaken it would be sensible to use an object-oriented approach. Although C++ may be the most popular language it would not be the best choice. Objective-C would be a much more flexible approach. The distinction between internal and external process communication would disappear. Furthermore load-balancing is also possible because there is no distinction between messages on a local or a remote host. Another advantage would be to add functionality at run-time. Other reasons which speak for Objective-C are argued in [Lozinski91]. The fact that Sun is going to support OpenStep (which is based on Objective-C) in the near future is yet another advantage.

A concept with the necessary classes for the Event Manager has already been created, see figure 5.1. An event parser is also developed, it is based on obstacks which are present in the GNU C library (see section A.2). These obstacks make it possible to parse unknown amounts of data without continuously allocating and releasing memory, which is necessary for a fast parser.

![Figure 5.1. Object tree for Event Manager](image-url)
6. Conclusions

The current prototype of CALView can be regarded as a distributed virtual reality application. It is adequate for the initial testing purposes between the Calibre Institute and the Kantoor van de Toekomst.

It has become clear that in most cases bandwidth is not a necessary commodity but a low latency is. Although TCP might be regarded as a bad choice when latency is of concern, at this moment a latency of some seconds is not relevant because of the master/slave configuration. It more important that all slaves are receiving the information without errors.

When a smaller amount of latency becomes important, then the interesting aspects of new network technologies like ATM and FDDI are necessary for responsive distributed virtual reality applications. Obviously, when real-time audio and video data are also part of the model simulated in CALView, larger bandwidths are required. This is also the case if the behaviour of the participants is changed from inactive viewers to participants who can interact with each other using the properties in the model.

A rewrite of CALView is inevitable for expanding its capabilities. The guidelines which have been set out in this report might help once a new version is going to be created using an object-oriented approach.
7. Bibliography


[Leffler86] Leffler, S.J. et al.; *An Advanced 4.3BSD Interprocess Communication Tutorial*. UNIX Programmer's Supplementary Documents, Vol. 1; Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, 1986; Pp. PS1:8

[Locke93] Locke, J.; *An Introduction to the Internet Networking Environment and SIMNET/DIS*. Computer Science Department, Naval Postgraduate School, USA; February 1993; email jxxl@taurus.cs.nps.navy.mil

[Lozinski94] Lozinski, C.; email conversation August-October 1994

[NeXT93] NeXT Publications; *NeXTSTEP Object-Oriented Programming and the Objective-C Language*. NeXTSTEP Developer’s Library, Release 3; Reading, MA: Addison-Wesley, April 1993


[Sechrest86] Sechrest, S.; *An Introductory 4.3BSD Interprocess Communication Tutorial*. UNIX Programmer’s Supplementary Documents, Vol. 1; Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, 1986; Pp. PS1:7


[Smeltzer93b] Smeltzer, G.T.A.; *Virtual Reality en bouwkundig ontwerpen in een telematica-omgeving: fantasie of werkelijkheid?*. internal report of the Calibre Institute (Eindhoven University of Technology, faculty of Architecture, Building and Planning), September 1993 (in Dutch)


[UKWorkshop93] diverse authors; *Report on the One Day Immersive Environments Workshop*. Department of Computer Science and London Parallel Applications Centre, Queen Mary and Westfield College, London. July 1993. available by email from A. Steed <steed@dcs.qmw.ac.uk>
A. Event Description

A description of an event consists of a string of characters. In this string there are groups of successive characters which represent a single object, a token. These tokens need to be parsed in order to check the overall structure of this event. We will first discuss the lexical scanner and after that the parser necessary to work with our event structure.

A.1. Lexical scanner

The task of the lexical scanner is to partition the input into tokens. In order to do this we need to describe these tokens using regular expressions.

Definitions:

- \( \varepsilon \) denotes the empty sequence
- \( e_i \) denotes the regular expression belong to symbol \( i \)
- \( L(r) \) denotes the language belonging to the regular expression \( r \)

\[
\begin{align*}
\text{letter} & = 'a' .. 'z' | 'A' .. 'Z' \\
\text{digit} & = '0' .. '9' \\
\text{digseq} & = \text{digit}^+ \\
\text{printchar} & = ' ' .. '~' \\
\text{sep} & = '\#' | '\n' | '\r' | ','
\end{align*}
\]

Tokens \( = \{ \text{EventSym, VersionSym, IDSym, SourceSym, ClassSym, TypeSym, LBrackSym, RBrackSym, NegSym, EOFSym, DataSym, NumSym, StringSym} \} \)

