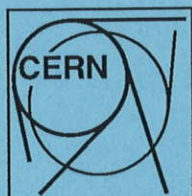


Høgskolen i Molde
Molde College
Molde, Norway



EDWIN

Event Display with World-Wide Web Interface

Thesis in Computer Science

by

Frode Tennebo

CERN

3rd January - 3rd July 1995

Thesis-1995-Tennebo

ABSTRACT

The World-Wide Web has created a new paradigm for accessing on-line information, but on-line interaction through this new medium has not yet been that well catered for.

The project described in this paper was conceived to investigate the concept of providing an interactive application on the World-Wide Web accessing the Online Event Display system at CERN.

Several different strategies and methods of implementation are discussed in this paper, and the prototype developed, which is described in detail here, does not require any configuration to be performed by the end-user, minimal configuration for the WWW-administrator at the site running the application and a simple interface for developing applications using the technology described.

CERN LIBRARIES, GENEVA

CERN LIBRARIES, GENEVA



CM-P00081089

CONTENTS

	ABSTRACT	III
	CONTENTS	IV
	LIST OF FIGURES.....	VII
1	INTRODUCTION	1
1.1	CERN	1
1.1.1	<i>ECP/PT/CSE</i>	2
1.2	LEP and DELPHI.....	2
1.3	The Project	4
1.4	Acknowledgements	5
1.5	Organization.....	5
2	ENVIRONMENT	7
2.1	Internet	7
2.1.1	<i>What it is</i>	7
2.1.2	<i>How it started</i>	7
2.1.3	<i>How it works</i>	8
2.1.4	<i>TCP/IP</i>	10
2.2	World-Wide Web	11
2.2.1	<i>What it is</i>	11
2.2.2	<i>How it started</i>	11
2.2.3	<i>How it works</i>	12
2.2.4	<i>HTTP</i>	13
2.2.5	<i>HTML</i>	14
2.2.6	<i>Forms</i>	15
2.2.7	<i>CGI</i>	16
2.2.7.1	<i>Perl</i>	17
2.3	Online Event Display	18
2.4	X.....	18
3	CONCEPTS	20
3.1	WWW Solution	21
3.1.1	<i>WWW-part</i>	21
3.1.2	<i>Gateway</i>	22
3.1.3	<i>OED-part</i>	25
3.2	General	27
3.3	History.....	27
3.3.1	<i>Timeline</i>	28
3.4	Existing examples	28

3.4.1	<i>Teletext</i>	29
3.4.2	<i>Eòlas</i>	29
4	WWW SOLUTION	31
4.1	XTCP	31
4.1.1	<i>Stage I</i>	31
4.1.2	<i>Stage II</i>	33
4.1.3	<i>Stage III</i>	35
4.1.4	<i>Stage IV</i>	37
4.1.5	<i>Implementation</i>	38
4.1.5.1	XTCPlisten()	38
4.1.5.2	XTCPschedule_dump().....	39
4.1.5.3	XTCPwrite().....	39
4.1.5.4	XTCPunlisten()	39
4.1.5.5	XTCPportno()	39
4.1.5.6	XTCPuserpar()	39
4.1.5.7	XTCPappcontext()	39
4.1.5.8	XTCPtimeout().....	39
4.1.5.9	connection_read().....	40
4.1.5.10	reschedule()	41
4.1.5.11	connection_close()	41
4.1.5.12	resize_window().....	41
4.1.5.13	resize_wait().....	42
4.1.5.14	shell_widget().....	42
4.1.5.15	dump_image()	42
4.1.5.16	get_scline().....	43
4.1.5.17	put_byte()	43
4.1.5.18	sync_display()	43
4.1.5.19	free_dump().....	43
4.1.5.20	tmpname()	43
4.1.5.21	strand()	44
4.1.6	<i>Improvements</i>	44
4.2	CGI Gateway.....	44
4.2.1	<i>Configuration file</i>	45
4.2.2	<i>Implementation</i>	46
4.2.2.1	dbLock	46
4.2.2.2	dbUpdate	46
4.2.2.3	dbUnlock.....	47
4.2.2.4	LogUpdate.....	47
4.2.2.5	dbCheck	47
4.2.2.6	ParseForm	48
4.2.2.7	TCPsend.....	52
4.2.2.8	Tempname.....	53
4.2.2.9	SendOutput.....	54
4.2.2.10	LocalDie.....	54
4.2.2.11	RemoteDie	54
4.2.3	<i>Improvements</i>	54
4.3	Installation.....	55

5	THE ALTERNATIVES.....	58
5.1	Alternative Solutions.....	58
5.1.1	<i>X Solution.....</i>	<i>58</i>
5.1.2	<i>Browser Solution.....</i>	<i>60</i>
5.2	Alternative Implementations.....	60
5.2.1	<i>C/C++.....</i>	<i>60</i>
5.2.2	<i>HotJava.....</i>	<i>61</i>
6	CONCLUSION.....	63
A	PROGRAM LISTINGS.....	64
A.1	xtcp.h.....	65
A.2	xtcp.c.....	67
A.3	xtcp_test.c.....	77
A.4	edwin.....	79
A.5	edwin.conf.....	83
B	ADDITIONAL PAPERS	84
B.1	XTCP - Communication Channel Interface for a Generic Server or X Client.....	85
B.2	WEBED - Preliminary report.....	95
C	GLOSSARY.....	101
D	BIBLIOGRAPHY	105

LIST OF FIGURES

Figure 1	The DELPHI detector.	3
Figure 2	The postal system.	9
Figure 3	The Internet system.	10
Figure 4	The WWW system.	12
Figure 5	Two views of the DELPHI Online Event Display.	20
Figure 6	Basic WWW Browser-Server Interaction.	21
Figure 7	WWW Browser-Gateway-Server Interaction.	22
Figure 8	Overview of EDWIN.	23
Figure 9	Initial form from the EDWIN prototype.	24
Figure 10	Prototype EDWIN form with rotation and zoom.	25
Figure 11	The EDWIN internals.	26
Figure 12	Screenshot of NOS' teletext gateway 'Teletekst'.	29
Figure 13	Screenshot of Eòlas' enhanced Mosaic-browser showing a molecule-demo.	30
Figure 14	STD of XTCP in stage I.	32
Figure 15	CSTD of stage I.	33
Figure 16	CSTD of stage II.	34
Figure 17	CSTD of stage III.	36
Figure 18	CSTD of stage IV.	37
Figure 19	STD of stage IV.	40
Figure 20	EDWIN gateway flowchart.	45
Figure 21	Flowchart of dbCheck	48
Figure 22	Flowchart for ParseForm.	49
Figure 23	Flowchart of CMD specific actions.	51
Figure 24	Flowchart of TCPsend.	52
Figure 25	Flowchart of the processing of input within TCPsend.	53
Figure 26	X solution.	58
Figure 27	Screendump of a DELPHI OED session.	59
Figure 28	Chemical models in HotJava.	62

All figures by the author except figure 1, courtesy of the DELPHI Collaboration, and figures 2, 3 and 4 courtesy of Robert Cailliau.

1 INTRODUCTION

1.1 CERN

It was at the European Cultural Conference in Lausanne, Switzerland, in December 1949 where the French scientist Louis de Broglie, the father of wave mechanics, first proposed that European countries join forces to create an European research laboratory to undertake research together on a scale that would be impossible for individual states to carry out.

The real birth of CERN¹ was marked by the signing of the Convention in Paris on 1st July 1953. It came into force in 1954. An underlying reason involved in setting up such a huge research facility in the centre of Europe, was to stop the 'export' of scientists to the United States of America, many of whom had been forced into exile due to the Second World War. Many young scientists were in turn tempted to cross the Atlantic in search of adequate research facilities which in its weakened state, Europe lacked.

A political as much as a scientific development, the CERN project was the first step on the road to reconciliation through co-operation in the neutral and fertile field of pure research. It also gave Europe the hope of being able to keep its scientists and regain its position at the forefront of scientific research. After 35 years of existence, CERN has largely achieved that objective, 6,000 particle physicists - half of the world total - from over 80 nations² are now working in CERN-related projects. For several years there have been more American physicists at CERN than European physicists on the other side of the Atlantic.

Located on Franco-Swiss territory, it houses some of the most advanced scientific instruments ever built: particle accelerators. The latest of these, Large Electron-Positron collider (LEP), the world's largest particle collider and one of the most ambitious scientific projects ever realized with a strong technological components from Europe's high technology industries, was made available for research in August 1989.

¹ The European Laboratory for Particle Physics

² CERN Member States as of January 1995: Austria, Belgium, Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Italy, Netherlands, Norway, Poland, Portugal, Slovak Republic, Spain, Sweden, Switzerland, United Kingdom.

1.1.1 ECP/PT/CSE

High energy physicists have an insatiable appetite for fast access to their experimental data and to any information which keeps them abreast of progress. They want to follow a tutorial in Hamburg, look up a colleague's phone number in Amsterdam, fetch a pre-print from California, or read the latest news on the top quark from Illinois: all without needing to be 'computer freaks'.

This is now possible, largely due to the invention of the World-Wide Web, or WWW. This resulted in the creation of the Communications Systems for Experiments (CSE) section within ECP/PT. The Programming Techniques Group (PT) in the Electronics and Computing for Physics Division (ECP) at CERN was formed in 1990 to help the teams that produce software for and within CERN experiments to improve software development productivity and quality.

The activities of the CSE section are based on the World-Wide Web technology. The section runs the www.cern.ch HTTP-server, which is the WWW-server for the Laboratory and for the Experiments, and provide assistance to large and small experiments and also to CERN Divisional Administrations. Part of this support consists on the developing of general tools to run in the WWW server and advise on software and hardware for use with WWW. It was within this section that I was given a six months contract as Technical Student.

1.2 LEP AND DELPHI

The operation of LEP started in August 1989 opening a new chapter in the history of CERN. With a circumference of 27 km, LEP is the largest accelerator yet built: it accelerates electrons and their anti-particles, positrons, in opposite directions in a vacuum pipe inside a retaining ring of magnets, before inducing them to collide head-on. In their annihilation the energy released in a small volume is comparable to that which existed in the Universe a fraction of a second after its creation in the Big Bang. There is thus a synthesis between the very large (astrophysics) and the very small (particle physics).

The LEP accelerator has its initial energy chosen so that in these collisions a Z^0 particle would be produced. This particle, discovered at CERN in 1983, is responsible for the weak nuclear force which drives our Sun. The theory of fundamental particles called the "standard model" is being critically tested by studying the creation and decay of the Z^0 . The Z^0 is very short-lived so its presence has to be inferred by its disintegration fragments which may vary from two to nearly a hundred. These are the familiar particles studied over the past 40 years, and perhaps others yet to be discovered. In the next phase the energy of LEP will possibly be nearly doubled to allow production of W^+W^- pairs, the charged counterparts of the Z^0 . This will open a new domain of investigations and "standard model" tests.

In LEP, the beams of electrons and positrons are made to collide many times during their circuit. LEP has four intersection regions, each of which is surrounded by a particle detector to measure the properties of the secondary particles from Z^0 decay. Each of the detectors has been tuned to increase its ability to study various physics aspects. DELPHI (DEtector with Lepton, Photon and Hadron Identification) is an advanced detector which, as well as having high precision and 'granularity', has the specific ability, using the Ring Imaging Cherenkov technique to differentiate between the various secondary particles. It also has an advanced silicon strip micro-vertex detector to detect very short lived particles.

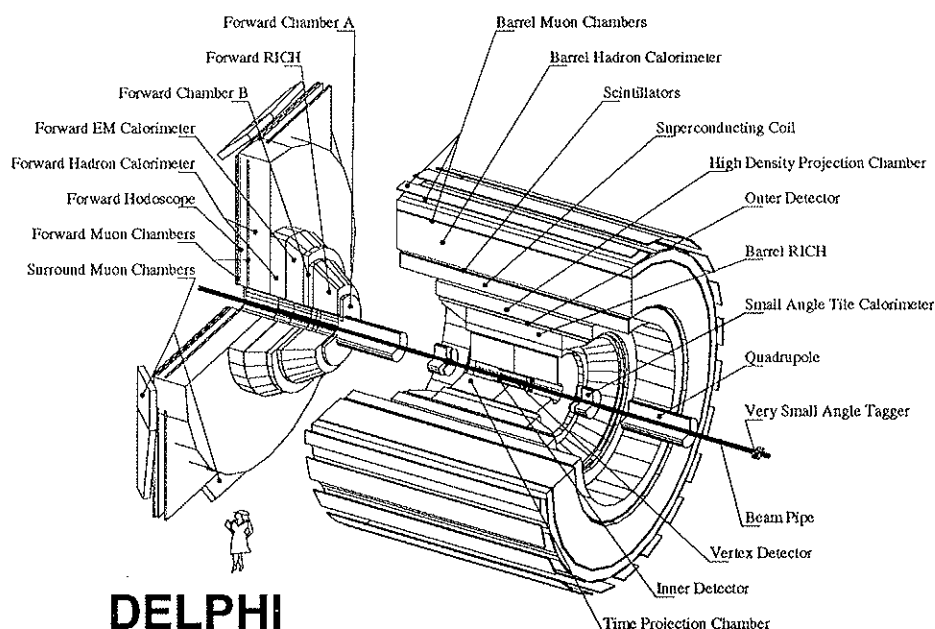


Figure 1 The DELPHI detector.

Figure 1 shows a cut-out view of DELPHI¹. It consists of a central cylindrical section and two end-caps (one shown); the overall length and the diameter are over 10 meters and the total weight is 3500 tons. The electron-positron collisions take place inside a vacuum pipe in the centre of the detector, and the products of the annihilation fly radially outwards. A huge superconducting solenoid - the largest so far built - produces a strong uniform magnetic field (1.2 tesla), bending the trajectories of the charged particles to measure their momentum. To this end, there are several components within the detector devoted to tracking the charged particles to measure their trajectories. The position of each of the charged particles is measured with great precision; the accuracy varies from five micrometers near the collision point, to 0.1-0.2 mm half way out, and to about one mm after traversing five meters of the detector. The Ring Imaging Cherenkov devices measure the velocity of each particle hence

¹ Design and construction of DELPHI took seven years. The present collaboration consists of about 500 physicists, 47 universities and institutes in 20 countries.

enabling the mass to be determined. Other layers called “calorimeters” measure the energy of both charged and neutral particles. Most of the elements of DELPHI provide information directly in three dimensional form, which is read out via some 200,000 electronic channels, 14 dedicated computers, and one main central computer. A typical event requires one million bits of information to be written on magnetic tapes for subsequent analysis.

This analysis is usually done off-line with huge machines calculating for days. However, as there is one event generated every three seconds, analysing all events generated is not feasible. Prior to this analysis, physicists therefore visually look at representations of these events in what is called Online Event Displays (OED). These are complicated systems which collect data for an event, analyse it during run-time, converting the three dimensional recordings into a two dimensional mapping which can be visualized it on the screen of a normal workstation. The physicists can then transform and inspect the events with a minimum of computer power as required, and in an early stage select events that are of interest for further analysis.

1.3 THE PROJECT

Soon a demand for this sort of visualisation of events outside CERN, for instance in education, surfaced. It was therefore decided to try to somehow implement a possibility of showing events without having to sit in front of the machine actually running an OED. A natural choice for this was to use the Internet as a medium being the fastest growing interactive medium in the world right now, and find an appropriate ‘transportation’ method using the Internet.

Three criteria were set as to the nature of this method:

- Accessibility: As many machines and types of machines (e.g. PC, UNIX, Mac) should be able to use it, without any particular requirements to hardware.
- Low network traffic: It should be possible to use it over low bandwidth connections like phone lines.
- Easy user interface: No or little training in using the interface should be required.

My work here at CERN has been to look at different ‘transportation’ methods, select one and develop a functional prototype. DELPHI and its Online Event Display was used as a test case. This paper is about this project.

In addition I have also done work connected to WWW-Support, which is the main activity for the CSE section. This includes general help to users at CERN, updating the HTML-pages and writing small scripts for day-to-day administration of the www.cern.ch-server, assisting in arranging World-Wide Web oriented conferences and courses and the likes.

As I will show, I found that a WWW-interface/gateway to the OED is a solution that fulfils all the above goals:

Any machine that is capable of running a graphical WWW-browser will have access to it. As soon as all data has been transferred, the connection is closed - hence only necessary data is transmitted. The WWW has a very simple interface that many people already know, or are able to learn in relatively short time.

Basically, as there were two incompatible systems that had to interact, a method of communication between those had to be devised. In principle, the problem can be divided into two parts:

- 1 The requests from the WWW-client have to be translated into something that can be passed on to the OED.
- 2 The OED must be able to interpret whatever it receives.

1.4 ACKNOWLEDGEMENTS

It was early evident that I had to work closely with the developer of DELPHI's OED, Serge Du of Orsay, Paris.

I also have to thank Mark Dönszelmann for invaluable support. Without his original ideas and implementations, I would have had a much more difficult task. Also thanks to Robert Cailliau and the DELPHI Collaboration for contributions.

Many thanks to Susana Fernandez Vega and the other members of the PT group who I have been working with. Also a thank to Evgeni Chernyaev whose xwpick220-code I was allowed to use, and to Jan Fyen for interesting historical information.

Furthermore, I would like to thank my flat-mates Eva 'Evita' Sanchez for (almost) always being cheerful and Fanny Wong for teaching me the art of boiling rice. Also thanks to Allan Skillman for British inspiration, the people on IRC-channel #ircNorge for wasting my precious time and to all the people on Internet, especially on the USENET newsgroups comp.windows.x, comp.lang.perl and comp.lang.c, that have somehow contributed in helping me solving problems along the way.

Finally, to Anne Grethe who waited. I love you.

1.5 ORGANIZATION

Chapter 1, Introduction, gives a description of CERN itself, the DELPHI experiment and how this is connected to this project.

Chapter 2, Environment, describes the environment in which the work was done. That is: the Internet, the World-Wide Web and its protocols;

HTTP and HTML, CGI programming, the X Window System and the OED itself.

Chapter 3, Concepts, gives a more thorough definition of the project that resulted in EDWIN, a brief description of the parts and how they interact and how the solution ended up.

Chapter 4, WWW Solution, gives a thorough description of the history and implementation of the system, including problems and different solutions tested.

Chapter 5, Alternatives, gives an overview of what could have been made differently, other implementation methods and possible other tools that could have been used.

Chapter 6, Conclusion, describes the conclusion that was derived.

2 ENVIRONMENT

2.1 INTERNET

2.1.1 What it is

A commonly asked question is “What is the Internet?” The reason such a question gets asked so often is because there is no agreed upon answer that neatly sums up the Internet. The Internet can be thought about in relation to its common protocols, as a physical collection of routers and circuits, as a set of shared resources, or even as an attitude about interconnecting and intercommunication. Some common definitions given in the past include:

- A network of networks based on the TCP/IP protocols.
- A community of people who use and develop those networks.
- A collection of resources that can be reached from those networks.

Today’s Internet is a global resource connecting millions of users that began as an experiment over 20 years ago by the U.S. Department of Defence. While the networks that make up the Internet are based on a standard set of protocols, the Internet also has gateways to networks and services that are based on other protocols.

2.1.2 How it started

In 1969 Advanced Research Projects Agency (ARPA) of U.S. Defence Department (DoD), founded a research and development project to create an experimental packet switching network, which had the promise of letting several users share just one communication line. This network, called ARPANET¹, was built to study techniques for providing robust, reliable, vendor-independent data communications. More specifically, it was designed to withstand partial ‘malfunctions’ (typically like (nuclear) bomb attacks) and still function.

¹ ARPA later changed name to DARPA, Defence Advanced Research Projects Agency, and then back to ARPA again. This has lead to some confusion whether at any point the actual net was called ARPANET or DARPANET. As far as it has been possible to find out, the name never changed from ARPANET.

In the 1970s, ARPA helped support the development of rules, or protocols, for transferring data between different types of computer networks. These “internet” (from “internetworking”) protocols made it possible to develop the world-wide net we have today. In 1975 the ARPANET was converted from an experimental network to an operational one, and by the close of the 1970s, links had developed between ARPANET and counterparts in other countries¹. The world was now tied together in a computer web.

In 1983, the old ARPANET was divided into MILNET, the unclassified part of the Defence Data Network (DDN), and a new, smaller ARPANET. The term “Internet” was used to refer to the entire network; MILNET, ARPANET and several smaller networks that had already been connected to it. By the end of the 1980s, thousands of research companies, government agencies, educational institutions and even businesses had connected themselves to the Internet.

In the 1990s, the Internet grows at exponential rates. Some estimates are that the volume of messages transferred through the Internet grows by 20 percent per month. In response, the U.S. government and other users have tried in recent years to expand the Internet itself. Once, the main Internet “backbone” in the U.S. moved data at 1.5 million bits per second (BPS). That proved too slow for the ever increasing amounts of data being sent over it, and in recent years the maximum speed was increased to 45 million BPS. Even before the Internet was able to reach that speed, however, Net experts were already figuring out ways to pump data at speeds of up to two billion BPS - fast enough to send the entire Encyclopedia Britannica across the country in just one or two seconds.

As of October 1994, a careful estimate was 13.5 million users on 3.5 million computers can use the interactive services supplied by the Core Internet, for example people who can use Mosaic or Lynx to browse the World-Wide Web. If one includes the entire Matrix, i.e. also the people that can only send and receive e-mail, the figure is probably around 27.5 million users. An estimate for the Core Internet is that it will be as big, if not bigger than the Matrix, typically one year after. More than 30 million are estimated within the end of 1995².

2.1.3 How it works

The world-wide network is actually a complex web of smaller regional networks. To understand it, picture a modern road network of highways

¹ An interesting curiosity is that the first international node to be connected was NORSAR in Norway in 1973. NORSAR was then mainly a research institute in the field of seismology for detecting underground nuclear explosions, and the NORSAR seismometer array was, and still is, located very favourable with respect to detecting nuclear tests in the former Soviet Union.

² From MIDS, <http://www.mids.org/mids/>.

connecting large cities. From these large cities come smaller freeways and parkways to link together small towns, whose residents travel on slower, narrow residential ways. To drive on these roads you have traffic rules. The analogy is that you have transport routes, whether they are optical, coaxial, wireless satellite links, etc. with different bandwidth (or capacity) for roads, and transport protocols for traffic rules. Together they make out the Internet and lets network users connect to computers around the world.

In the ARPANET model, communication always occurs between a source and a destination computer. The network itself is assumed to be unreliable; any portion of the network could disappear at any moment. It was designed to require a minimum of information from the computer clients. To send a message on the network, a computer only had to put its data in an 'envelope' and address the packets correctly, in much the same way as the postal mail system works today (utilizing the infrastructure of existing transportation network to deliver shipments from one person to another, irrespective of where those persons are), see figure 2.

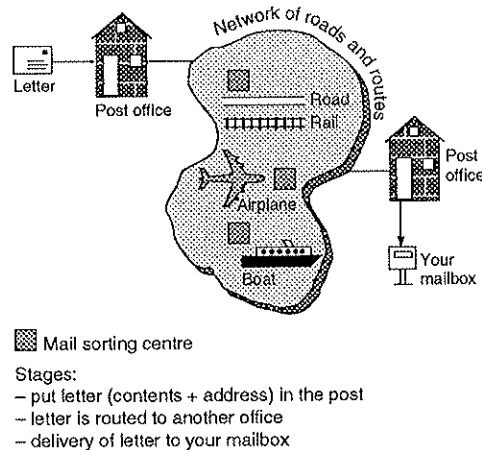


Figure 2 The postal system.