ReservedWords \( = \{ \text{"event", "id", "source", "class", "type"} \} \)

token: \hspace{1cm} \text{regular expression:}

<table>
<thead>
<tr>
<th>Token</th>
<th>Regular Expression</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventSym</td>
<td>“event”</td>
<td></td>
</tr>
<tr>
<td>VersionSym</td>
<td>“version”</td>
<td></td>
</tr>
<tr>
<td>IDSym</td>
<td>“id”</td>
<td></td>
</tr>
<tr>
<td>SourceSym</td>
<td>“source”</td>
<td></td>
</tr>
<tr>
<td>ClassSym</td>
<td>“class”</td>
<td></td>
</tr>
<tr>
<td>TypeSym</td>
<td>“type”</td>
<td></td>
</tr>
<tr>
<td>LBrackSym</td>
<td>‘(’</td>
<td></td>
</tr>
<tr>
<td>RBrackSym</td>
<td>‘)’</td>
<td></td>
</tr>
<tr>
<td>NegSym</td>
<td>‘¬’</td>
<td></td>
</tr>
<tr>
<td>EOFSym</td>
<td>‘\0’</td>
<td></td>
</tr>
<tr>
<td>DataSym</td>
<td>( e_{\text{data}} ) ) such that ( L(e_{\text{data}}) = L(\text{letter} . (\text{letter}</td>
<td>\text{digit})^*) ) Reserved-Words</td>
</tr>
<tr>
<td>StringSym</td>
<td>‘”’. printchar . ‘”’</td>
<td></td>
</tr>
</tbody>
</table>
NumSym \hspace{1cm} \text{digseq}.(\varepsilon | \cdot \cdot \cdot \text{digseq}).(\varepsilon | (\text{E} \cdot \text{e}^+ | \text{e}^- | \varepsilon \cdot \text{digseq})\],\\

\text{f: ReservedWords} \rightarrow \text{Tokens} : "f(x) is the token corresponding to the string x as shown in the table above"}

\section*{A.2. Header files}
The following two header files are necessary to use the scanner. Note that the software requires the GNU C library to make it work.

\subsection*{A.2.1. ScanEvent.h}

\begin{verbatim}
/* ScanEvent.h

Written by Annard Brouwer <annard@stack.urc.tue.nl>

Copyright 1994, Calibre Institute

*/

#ifndef _EVENTMANAGER_SCANEVENT_H
#define _EVENTMANAGER_SCANEVENT_H

#include "String.h"

#define TOKENLENGTH 32 /* length in chars of a token */
#define EOE '\0'    /* end of event char */

/* All the tokens we can scan, note that DataSym must follow the symbol for the last reserved word, which is TypeSim */
enum Tokens 
{EventSym, VersionSym, IDSym, SourceSym, ClassSym, TypeSym, 
DataSym, LBrackSym, RBrackSym, EOE, NumSym, StringSym, NegSym};

typedef int Token;

/* Initialize the Event Scanner, this must be done for each event which is to be scanned.

IN
    input: string representing the event
    len : the length of input

RETURNS
    NOTENUMMEM if space can't be allocated
*/
Error StartScan(char *inputString, int inputLen);

/* Retrieve a symbol from an event.

IN
    none

RETURNS
    sym : the recognised symbol
    data : pointer to data belonging to symbol (this data must be copied for permanent usage, after FinishScan it will be invalid!)
*/
Error NextSym(Token *sym, char **data);

void FinishScan();
\end{verbatim}
#ifndef _EVENTMANAGER_STRING_H
#define _EVENTMANAGER_STRING_H

typedef struct _string
{
    size_t stringSize; /* length of string in characters */
    size_t stringAllocated; /* allocated size for string */
    char *stringData; /* string itself */
} String;

/* Allocate a string. */

IN
    str : pointer to array of char
    grow : boolean variable indicates if the string can grow at a later stage
RETURNS
    a String which is empty if it couldn't be allocated
*/
String *AllocString(char *str, int grow);

/*
Free a string.
*/

IN
    str : a previously allocated String
RETURNS
    nothing
*/
void FreeString(String *str);

/*
Clear the contents of the string.
*/

IN
    str : String
RETURNS
    a String with empty data but the memory is still allocated
*/
void ClearString(String *str);

/*
Add two strings together.
*/

IN
    str1, str2 : two Strings
RETURNS
    str1 which is concatenated with str2
String *AddString(String *str1, String *str2);

/*
 * Add a character to the end of a String

IN
 c : a character
 str : a String

RETURNS
 str with c appended
 */
String *AddCharToString(int c, String *str);

/*
 * Get the encapsulated string data

IN
 str : a String

RETURNS
 const char *
 */
const char *GetStringData(String *str);

#endif /* _EVENTMANAGER_STRING_H */
B. CCM Header Files

This appendix contains the three most important header files which are necessary to use the CCM in other projects.

B.1. config.h

/* config.h

This is the system dependency file for the CCM project.

Written by Annard Brouwer <annard@stack.urc.tue.nl>

Copyright 1994, Calibre Institute

$Modified: Tue Nov 29 00:30:32 1994 by annard $

*/

#ifndef _CCM_CONFIG_H
#define _CCM_CONFIG_H

/* Define your system here, currently supported are NeXT,
   GNU_libc, SGI, SunOS4, Solaris. */
#define GNU_Libc
#define SGI
#define Solaris
#define SunOS4

/* When you use CCM with X, set the following define. 
At this moment only XView is supported. */
#define XWINDOWS

#if defined(NeXT)
#include <c.h>
#endif /*include <libc.h> not needed for POSIX */
#include <limits.h>
#include <unistd.h>
#include <stdarg.h>
#include <fcntl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <bsd/netdb.h>
#define MAXHOSTNAMELEN 65 /* needed for POSIX */

#elif defined(GNU_Libc)
#include <c.h>
#include <stdio.h>
#include <stdarg.h>
#include <unistd.h>
#include <fcntl.h>
#include <limits.h>
#include <limits.h>
#include <stdio.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <stdio.h>
#include <limits.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MAXHOSTNAMELEN 64

#define XVIEW
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/notify.h>

#ifndef _XWINDOWS
#define_XWINDOWS
#endif /*_XWINDOWS*/

#define */ XWINDOWS */

#endif /* system dependent defines */
/** Error.h

This is the error manager file for the CCM project.