The communicating computers - not the network itself - were given the responsibility to ensure that the communication was accomplished. The philosophy was that every computer on the network could talk, as a peer, with any other computer, see figure 3

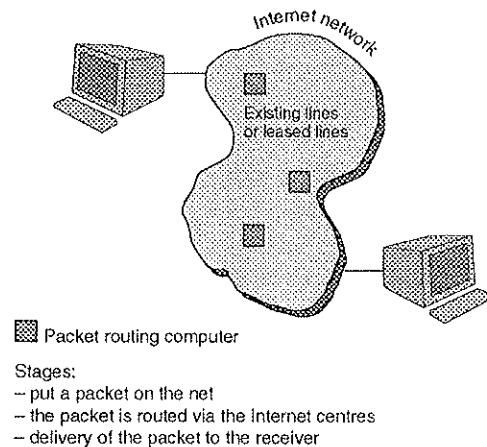


Figure 3 The Internet system.

The original ARPA standard for communication was known as Network Control Protocol (NCP), but as time passed and the technique advanced, and NCP was superseded by a higher-level, more sophisticated standard known as TCP/IP. The Transmission Control Protocol (TCP) converts messages into streams of packets at the source, then reassembles them back into messages at the destination, verifying the correct delivery of data from client to server. TCP also adds support to detect errors or lost data and to trigger retransmission until the data is correctly and completely received. The Internet Protocol (IP) handles the addressing, seeing to it that packets are routed across multiple nodes and even across multiple networks with multiple standards - not only ARPA's pioneering NCP standard - like Ethernet, FDDI, and X.25. TCP/IP is now the common name, derived from these two important protocols, for the original ARPA Internet protocol suite, which consists of a number of protocols.

2.1.4 TCP/IP

The TCP/IP protocols were developed by DoD and adopted as Military Standard by DoD in 1983 and all hosts connected to ARPANET were required to convert to the new protocols¹. To ease this conversion, ARPA funded Bolt, Beranek and Newman to implement TCP/IP in Berkley UNIX (BSD). Thus began the marriage of UNIX and TCP/IP. They have ever since followed hand in hand and helped promote each other.

It was initially successful because it delivered a few basic services that everyone needs (file transfer, electronic mail, remote logon) across a very

¹ The rationale for developing the TCP/IP protocol was that the different armed forces in the USA had incompatible systems. The Army put out a bid on a computer and DEC won the bid. The Air Force put out a bid and IBM won. The Navy bid was won by Unisys. Then the President decided to invade Granada and the armed forces discovered that their computers could not talk to each other.

large number of client and server systems. There were of course other reasons for why TCP/IP became a de-facto standard communication protocol suite on the Internet:

- Open protocol standards, freely available and developed independently from computer hardware or operating systems.
- Independent from specific physical network hardware. TCP/IP can be run over Ethernet, token-ring, telephone-line, X.25 and virtually any other kind of physical transmission media.
- A common addressing scheme that allows any TCP/IP device to uniquely address any other device in the entire network, even if the network is as large as the Internet (though, there is a limited number of addresses possible - it does not scale forever).
- Standardized high-level protocols for consistent, widely available user services.

Within UNIX, communication via TCP/IP is done with sockets. Sockets is a name given to the package of subroutines that provide access to TCP/IP, but when a reference to a socket is made one normally refers to the end-point from where messages are received (or a starting-point from where messages are sent). In UNIX this is a normal file descriptor like the ones that are being used for writing and reading to and from files (on VMS it is a bit different, see chapter 4.1.5.17).

2.2 WORLD-WIDE WEB

2.2.1 What it is

The World-Wide Web is a client-server information system on the Internet. Conceived as a unique way to give particle physicists easy access to their data wherever they works, the Web, or WWW or even W3, has grown into something much bigger. It now disseminates information from businesses, government agencies, universities, schools and even individuals. Documents are linked to each other, forming a single global network of information.

2.2.2 How it started

It began in 1989, when Tim Berners-Lee and Robert Cailliau proposed a distributed information system for CERN based on hypertext. During 1990, the first browser and server were produced and although they were limited to a particular operating system, NeXTStep, their sophisticated information handling capabilities set the standard for everything that has followed. The Web as we know it had arrived.

In 1991, a technical student at CERN wrote a simple browser which could be used on many different computers. Its adaptability to all existing computing systems generated the interest needed to start a snowball effect. Soon the parts of the CERN phone directory could be consulted from a computer in Chicago, the SLAC (an american nuclear research

facility) pre-print database could be interrogated from a desk in Geneva, all without having to know anything whatsoever about the remote computer systems.

In 1993 NCSA, the US National Centre for Supercomputing Applications, produced the X-Mosaic browser. This software allowed the display of coloured images, giving to the Unix platform a glamorous window on the Web. It stirred the exhibitionist in many Unix/Internet programmers, and drove them to set up a Web server showing off scenes of the local site. NCSA also produced versions for Apple Macintosh and Microsoft Windows, thus opening the Web to a large audience, and were soon to be followed by several other browsers, both commercial (e.g. Netscape) and experimental (e.g. Arena).

1994 has been called the “Year of the Web”. The world’s First International World-Wide Web conference was held at CERN in May. This event was heavily oversubscribed. It was attended by 400 users and developers, and was hailed as the “Woodstock of the Web”. As 1994 progressed, Web stories got into all the media. By the end of 1994, the Web had grown from 50 known servers in early 1993 to more than 10,000 servers, of which 2,000 were commercial, and 10 million users. Traffic was equivalent to shipping the entire collected works of Shakespeare every second. Already the largest interactive service on the Internet, the Web is now driving its current phenomenal expansion. Recent monitoring shows the number of servers are doubling every 53 days. By the end of 1995, there could be as many as 1.6 million World-Wide Web servers if this trend continues.

2.2.3 How it works

The WWW uses the infrastructure of the Internet to ship its request from the client machine to the server and then the documents from the server back to the client, illustrated in figure 4.

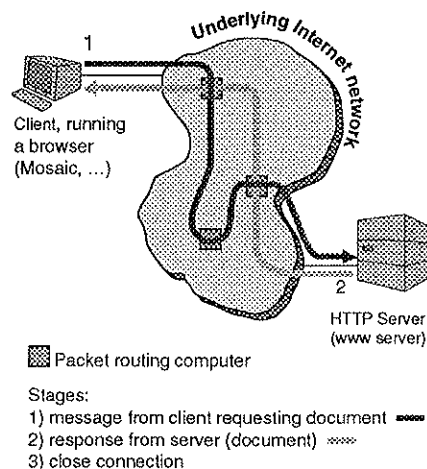


Figure 4 The WWW system.

To use it, you need a computer, a connection to the Internet, and a client application, often referred to as a browser.

When you run your browser, it finds and displays a page of information, which might be held on your own computer (your home page), or fetched from somewhere else, you need not know or even care where it comes from. It works by hiding a network hypertext link behind highlighted items which appear on the screen. The advantage of hypertext is that in a hypertext document, if you want more information about a particular subject mentioned, you can usually “just click on it” to read further details. In fact, documents can be and often are linked to other documents by completely different authors - much like footnoting, but you can get the referenced document instantly.

The World-Wide Web uses not only text for presentation, but also images, sounds, animations and even interaction. This is called hypermedia. Certain words, phrases or images are highlighted, and clicking on them causes the browser to go off and find another page, which probably contains more highlighted items, and so on. For example, starting from the CERN “Welcome page” in Switzerland your next click might fetch a document from a physics lab at the other side of the world. All the information seems to be in the little box in front of you, and in a sense it is. When you click on a piece of highlighted text your browser “phones up” another computer, asks it for the requested information, “hangs up” and displays it on your screen. You are then free to read the new page at leisure, without further consumption of network resources.

The Web has its own transport protocol, HTTP, and its own document structure based on SGML (the Web DTD, called HTML). However, it has also integrated the protocols of Gopher, FTP and telnet and it has gateways to others. Thus the documents available through these earlier systems are also automatically and indistinguishably visible through the Web.

The Web may be used to initiate processes on either the client or the server. A request can start a database search on a server, returning a synthesised document. A document returned in an unfamiliar format can cause the browser to start a process on the client machine in order to interpret it. The Web’s ability to negotiate formats between client and server makes it possible to ship any type of document from a server to a client, provided the client has the appropriate software to handle that format. This makes video, sound and anything else accessible without the need for a single application that understands everything.

Starting processes means it is for instance feasible to initiate a video conference or a terminal session entirely by a click of the mouse on a highlighted phrase in a document.

2.2.4 HTTP

The HyperText Transfer Protocol (HTTP) has been in use by the World-Wide Web global information initiative since 1990. HTTP is an

application-level, client-server protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol, which may be used for many similar tasks by extending the commands, or “methods”, used. With HTTP, the negotiation of data representation allows systems to be built independent of the development of new representations.

The basic functionality of HTTP allows a browser to request a document from a server. Documents are identified by a Universal Resource Locator (URL) that contains the necessary information to select transportation method, locate the server and query for the document.

This format negotiation means that the browser sends HTTP requests to the server specifying:

- What information it wants, e.g. the Uniform Resource Locator (URL).
- What format of information it accepts (txt, HTML, GIF, PS, etc.).

The server replies by:

- Looking for the formats available.
- Checking the quality factor for each of these.
- Returning the information in the best format negotiated.

What this means is that it is possible to implement a new method only by giving it a ‘name’ on the server side and supplying the necessary code to handle it (if necessary) into the server. For instance, it is possible to implement a vector graphics format as a text-file; whenever somebody requests such a document, the server first checks if the browser can handle that particular format (which none at the moment do). If it can, it returns the raw vector graphics data for the browser to display. If it can not it might convert the data into something the browser can handle.

2.2.5 HTML

HTML¹, or HyperText Mark-up Language, is a simple mark-up language designed to specify the logical organisation of a document, with important hypertext extensions, independent of platform. It is not a WYSIWYG word processor such as Word or WordPerfect. This is because the same document may be viewed by many different ‘browsers’ of very different abilities. Thus, for example, HTML allows you to mark titles or paragraph marks, and then leaves the interpretation of these marked elements up to the browser. For instance, one browser may indent the beginning of a paragraph, while another may only leave a blank line.

¹ HTML is an application conforming to International Standard ISO 8879 - Standard Generalized Mark-up Language (SGML), which is a system for defining structured document types

HTML instructions are called 'elements'. These can be divided into two broad categories: those that define how the BODY of the document is to be displayed by the browser, and those that define information 'about' the document, the HEAD, such as the title or relationships to other documents. Every HTML document should contain a head and body text, and use other HTML components as necessary.

In HTML documents, tags define the start and end of elements: headings, paragraphs, lists, character highlighting and links. Most HTML elements are identified in a document as a start-tag, which gives the element name and possible attributes, followed by the content, followed by the end-tag. Start tags are delimited by '<' and '>', and end-tags are delimited by '</' and '>'. There are also stand-alone tags which do not need an accompanying end-tag. For example:

```
<H1>This is a Heading</H1>
<P>This is a paragraph.
```

In start tags, attributes are allowed. An attribute typically consists of an attribute name, an equal sign, and a value, though some attributes may be a value only. For example:

```
<INPUT TYPE="submit" VALUE="OK">
```

A very popular feature is the possibility to have "in-line image". This gives the author the ability to mix graphics and text in the same document. One frequent use of this is to have a small icon which when clicked on, makes the browser download a bigger copy of the same picture, but with higher resolution. Another use is for the graphics to illustrate the text in the same way newspapers or magazines do.

HTML is constantly evolving and is attempting to be standardized by the WWW consortium but that standard is also evolving. HTML is currently defined at version 2, but the final version of the RFC (Request For Comment) has not yet been officially published. There is already active work on-going to finalize the specifications of a version 3 of HTML (previously called HTML+) which contains significant new features. In addition, there is some talk of an interim version 2.1 to standardize a few of the more desired version 3 features earlier than the expected final agreement on a version 3.

Not all browsers implement all features of version 2. In addition, some browsers define and handle some features of version 3 and/or their own extensions. A browser is supposed to ignore any element or attribute of the language that it is not designed to handle.

2.2.6 Forms

Forms within HTML documents (or "fill-in forms") are ways of giving the user an interactive front-end to some application you decide to put behind it. It is typically used for surveys, on-line order forms, feedback or

any Web page in which input is required from the user in order to accomplish a given task or provide a service to the user.

Data is entered through INPUT elements within the form. Each element is associated to a name with the NAME attribute and data is entered by the user.

A form is enveloped by a FORM start- and end-tag. The start-tag has also attributes that describe how the server and browser should handle the submitted form. Submission is normally done either by pressing return or pushing a special submit-button that you usually find in forms with more than one input-field. The two most used attributes are ACTION, which gives the server instructions on what to do when a form arrives, and METHOD, which describes how the data inside the form are sent back to the server.

2.2.7 CGI

The Common Gateway Interface, or CGI, is an standard for running external gateway programs to interface with information servers such as HTTP servers.

What is referred to as gateways are programs which handle information requests and return the appropriate document or generate a document on the fly. With CGI, your server can serve information which is not in a form normally readable by the browser (such as an SQL database), and act as a gateway between the two to produce something which browsers can use.

The most common use of gateways in combination with WWW is the handling of form requests for HTTP. Some examples of the uses of CGI are:

- Converting your system's manual pages into HTML on the fly and sending the result to the browser.
- Interfacing with databases, converting the results to HTML and sending the result to the browser.
- Allowing user feedback about your server through an HTML form and an accompanying CGI script.

Each time a browser requests the URL corresponding to your gateway, the server will execute your program. The output of your program will go back to the browser via the HTTP-server.

In the FORM start-tag, you specify an ACTION, i.e. which gateway to call, and a METHOD. In the case where METHOD="GET", the form-data are sent as the environmental variable 'QUERY_STRING', but you are then normally limited to 255 characters. Or they are sent directly via STDIN (standard input from a process) in the case where METHOD="POST". Because of the limitation of the GET method, POST is preferred. In addition, CGI uses environment variables to send extra server parameters.

To send the output back to the browser, the gateway merely has to print the output, together with a short MIME-header to inform the browser what type of document it is, to the counterpart of STDIN: STDOUT (standard output from a process)¹.

In short, what an interactive WWW-gateway has to do is:

- Receive the form as input from the browser via the HTTP server CGI, usually through STDIN.
- Parse the input. Perform the necessary tasks as a function of the input.
- Build the output from the generated data.
- Send it back to the browser through STDOUT via the HTTP server.

Gateway programs or scripts² are executable programs which can be run by themselves. Any language can be used; some of the more popular languages for use with HTTP include:

- Perl
- C/C++
- TCL
- C Shell
- Bourne Shell

2.2.7.1 Perl

Practical Extraction and Report Language, or Perl, is an interpreted language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. It has built in support for arrays and hash-tables and uses sophisticated pattern matching techniques to scan large amounts of data very quickly.

Perl is more of a 'practical' language in the sense that it is easy to use. It combines features of C, sed, awk, sh and even BASIC and Pascal. On the other hand it is not very strongly typed - modularity is very much up to the programmer, but supports reuse of code to a certain degree.

Usually there is also no work involved moving Perl-scripts from one platform to another as long as the interpreter is installed, which can easily be done for most popular platforms now.

As CGI-scripts normally are small and simple, thus complexity is not much of a problem, and much code and libraries for Perl have already

¹ For a client-server connection, what is STDOUT for one is STDIN for the other. Here it is looked at from the position of the gateway.

² The distinction is made in that a script is parsed every time when called from CGI, while a program is compiled once and is then executable when called from CGI.

been written for handling HTTP and HTML, Perl has become much a de-facto standard for writing and prototyping CGI-scripts.

The disadvantage with using Perl for CGI-scripting is actually the way Perl works. The 'source' code goes through three separate stages before it is finally interpreted and executed. For a script that runs once in a while, this is normally not significant at all. However, for CGI-scripts that is perhaps forked once every minute or perhaps even more often, this adds up to quite a lot of processing time.

2.3 ONLINE EVENT DISPLAY

High Energy Physics experiments investigate reactions between colliding sub-atomic particles. For this purpose, data on the particles leaving the collision point are recorded in large detectors and stored in digital form. The set of data recorded per collision is called an event. Practically all sub-detectors are sampling devices, which for each event, record the tracks of charged particles as a sequence of points, called hits, or record the showers of particles as a set of cells, for which the energy deposited is recorded as well. The events are the basic units for further investigations, which are done by powerful pattern recognition and analysis programs. For checking of these methods and for presentation, the display of single events is an efficient tool, as visual representation is the most efficient way to transfer data from a computer to the human brain.

The Online Event Display (OED) in the DELPHI project has been developed to visualize these events recorded from the DELPHI detector at LEP. It is based on conventional 2D and 3D methods and is designed for general applications.

2.4 X

The X Window System (or simply X) is a network transparent window system which runs on a wide range of computing and graphics machines independent of hardware and operating system. It was developed jointly by MIT and Digital Equipment Corporation, and has been adopted by the computer industry as a standard for graphics applications.

Like other windowing systems, X divides the screen into multiple input and output areas called windows. Input can also be taken from a pointer, usually a mouse. This pointer is among other things, used to describe which window input from other devices (e.g. keyboard) are directed to.

What is unusual about X is that it is based on a network protocol instead of on system-specific procedure and system calls. These network protocols enables X to be relatively easily ported to different architectures. It also allows programs to run on one architecture or operating system while displaying on another using a pre-defined set of requests and replies between two processes; a client application program and a server that controls the display, keyboard and pointer.

The default tool to program X-applications is called Xlib and is a C-library with rather low-level support. On top of Xlib one often finds higher-level tools. The most popular of these is Xt Intrinsic, or only Xt. The purpose of Xt is to provide an object-oriented layer that supports the user-interface abstraction called a widget. A widget is a reusable, configurable piece of code that operates independently of the application except through prearranged interactions. The XTCP-library is based on this Xt-library.

The X-loop is the main 'environment' in which an X-application runs. It can be considered a closed, multiprogramming system which consist of two parts: The application and the X-loop. If the application is not running, the X-loop of that application is; however, it is still one process with the X-loop running as a kernel around the application code. The X-loop deals with sending events that rely on the event queue to the X-server and receiving X-events from the same server (e.g. mouse clicks, keyboard input, etc.).

Whenever the application does something to the display, this request is queued on the event queue for dispatching when the X-loop regains control. This happens when the application 'exits'. However, the X-loop is still running so the X-application in the whole has not exited. For the application to regain control, it normally has to inform the X-loop that it wants to be called again.

3 CONCEPTS

As explained, my main task here at CERN has been to find out if it would be possible to put a functional event display on the Internet, while ensuring that the three main criteria: high accessibility, bandwidth- and userfriendliness, were fulfilled.

Figure 5 shows two views of a the same DELPHI event from different angels. As can be seen, there are buttons for zooming, rotating, adjusting the number of detectors viewed, etc.

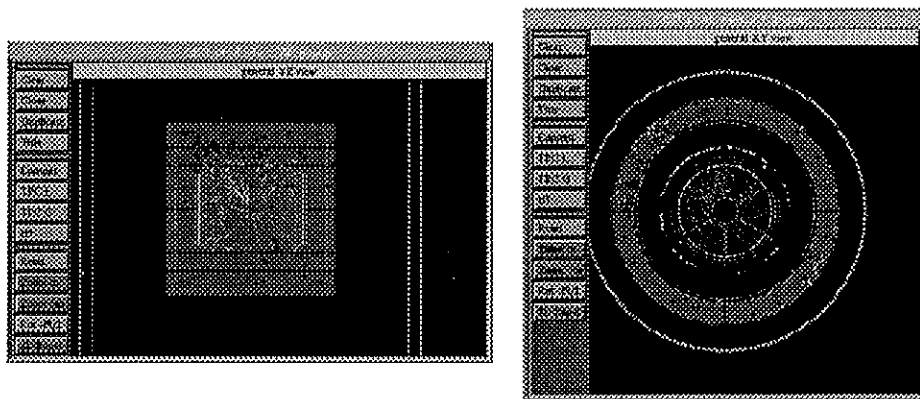


Figure 5 Two views of the DELPHI Online Event Display.

The idea was to recreate the functionality as close to the original OED as possible. All in all three possible concepts were considered:

- 1 An X solution in which the user attached him- or herself to a full-scaled OED running X over the Internet.
- 2 A browser solution in which a special enhanced WWW-browser, or a independent application, was developed which had support for a new data type and special zoom and rotation commands so that it could download the whole event and do all transformation locally.
- 3 A solution in which any WWW-browser can be used, where transformation-commands are sent to a OED-server and the result is sent back to the browser for display.

It was this last alternative that was chosen as it is the one most compliant with the criteria mentioned in chapter 1.3.

3.1 WWW SOLUTION

All in all, the WWW solution requires three main parts to co-exist:

- 1 A WWW-part which acts as an interface to the users.
- 2 An OED-part which receives commands, acts upon them and returns a result.
- 3 A gateway in between which converts between these two initially incompatible systems.

An important concept was to make the system as compact as possible: no need to patch neither the HTTP-server nor the WWW-browser, and as little as possible for the external developer to code extra in his or her application.

3.1.1 WWW-part

The principle of WWW has already been discussed, but exactly how does the browser interact with the HTTP server, and how does the HTTP server interact with the gateway?

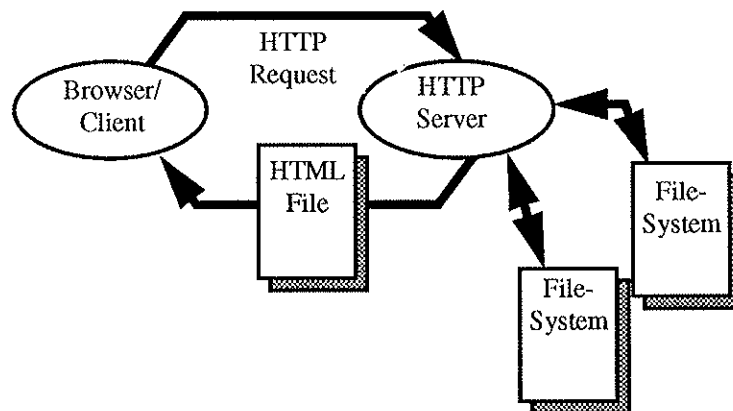


Figure 6 Basic WWW Browser-Server Interaction.

As we can see in figure 6, when the browser sends a request for a document to the server, the server looks up in the filesystem (depending on the rules given in the configuration-file of the server) and returns the file to the browser together with a small header, called MIME-header, to inform the browser what type of file it is (e.g. text/html, image/gif, etc.).

If the requested document is not a file, but an interactive application connected through CGI, the procedure is somewhat different as can be seen in figure 7.