Written by Annard Brouwer <annard@stack.urc.tue.nl>

Copyright 1994, Calibre Institute

$Modified: Fri Nov 25 11:29:22 1994 by annard $ */

#ifndef __CCM_ERROR_H
#define __CCM_ERROR_H

#include "config.h"
#include <errno.h>
#include <string.h>
#define NO_ERROR

/* General Errors */
#define ERRBASE1000
#define NOTENUM.ERRBASE+1

/* Connection Errors */
#define CONN_ERRBASE 2000
#define CONN_TIME_OUT_WAITING CONN_ERRBASE+1
#define CONN_CREATE_SOCKET_ERRCONN_ERRBASE+2
#define CONN_BIND_SOCKET_ERRCONN_ERRBASE+3
#define CONN_HOST_LOOKUP_FAILCONN_ERRBASE+4
#define CONN_WRITE_ERRORCONN_ERRBASE+5
#define CONN_ILLEGAL_HOSTCONN_ERRBASE+6
#define CONN_PROTO_NOT_FOUNDCONN_ERRBASE+7
#define CONN_BUF.TOO_SMALLCONN_ERRBASE+8

/* machine and user retrieval errors */
#define MACH_ERRBASE2100
#define MACH_USER_NOT_FOUND MACH_ERRBASE+1
#define MAC_F.H_HOST NOT FOUND MACH_ERRBASE+2

#define CheckError(E) do if (E) return E; while(0)
#define PrintError(E, S) do 
    if (E > ERRBASE) fprintf(stderr, "CCM: %s\n", S);
    else fprintf(stderr, "CCM: %s %s\n", S, strerror(E));
    while (0)
#endif
#endif

ifdef DEBUG
#define FatalError(E, S) do ( PrintError(E, S); abort(); ) while (0)
#else
#define FatalError(E, S) do ( PrintError(E, S); exit(E); ) while (0)
#endif

typedef int Error;

#endif /* _CCM_ERROR_H */

B.3. ccm.h

/* ccm.h */
This is the main interface for the CCM project.

Written by Annard Brouwer <annard@stack.urc.tue.nl>

Copyright 1994, Calibre Institute

$Modified: Mon Nov 28 23:08:36 1994 by annard $

*/

#ifndef _CCM_H
#define _CCM_H

#include "config.h"
#include "Error.h"
#include "Malloc.h"

#define EVENTPORThtons(4240) /* default port number for event socket */
#define FILEPORThtons(4241) /* default port number for file socket */
#define TCPTIMEOUT60 /* time-out for connection in seconds */

struct _ConnectInfo
{
    int eventSocket; /* this socket will be used to for events */
    int fileSocket;  /* this socket will be used for files */
};

typedef struct _ConnectInfo ConnectInfo;

/*
 This function tries to establish a stream connection to the given host.
 first it tries to make a connection by trying to connect to a socket, then if 
 the host is not reachable it will wait till a connection request is received, 
 after a reasonable time-out period it returns an error.

 It returns 2 descriptors which can be used in select() to listen for activity. 
 It either returns NO_ERR or CONN_TIME_OUT_WAITING when a time-out occurs, 
 otherwise it halts by calling FatalError with the error number.
 */
Error InitConnections(const char *host, int *eventSocket, int *fileSocket);

/*
 This function closes all established connections.
 */
Error CloseConnections(void);

/*
 Sends an event to the connected host.
 */
Error SendEvent(void *data, int length);

/*
 Receives an event from a host, call this function if you have noticed activity 
 on the event socket. If you have malloc-ed the pointer enter the length as 
 of this pointer as well otherwise enter a null-pointer and a zero length.

 If the data pointer isn't big enough then length will indicate the amount 
 needed and a CONN_BUFF_TOO_SMALL error will be set in errno, and data will contain 
 as much data as it could contain.
 It is your responsibility to free the data eventually.
 If another error occurs then length will be zero indicating invalid data and the 
 function returns NULL, the error can be found in errno.
 */
void *ReceiveEvent(void *data, int *length);

/*
 * Opens the given file and sends first the name of the file and after it the data to the connected host.
 * It is your responsibility to open and close the file.
 */
Error SendFile(int fileDesc);

/*
 * Writes the file to be received to the given file descriptor. Call this function when you have received activity on the file socket.
 * It is your responsibility to open and close the file.
 */
Error ReceiveFile(int fileDesc);

#endif /* _CCM_H */