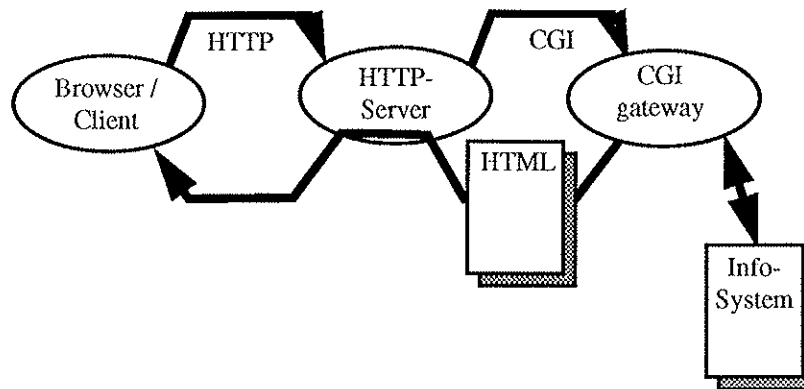


Figure 7 WWW Browser-Gateway-Server Interaction.

As for a normal document, the browser first sends a request to the server. The server then discovers that it is a special document (through the same configuration-rules) and launches the CGI-gateway specified in the URL. The resulting document is then sent back to the browser through the HTTP-server.

3.1.2 Gateway

The gateway looks up the requested information and builds the HTML-file together with any graphics if necessary, and returns it to the browser via the HTTP-server. The CGI-gateway is here responsible for supplying the necessary MIME-header.

More specifically with EDWIN, the gateway receives the data from the HTTP-server that the user has entered. It then parses these data (with the help from the `cgi-lib.pl` library). From this it builds the commands which are sent to the OED, sends these, simultaneously building the output to the HTTP with input from the OED. And finally sending everything off back to the HTTP-server together with the MIME-header.

The main problem with CGI is that it is all stateless. The HTTP-server does not record any information about the current state of any single browser. This means that you cannot do something via a CGI-gateway, go on holiday for 14 days, come back and expect that everything is left the way it was when you left and that the HTTP-server remembers where you were when you left, what you have done so far, etc. The statelessness of the HTTP protocol makes constructing a “stateful” interaction tricky.

What is then taking care of the state in EDWIN?

The OED itself? No. This would involve the OED having to know who has been and who is using the OED at any time. This is not impossible, but rather impractical for the developer.

The browser? No. There is a possibility to hide information in so called hidden fields within a HTML document, where it is possible to store information that the user can not change nor normally see (though, most

browsers have the feature to view the source of the HTML in which these hidden fields will appear together with their respective data). However, this is not supported by all current browsers and there is some uncertainty about the future support in HTML3.0.

The possibility left is the gateway itself. In EDWIN this problem was solved by having a database connected with the CGI-program. Figure 8 shows roughly how EDWIN was implemented.

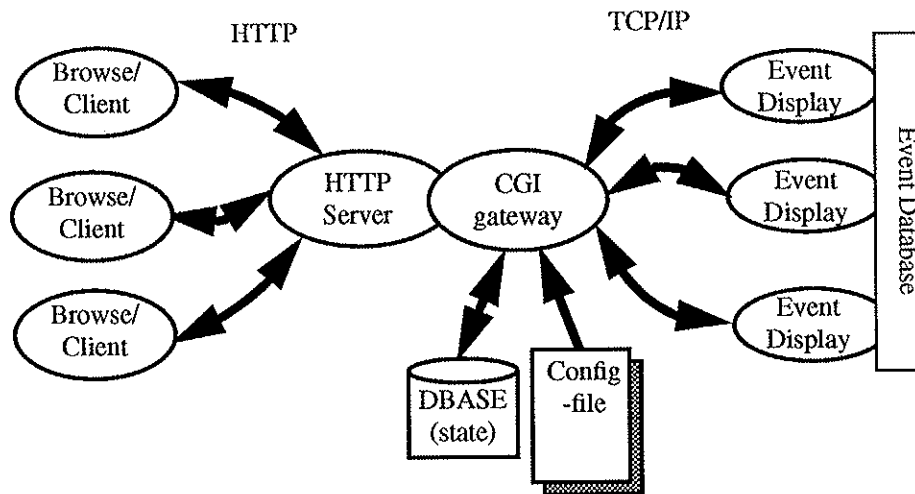


Figure 8 Overview of EDWIN.

When a request for EDWIN is received by the HTTP-server (detected on basis of the URL and the server configuration rules), the HTTP-server calls the EDWIN-gateway script. This script finds out if this was the first request, and if so allocates a free On-line Event Display (OED) server and returns an initial form based on the configuration file (see example in Appendix A.5), to the WWW-client. For the prototype this initial form looks like figure 9.

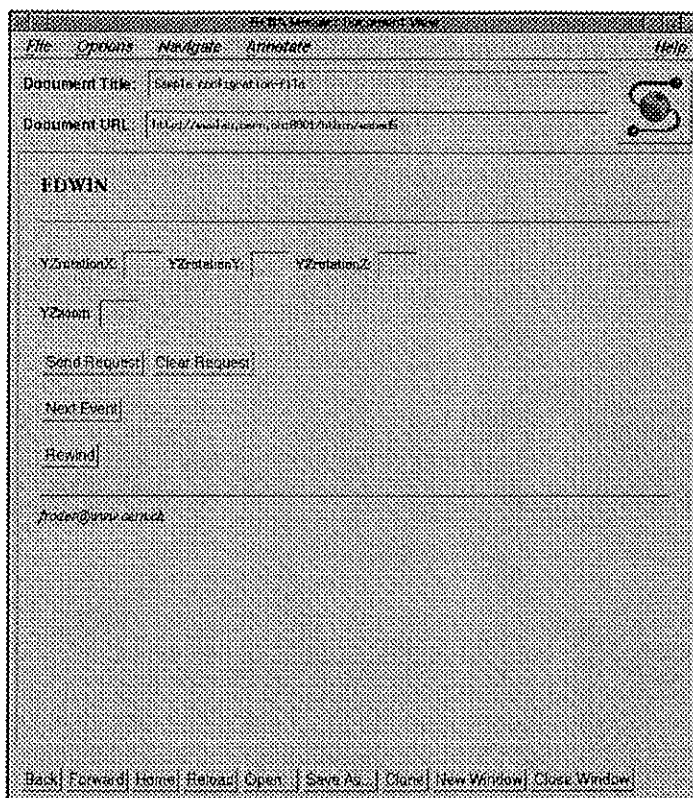


Figure 9 Initial form from the EDWIN prototype.

The user is then free to input data, ask for next event, etc. When one now submits this new request, the gateway will look up in the database and find out which OED this browser has been allocated (if no timeout has occurred). The script then connects to the OED (which has been modified to handle commands through TCP/IP), sends over the necessary commands, waits for the OED to generate the requested image(s), builds an HTML-file with the images in-line, and sends it all back to the client. This new view can look like figure 10.

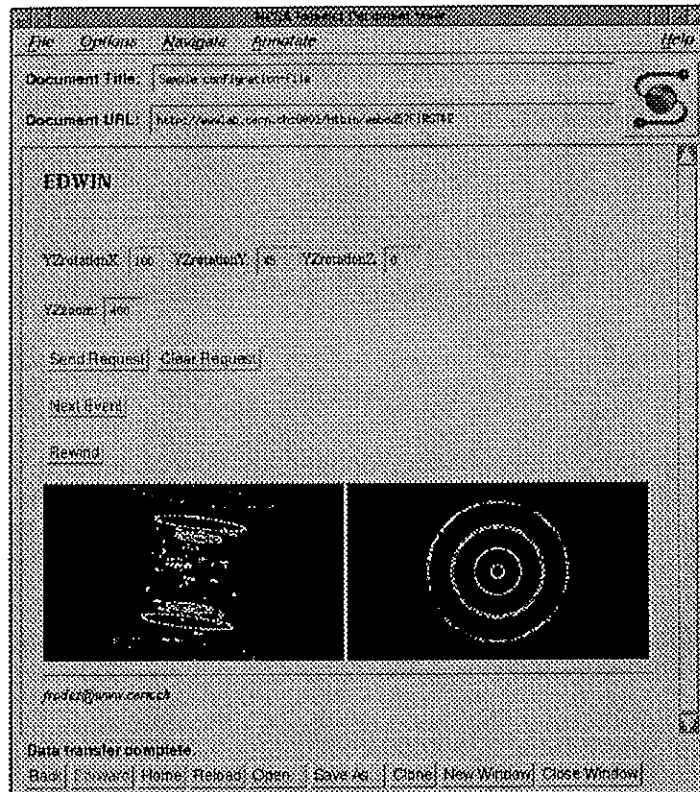


Figure 10 Prototype EDWIN form with rotation and zoom.

Both the commands that are sent to the OED and the appearance of the HTML page in the browser are defined by a common configuration file that is loaded into the gateway each time it starts. This file is a combination of ordinary HTML and special embedded commands. An example configuration file is given in appendix A.

3.1.3 OED-part

Developing on the existing OED code was well beyond the scope of this project as it involves substantial amounts of code. It was therefore instead decided to build a set of library-functions that handles the actual TCP/IP connection and interaction with the X-server itself. It is then up to the OED programmer to build an interpreter around this library.

The XTCP-library, as it was called, consists of mainly two functions for the OED-implementer to deal with:

- 1 XTCPlisten() which sets up a callback-routine, XTCPread(), which again sets up a TCP/IP socket that it keeps listening too.
- 2 XTCPschedule_dump() which sets up the necessary callback-routines to grab a Xt-widget from the screen, convert it and send it back to the client.

Figure 11 shows roughly how the different parts of EDWIN interact.

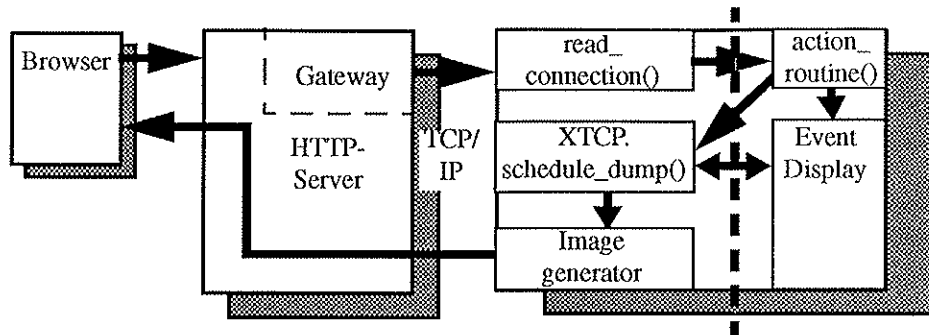


Figure 11 The EDWIN internals.

It is then the developer's responsibility to code the OED in such a way that `XTCPlisten()` is called with the necessary parameters (see chapter 4.1.5.1), to set the OED up listening. The routine `connection_read()` will now frequently check the associated socket to see if anything has arrived at it. As soon as the gateway now connects to the OED and sends a message, `connection_read()` will call `action_routine()` as specified in the `XTCPlisten()` call.

This `action_routine()`¹ is the part that the developer has to write him- or herself. It receives the command-string received by XTCP from the gateway as a parameter, parses this as necessary and calls the associated functions (or basically whatever it wants). Typically if `action_routine()` receives a message telling it to produce a dump of a certain widget, a call to `XTCPschedule_dump()` is normally made together with the identifier for the desired widget. An example of an implementation of `action_routine()` is given in Appendix A.3, in the example called `receive_action()`. This is the code that was used for testing purposes.

The `XTCPschedule_dump()` will then take the contents of the specified window (through a widget identifier) and run it through a GIF-converter (Graphics Interchange Format - the most widely used format for in-line images in HTML), developed with the help of `xwpick220`-code by Evgeni Chernyaev. Depending on the method requested, the image is then either written to a file or sent directly back via TCP/IP to the gateway.

A return to `connection_read()` is always made as the last event within such a session, to close the current connection and set everything up for a new session.

A more thorough discussion of all aspects in these three parts is given in chapter 4.

¹ When reference is made to `action_routine()`, it is not a specific named routine, but a generic name for the routine that the developer has to write him- or herself. He or she can call it exactly as they wish as long as they remember putting the correct name into the parameters in `XTCPlisten()` (see chapter 4.1.5.1)

3.2 GENERAL

Tests were done using simple telnet-sessions with visual inspection. This got a bit inefficient, and a short script was made to automate this; at the same time trying out techniques for use in the CGI-gateway.

Development was done on a SPARCstation 20 running the SunOS4.1.3 UNIX operating system. However, as the finished system was going to run on VAX or Alpha machines running the VMS operating system, it had to be tested for portability all the way during the development.

The GNU C Compiler, or GCC, version 2.6.3 was used to compile the C-source on the SPARCstation 20, while DECC was used on the VAXstation 4000-90 that was used as the target machine for the prototype. For debugging the GNU debugger, or GDB, version 4.14 was used on the SPARCstation 20, while the internal kernel-debugger was used on the VAXstation.

The Perl interpreter version 4.0pl36 was used for the CGI-gateway on the SPARCstation 20 where the CERN HTTP-server version 3.0 was running as an experimental WWW-server. Two external Perl-libraries were used: time.pl by Ove Ruben R. Olsen and cgi-lib.pl, version 1.8, by Steven E. Brenner.

3.3 HISTORY

As mentioned, this project was closely coupled with the development of DELPHI's next OED done by Serge Du. He was working on an entirely new OED when he was approached for this project. He suggested that he implemented the necessary extras parallel with the current development. This to minimise the delay in development on both parts and ensured up-to-date software technology being used. All specifications were discussed through electronic mail (e-mail), and implementations and tests were run by both of us.

While everything was being worked out with Serge Du and he was far enough with the new OED to start experimenting with, I wrote a short paper roughly describing how I pictured the implementation of WEBED, as it was preliminary called at that time, could be done. This paper is in Appendix B.2. Further develop on the XTCP-package that Mark Dönszelmann provided was then initialized.

The gateway-script and the XTCP-library followed each other closely in the development, but as it was the library that was most important to get as functional as possible, it was the gateway-script that more followed the library than the other way around.

Table 4 shows a rough timeline of important events that happened during the development process.

3.3.1 Timeline

Table 1:

Week	Objectives
1	Arrived at CERN.
2	Familiarizing with CERN, bought car and found a place to live.
3	Preliminary report on the project with working title 'WEBED'. Familiarizing with exciting code.
4	Moved from drug nest into a shared apartment.
6	Initial experimentation with XTCP. Makes an RTF2HTML-gateway for testing of CGI and Perl.
10	CERN World-Wide Web Days. Arranging and preparations.
12	Development begins on interfacing XTCP and OED by Serge Du and myself.
15	Easter Holiday.
16	Working prototype of OED that can accept commands, but no window-dump.
17	XTCP implemented with window-dump.
18	Re-implementation of the state machine to accept multiple commands at once.
19	First working prototype of OED with window-dump.
20	Implementation of random filename generator.
23	Re-implementation of state-machine to include close after each connection. Project given the name 'EDWIN' (Event Display with World-wide web INterface).
24	Report-writing main priority. Presentation of EDWIN at CERN.
26	Final touches on report, Termination.

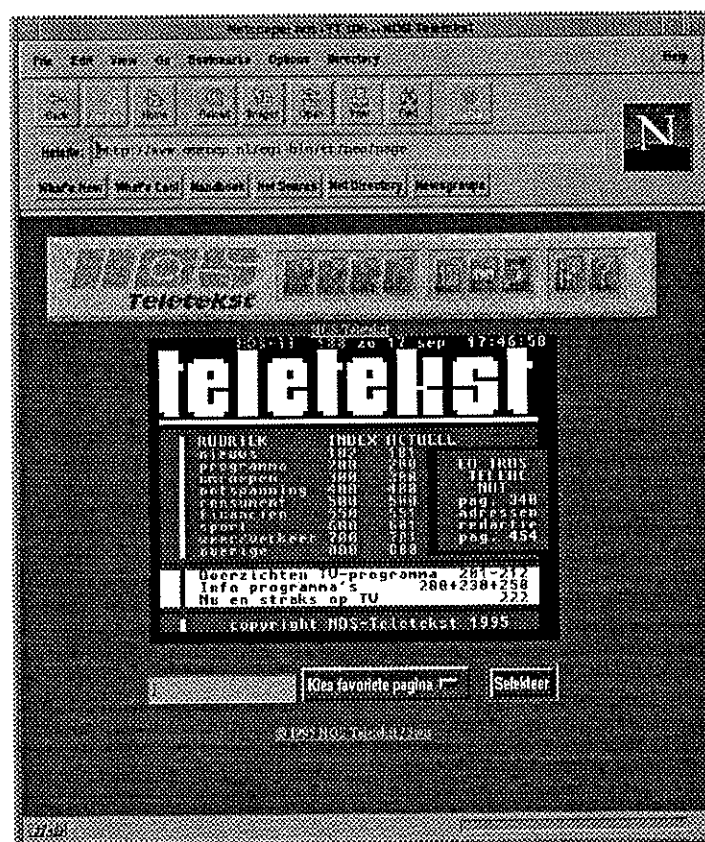
3.4 EXISTING EXAMPLES

Today, there are a number of interactive WWW-applications. They include games, database-interfaces, currency-converters, text-converters, surveys, e-mail-gateways, etc. Most of these either route the data to another application and never bothers with it again (e.g. e-mail-gateway), or have a set of resources located in which the gateway looks up the

necessary information and builds the resulting HTML-document (e.g. database-interface).

3.4.1 Teletext

The only other application that used a connection to an independent realtime application that was found, is the NOS (Nederlandse Omroep Stichting) teletext-gateway¹. It has, as can be seen in figure 12, a frontend looking very much like a genuine teletext screen, and works by allowing the user to click on the desired page numbers. Whenever a user clicks on a page number, the gateway goes off and fetches that page from the on-line teletext system and displays it ready for further clicking.



extra cost to the client with an extra process to maintain the connection and the state of that connection. However, this solution only works with one browser on a very limited number of platforms.

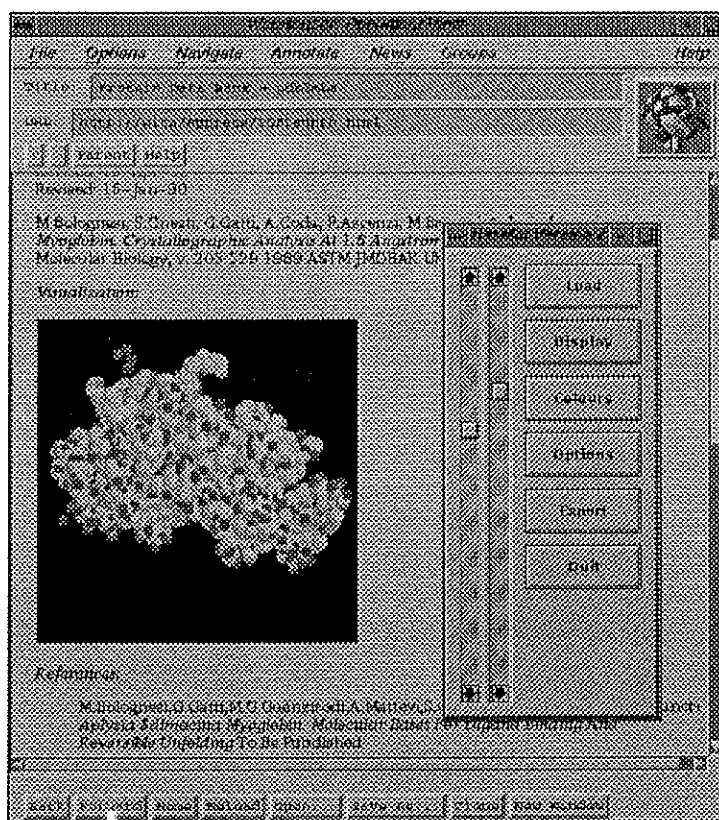


Figure 13 Screenshot of Eòlas' enhanced Mosaic-browser showing a molecule-demo.

4 WWW SOLUTION

4.1 XTCP

The XTCP-library was developed from Mark Dönszelmanns original 'skeleton-library' of the same name. It acts as a frontend between the TCP/IP connection and the host X-application, in this case the DELPHI Online Event Display.

Any one of the current IPC (InterProcess Communication) methods could have been used, but TCP/IP was chosen as it:

- does not restrict itself to a single system (like pipes, named pipes, message queues, shared memory, domain sockets, etc.), i.e. the gateway and the OED can run on different machines,
- is very portable - practically all computer systems support it, i.e. porting is less of a problem,
- is connection oriented and therefore more reliable than connection-less protocols.

It was important to ensure that the library was as self contained as possible: As few functions as possible for the user (i.e. developer) to deal with, but at the same time make it complete. With mainly two public functions in the finished prototype, `XTCPlisten()` and `XTCPschedule_dump()`, this goal can be considered reached.

During the development, several solutions were tried out. These can be divided into four stages, where the fourth is the finished prototype. Before a rather complete discussion on the finalized library, a short summary of these stages now follows.

4.1.1 Stage I

This stage involved early testing of TCP/IP and X-interaction. A small X-application was written and linked with the early XTCP-library. It consisted of merely a window where text could be displayed. Whenever it received a connection, it took the string sent through that connection and simply displayed it for inspection.

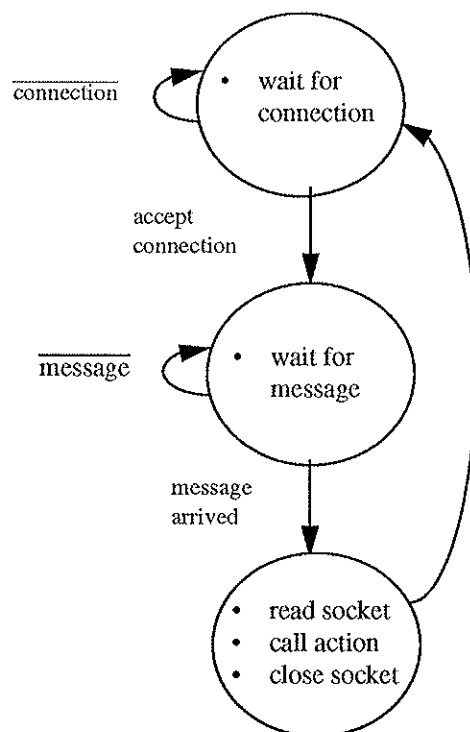


Figure 14 STD of XTCP in stage I.

Figure 14 shows an early, and not completely by-the-book, draft STD (State Transition Diagram) of how we thought implementing this early stage. It initiates by starting to wait for a connection. As soon as one is detected it goes into the state of awaiting the message, and when that arrives, XTCP takes the message and calls the `action_routine()` in the X-application. Upon return from `action_routine()`, XTCP immediately closes the connection and goes back waiting for a new connection. If more than one messages has to be transmitted, one would have to open several connection in turn and send one message at a time.

The client-server interaction can roughly be described as in the Client-Server Transition Diagram (CSTD)¹ in figure 15. (Note that in the following figures describing the client-server interaction, a focus has been put on the server part as the client-side will be described later).

¹ This type of diagrams can, as far as it has been possible to find out, not be found in any reference literature and are rather experimental. Personally, I found it a nice way to depict the logical flow of data through the client-server connection.

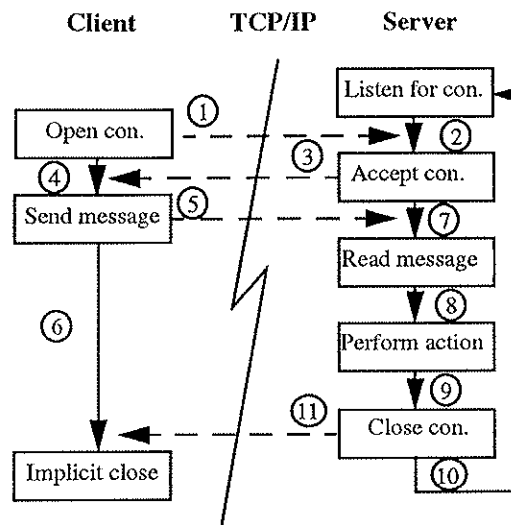


Figure 15 CSTD of stage I.

The events that triggers the transition between two states, whether they are on the same side of the connection or not, have been numbered, but note that they do not necessarily have to happen in that order. Usually both the client and the server run on multitasking machines where one cannot guarantee that one event happens before another in two unrelated processes, especially if they run on different machines (there are ways, but then the processes are no longer unrelated). However, this is not much of a problem as TCP/IP takes care of queuing of data until the receiver is ready to receive it.

What goes over as dotted lines is TCP/IP traffic, which includes control information (1, 3 and 11) and data (5). The other numbered arrows can be imagined as messages which trigger another state. “Perform action” in this context means that the application programs `action_routine()` is called. When it has finished processing one connection, it closes this and goes back listening for a new one.

At that moment the content of the strings were not important and telnet sessions, and later on perl-scripts simulating this client connection, were used for testing.

When finally an early prototype of the OED was ready for testing, linking them together went surprisingly without any problems, and it was possible to issue OED-commands for visual inspection.

4.1.2 Stage II

Naturally, in this stage II, a feature to dump the image of a specified window (or XT-widget) was implemented. This client-server interaction is shown in figure 16.

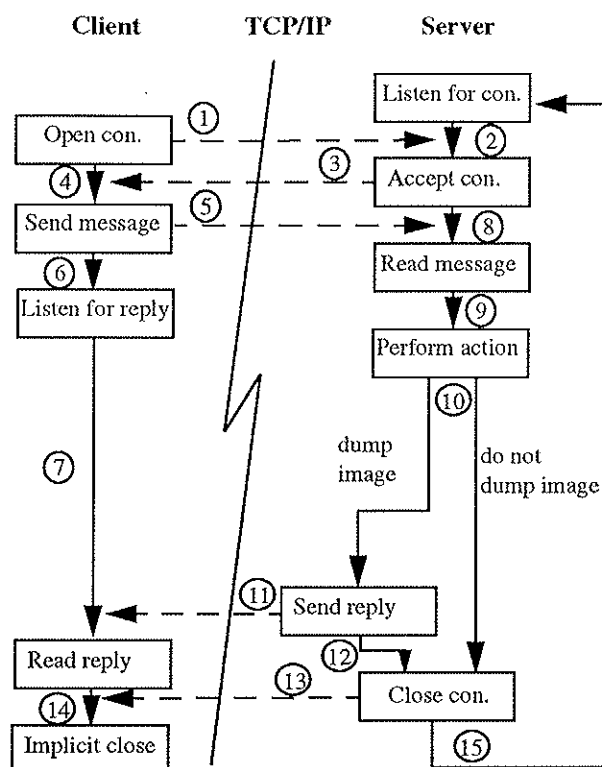


Figure 16 CSTD of stage II.

First of all, it is important to note that the decision at 10 in the figure, whether to dump or not to dump an image, was made within the application program, not the XTCP-library. When `action_routine()` was called, it had to parse the parameterized message. If it decided that the message was "dump image", it called the old library-function for doing that, `XTCPdump_image()` (this has been replaced in the prototype by `XTCPschedule_dump`), otherwise it behaved exactly as in stage I.

`XTCPdump_image()` would on its part grab the window and dump it to file, in which case it would send a references to the file over the TCP/IP connection, or simply dump the whole image over the connection. However, first, for both methods, a small header was sent telling what came next. This header could be one of the strings in table 2.

Table 2:

Type	Meaning
FAILED	Indicating an error from the server, e.g. unable to open file
URL	Indicating that the next line is a URL to the remote image file
RAW	Indicating that the next sequence of lines are raw image data

For instance, the messages sent from the XTCP to the client during a single connection would for a raw image dump, look something like this:

```
RAW
[ARBITRARY AMOUNT OF BINARY IMAGE DATA]
<CLOSE CONNECTON>
```

When everything had been sent, XTCPdump_image() would return back to action_routine() which in turn would return back to the XTCP-library which finally closed the connection.

This scheme worked fine with our simple test program (as seen in Appendix 4.3) - just because it was simple. When we linked XTCP with the full OED, it always closed the connection before anything was sent back over the connection. The trouble was that the OED itself used an internal call-back system which basically meant that it would schedule the dump itself immediately returning back to the XTCP-library, which would closed the connection, before the OED got round to work on its call-back queue.

4.1.3 Stage III

A more elaborate method was devised which involved having a continuous connection which had to be explicitly closed by a close-message from the gateway. With this method one could ensure that all responses from the OED would be received before the gateway sent the close-message. This basically meant that all messages sent to the OED had to give some sort of acknowledge before the next message could be sent. It also had the advantage that there was no overhead involved with opening and closing the connection rapidly.

Again, the first thing sent over the connection was the header as in stage II. In addition, after everything had been sent, a final close-message was always sent. The header strings was then as in table 3.

Table 3:

Type	Meaning
FAILED	Indicating an error from the server, e.g. unable to open file
URL	Indicating that the next line is a URL to the remote image file
RAW	Indicating that the next sequence of lines are raw image data
END	Indicating everything had been sent

Due to the encoding method of the GIF picture format, encoding a given sequence of bits to another, sequential increasing sequence of bits which, there is a high degree of probability that the line-separator ($\backslash n$ = ASCII

value 13 dec) will appear inside a GIF-image. It is therefore not possible to guarantee a certain number of lines to be sent to the client after the 'RAW'-header. Instead of having to send an indicator on how many bytes will be sent, the 'END' string was sent as the last response to all requests, even when nothing else was sent, acting as an 'acknowledge'. In this way the client would know that when it received the 'END' string, the server had finished transmitting.

The same example as in stage II would then look like this:

```
RAW
[ARBITRARY NUMBER OF BINARY IMAGE DATA]
END
```

The client-server interaction in this stage is shown in the CSTD in figure 17.

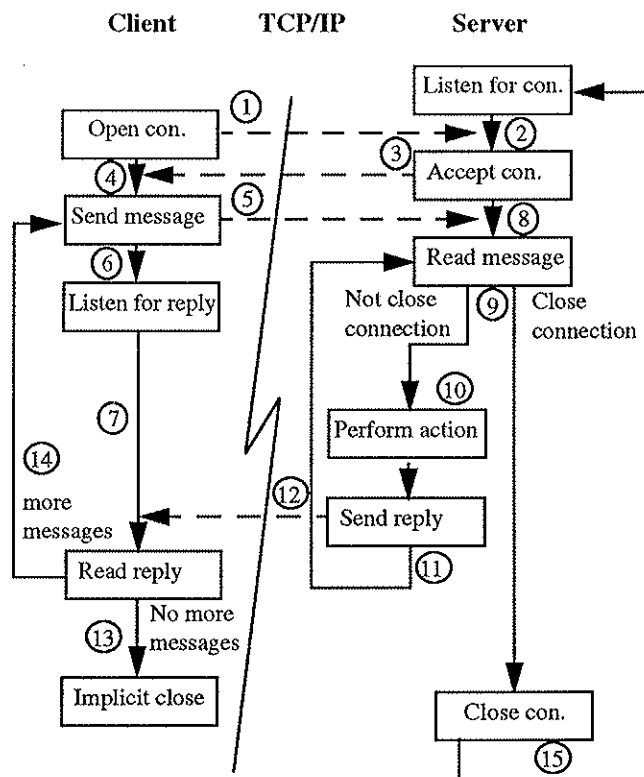


Figure 17 CSTD of stage III.

It differs from stage II with respect to that a request for an image dump is treated exactly as any other command, and is executed accordingly, while the "close connection" message is treated specially; XTCP would only close the connection when explicitly told to do so by the client through a special "close connection" message. This also involved having one extra command, XTCPclose(), for the developer to deal with - something we tried to avoid as much as possible.

There were a number of problems connected with this scheme. First of all the client program (i.e. the gateway) had to be more complex to handle the different possibilities of responses from the server. In addition, if the connection went bad (e.g. by the client-program dying), there was no guarantee that the connection would go down by itself; one would either have to rely on timeout of the connection (which takes minutes) or write in more support code for handling error situations.

However, the most serious problem came from the asynchronous nature of X, which basically means that you can never be sure of in what order X-commands are served. In practise this means that it is possible to first send a command to the X-server to request an update of the display, and then immediately after send a dump-image command which grabs the screen before the update has been done (or perhaps even worse, half-way through). With this problem it does not help to use acknowledge replies as they will be received by the gateway, indicating that everything has been done, before the application has released the control to the X-loop.

This scheme relied on the client waiting some seconds between sending each message. This is not effective use of the OED as the various operations performed take different amounts of time depending on the load on the machine and the X-server and the complexity of the operations.

4.1.4 Stage IV

Stage IV basically involved mixing something from all previous stages, but implementing it slightly differently. The CSTD of this stage is show in figure 18.

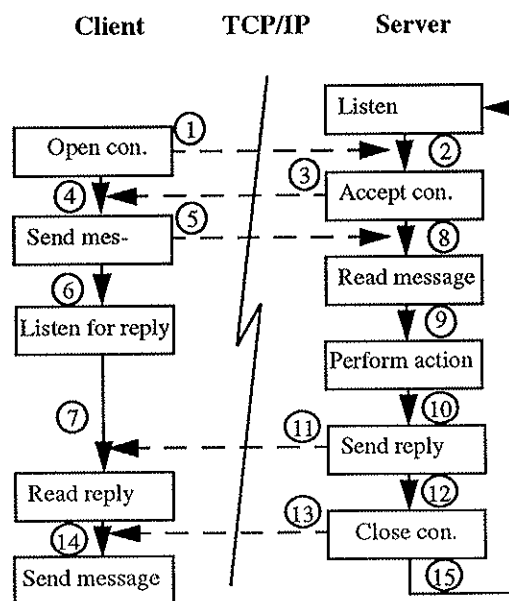


Figure 18 CSTD of stage IV.

Basically, the client-server interaction is much the same as in stage I except that here the client also has to receive feedback from the server on what it has done, similar to stages II and III. However, as the server closes the connection immediately after everything has been sent, there is no need to explicitly send an 'END'-string to indicate the end of the transmission. This also has the advantage that if anything should happen to the client, the server will still close the connection and be ready for a new session.

So, how do we ensure that everything is actually sent, and that there are no more X-events laying in the event queue before dumping an image? The solution is to make use of the X-loop and the possibility to schedule functions with lower priority than the events on the event queue, for later execution within the X-loop. In the Xt-library, which the XTCP-library uses, this is done with the system-call `XtAppWorkProc()`. Each call to `XtAppWorkProc()` puts a function on a separate stack, which is emptied after the event queue when the application returns to the X-loop. This ensures that the X-loop has finished its priority tasks, e.g. updating the screen, before the callback stack is emptied.

In addition, a support for resizing the final image is also provided. The reason for this is to allow greater flexibility to have, e.g. several small 'icon' type views which when clicked on are expanded to full sized views.

4.1.5 Implementation

A description of each individual function of this solution will now follow; refer to the commented source code for technical details. Public functions are prefixed with XTCP for clarity.

4.1.5.1 XTCPlisten()

This is the function the developer initially has to call to set up XTCP from his or her application. It takes its calling parameters and places them into a data-structure associated with the connection. Additional space is also provided for parameters that are used during run-time.

Then the initial callback procedure to `connection_read()` is set up so that the X-loop knows which function in the library to call first when handing over the control. For this purpose `XtAppAddTimeOut()` is used. It takes as parameters the application context (as multiple instances of an Xt-application may be implemented in a single address space, the context is used to distinguish one application instance from another), a time-out value in milliseconds, the function pointer to `connection_read()` - which basically is the 'engine' in the system - and the parameter for that function (the aforementioned data-structure).

Finally it initializes the random part of the temporary file name by calling `strand()`.

When this function now exits, control is released back to the main application which called it. From there the control must sooner or later be given to the X-loop, which will again give, at some point, control back to

`connection_read()` in the XTCP-library. `Connection_read()` will on its end have to release control back to the X-loop in an eternal loop until the application as a whole exits.

4.1.5.2 XTCPschedule_dump()

This is the second of the two important public functions. This is not the function that actually performs the dump of the image, but merely schedules it, hence the name. It first checks to see if it has to do any resizing on the widget-window, if so it also calculates the height ratio from the width specified. It then takes the necessary parameters and places them into a data-structure associated with that particular dump.

If the size is not the desired one, it sets up callback functions for resizing and repeated testing of the widget (note the reverse order - see code). Again, this has to be done with callback functions because the actual resizing of the window belonging to the widget has to be done by the X-server from instructions given to it by the X-client from within the X-loop.

If the size is correct, callback-routines for the actual dump are set up. This includes synchronization of the display (although this should not be necessary, it was recommended in the Xlib documentation) prior to the dump, and a call to clean up after the dump.

4.1.5.3 XTCPwrite()

This function writes a given string out to the connection. It is supplied as a public function so that developers can write directly to the client to supply additional information, e.g. status messages, if needed.

4.1.5.4 XTCPunlisten()

Closes the connection completely and frees up the memory allocated to the XTCP data-structure. To set up the connection again, a call to `XTCPlisten()` must be made.

4.1.5.5 XTCPportno()

Returns the portnumber of the connection specified.

4.1.5.6 XTCPuserpar()

Returns the user parameters of the connection specified.

4.1.5.7 XTCPappcontext()

Returns the application context of the connection. specified

4.1.5.8 XTCPtimeout()

Returns the time-out value of the connection specified.

4.1.5.9 connection_read()

This is the 'heart' of XTCP. It basically does all the work involved in initializing the port to listen to (bind and listen to the socket), the accept of an incoming connection and read one line from that connection. It then calls the users action routine before it closes the connection, provided that the dump, if any, has been done.

The STD for this final version looks like figure 19.

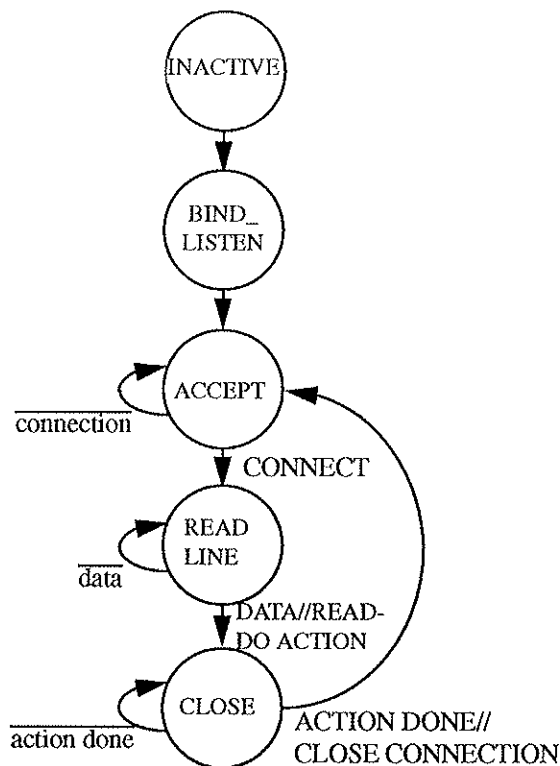


Figure 19 STD of stage IV.

Each call to connection_read() will potentially, given the criteria is fulfilled, shift the control downwards to the next state starting with INACTIVE and circulating from CLOSE back to ACCEPT. Otherwise, it will remain in the current state. Also note that this is a state diagram, e.g. the state CLOSE does not close the connection itself, it waits until it is safe to close the connection and as it exits, the connection is closed.

The differences between this STD and the one shown in figure 14 are the two extra states that have been introduced on the top; INACTIVE and BIND_LISTEN. In addition, the 'close socket' part of the old STD (see figure 19) has been moved to a state of its own, named CLOSE, at the bottom. The reason for this is to let the application return to the X-loop at least once so that it can update the display if necessary and execute any intermediate code (e.g. for image dumping) before closing the connection.

All functionality related to the handling of the connection has been put into this `connection_read()` for simplicity.

An extra state, `INACTIVE`, which does nothing except set the state to `BIND_LISTEN`, was introduced to provide for possible, future extensions to the setup sequence.

It finally reschedule itself using the `reschedule()` function.

4.1.5.10 `reschedule()`

This function is used within `connection_read()` to reschedule itself so that the state machine never terminates (the latter can however be done with `XTCPunlisten`).

Originally two methods were used: `TIMEOUT` and `ADDINPUT`. The latter of these takes advantage of a UNIX feature that involves setting up an interrupt which is invoked when something arrives at a file descriptor, in this case a socket. This possibility is unfortunately not available on VMS.

On VMS you therefore have to use the `TIMEOUT` mechanism, which basically involves polling, or continuous listening, at the socket. As can be seen in the XTCP-header file (Appendix A.1), the option of using `ADDINPUT` has been commented out, so that now all platforms use the `TIMEOUT` method.

This was done to keep `connection_read()` as simple as possible. Using `ADDINPUT` requires 'something' to happen on the socket before `connection_read()` is executed again, e.g. the state machine as shown in figure 19, will switch from the `BIND_LISTEN` state to the `ACCEPT` state as soon as a connection arrives at the socket. It would therefore be impossible to use this technique to close the socket (i.e. go from the `READ_LINE` state to the `CLOSE` state) after all the data had been transmitted, unless something is returned from the client, e.g. an acknowledge. This is not necessarily a bad thing, but has the same disadvantages as having a connection open during the whole client-server interaction as described for stage III.

Supporting both techniques would result in having extra code inserted into `connection_read()` to differently deal with the problem of closing the connection, increasing the complexity of the routine. For a prototype, this was not considered necessary.

4.1.5.11 `connection_close()`

Closes the current connection.

4.1.5.12 `resize_window()`

This function performs the actual resizing of the window associated with the shell widget of the widget associated with the connection.

4.1.5.13 `resize_wait()`

As `resize_window()` has to be executed from within the X-loop, this routine is needed to ensure that the window is of correct size before grabbing and dumping of the window is performed. If the window has incorrect size, it reschedules itself successively using a `XtAppAddTimeOut()` until the window is of correct size. Normally, this happens only once as the X-loop will try to make all necessary updates during the period before the next time-out arrives, which invokes a new call to `resize_wait()`. But if the X-server is busy, this might not always happen, and it will therefore reschedule itself with successively longer time-outs.

When the window has the correct size, it calls `XTCPschedule_dump()` again, which schedules the dump directly as the window is then of the correct size.

4.1.5.14 `shell_widget()`

This function traverses up the Xt-widget tree and finds the shell instantiation of the widget. This is necessary because the window is associated only with the shell widget.

4.1.5.15 `dump_image()`

This routine performs the actual grabbing, proper encoding and dumping of the image.

It first gets the position and size of the window before it actually grabs that piece of the display in which the window resides. Note that this can be a problem if the window that is grabbed is being obscured by other windows. However, this was not a problem for the OED as it always brings the window at the front for every update.

The window is stored in an internal X-representation as an `Ximage` pointed to by a global pointer-variable. The pointer had to be global because of the limitations imposed by `GIFencode()` in the `ImgToolkit`-library using the `get_scline()` and `put_byte()` functions.

It then normalises the colours to 8 bits. This is done because most X-servers operate with colours up to 24 bits, and the GIF-format can only handle 8 bits.

It then selects between two possible transfer methods, URL or RAW, sending a message "FAILED" if the request did not match any of them. Otherwise, it will send back the strings 'URL' or 'RAW' respectively.

For the URL method, `GIFencode()` will be called with the global file descriptor set to the recently opened file with a randomly generated filename, where the GIF-image will be written. A URL is then also sent back using the environmental variable `ENV_EDWIN_SERVER` or defaults as specified in `xtcp.h` (Appendix A.1) along with the random filename generated.

For the RAW method, GIFencode will be called with the global file descriptor set to the already open socket. This results in the image actually being sent as binary data, out on the socket to the client.

4.1.5.16 get_scline()

This function is used by the GIFencode() to grab one line of pixels from the Ximage captured in dump_image().

4.1.5.17 put_byte()

This is more precisely two functions: file_put_byte() and socket_put_byte(). They perform exactly the same function, write a byte to a file descriptor. They share a global variable for the file descriptor and are, as get_scline(), also used by GIFencode().

Ideally it should not be necessary to have two functions which does the same, but as VMS deals with writing to sockets with a different system call from UNIX, two different functions had to be used. To write to a file, the normal write-system call is used on both systems. To write to a socket, the same system call is also on UNIX, but for VMS a special socket_write()-system call was used. Hence, on UNIX 'socket_write' is aliased to 'write'.

4.1.5.18 sync_display()

This flushes the display using first XSync() to flush the output buffer, and then XPending() to empty the event queue. As this is executed from the X-loop, it should not really be necessary, but is recommended by the Xlib-documentation.

4.1.5.19 free_dump()

Releases the wait for the close and frees up allocated memory for the dump data-structure.

4.1.5.20 tmpname()

This function generates a random filename. There are already a number of existing functions that do this, but all of them have the limitation that you cannot supply an filename extension of your own choice to the name. As it is necessary for the HTTP-server to have a .gif-extension for it to recognize gif-files, a separate routine had to be implemented.

First of all, it builds part of the file name that starts with the directory specification and prefix using the environmental variables defined in xtcp.h as ENV_EDWIN_TMP and ENV_EDWIN_PREFIX, or respective defaults.

It then generates the rest of the filename with a random sequence of RANDOM_LEN letters and a counter that increases sequentially. The idea is that if two or more servers using XTCP share the same directory

for the GIF-files, they normally do not select the same random sequence, so a sequentially increasing counter is then enough to separate the files.

If a clash occurs, the instantiation that made the crash occur selects a new random sequence of letters and resets the counter. The counter has scope for quite a lot of files. In the case of a random sequence of three letters, length determined by the definition of `RANDOM_LEN`, and an unsigned integer as the counter, this gives in total 603,906,761,555,968 different file names, independent of choice of prefix.

4.1.5.21 `strand()`

This function creates a random sequence, with length defined by `RANDOM_LEN`, of letters in the range 'A'-'Z' or 'a'-'z'.

4.1.6 Improvements

What is most lacking in this prototype is error-handling and -control. There is only minimal error-checking associated with the connection to the gateway and support for reporting error is limited to two cases:

- If it is not possible to open a file for writing of the image to disk.
- If the transportation method selected is not valid.

During prototyping this was given a low priority, but is needed to make it a more robust tool.

4.2 CGI GATEWAY

On the first request for EDWIN from, an unique id-number is allocated to the client. The id-number is made part of the URL, and is therefore submitted to the gateway on all successive requests.

As previously mentioned, the development of the gateway had to follow the development of the XTCP-library. This required fast prototyping of different solutions more or less constantly, and was the main reason why Perl was chosen as programming language for the gateway.

The flow-chart in figure 20 outlines the baics of the EDWIN gateway. A discussion on how the different subroutines interact follows, and additional flow-charts give a graphical representation where needed. However, first of all an explanation of the configuration file is necessary.