Building and Prototyping Groupware

Alan Dix and Devina Ramduny

http://www.hiraeth.com/alan/tutorials/groupware/

These notes were produced for a tutorial given at HCI'95, Huddersfield, 29th August 1995.

They are being published electronically as part of the web teaching resources for:

Dix, Finlay, Abowd and Beale.
Human–Computer Interaction, Second edition.
Prentice Hall, 1998

http://www.hiraeth.com/books/hci/

Building and Prototyping Groupware

Alan Dix and Devina Ramduny

School of Computing and Mathematics
University of Huddersfield
Queensgate
Huddersfield, HD1 3DH
email: alan@zeus.hud.ac.uk
scomdr@hud.ac.uk

Tutorial Notes
HCI'95, Huddersfield, 29th August 1995

Agenda

Time	Topic	Page
2.30	Introduction	1
2.40	Software architectures	8
3.10	Toolkits	16
3.25	Difficulties and debugging	31
3.45	Networking Code	37
4.00	*Afternoon tea*	
4.30	Synchronous Applications	50
5.10	Asynchronous Applications	65
5.45	Rapid Prototyping Platforms	73
6.00	*Tutorial Finishes*	

Abstract

There has been a recent explosion of interest in groupware and computer-supported cooperative work, both in research and in commercial systems. This is evidenced by the large number of workshops and conferences and has been recognised in the UK by the DTI/EPSRC CSCW programme. This tutorial aims to give the attendee an introduction to the process and problems of building groupware systems. The implementation parts will consider issues for both synchronous and asynchronous groupware and example of prototypes will be discussed.

The builders of groupware have to face not only the complexities of user interface building, but also issues such as distributed computing and network protocols. This can be a daunting task both for project managers and for the programmer. I hope that this tutorial will help the attendee to set the problems in perspective and avoid some of the worst pitfalls!

The tutorial will consider the choice of software architecture and its impact on system performance. We will also examine some of the difficulties of using standard graphical interface toolkits and some of the alternative groupware toolkits and programming paradigms. Finally, we will consider the low level problems of using networks and building and debugging robust and reliable groupware.

The tutorial should be suitable for both academic and industrial attendees who wish to build (or manage the building) of groupware systems. It is assumed that attendees already have some idea of the different kinds of groupware which are currently available.. The latter parts of the tutorial will assume a reasonable level of programming competence, but the use of HyperCard examples will, I hope, make it accessible to the non-expert.

The HyperCard stacks and the UNIX code used in the examples will be freely available for the attendees own personal use and further study.

Instructor Biography

Alan Dix is Reader in Software Technology at the University of Huddersfield. He is most well known for work on the relationship between formal methods and HCI which was addressed in his first book "Formal Methods for Interactive Systems" as well as in numerous articles. His interest in computer supported cooperative work began through a project which studied design issues surrounding electronic conferencing and other synchronous cooperative systems. Since then many of his publications have been in the area of computer-supported cooperative work, especially semi-formal analysis techniques and architectural issues. He is one of the authors of "Human-Computer Interaction", a major textbook published by Prentice Hall. This book contains two substantial chapters on groupware and CSCW, and the early parts of the tutorial will be based on material from it.

Alan began his career as a mathematician and has a B.A. in Mathematics and a Diploma in Mathematical Statistics from Cambridge University. He worked for several years in agricultural engineering and commercial data processing before returning to academia in 1984 to study for his D.Phil. in Computer Science at York.

His current research interests in HCI and CSCW include issues of real time behaviour, status/event analysis, teleworking and formally based implementation paradigms and architectures.

Devina Ramduny is a research student at the University of Huddersfield. She began her studies for her first degree in Computer Science at the University of Mauritius which was completed at Lancaster University. During this time she produced a prototype awareness application for shared access to UNIX file systems.

Devina's current research is focused on temporal issues in CSCW. This has involved the analysis of the administrative procedures during the running of this conference! The principal goal of her work is the design of software architectures for CSCW which take into account the delays of wide area networks. This work will be exemplified in a toolkit to aid the construction of such systems.

Table of Contents

What is Groupware

Simple classifications

- by time / space matrix
- by function

Computer-mediated communication

direct communication support

e.g., email, bulletin boards, video-conferences

Meeting and decision support systems

establishing shared understanding

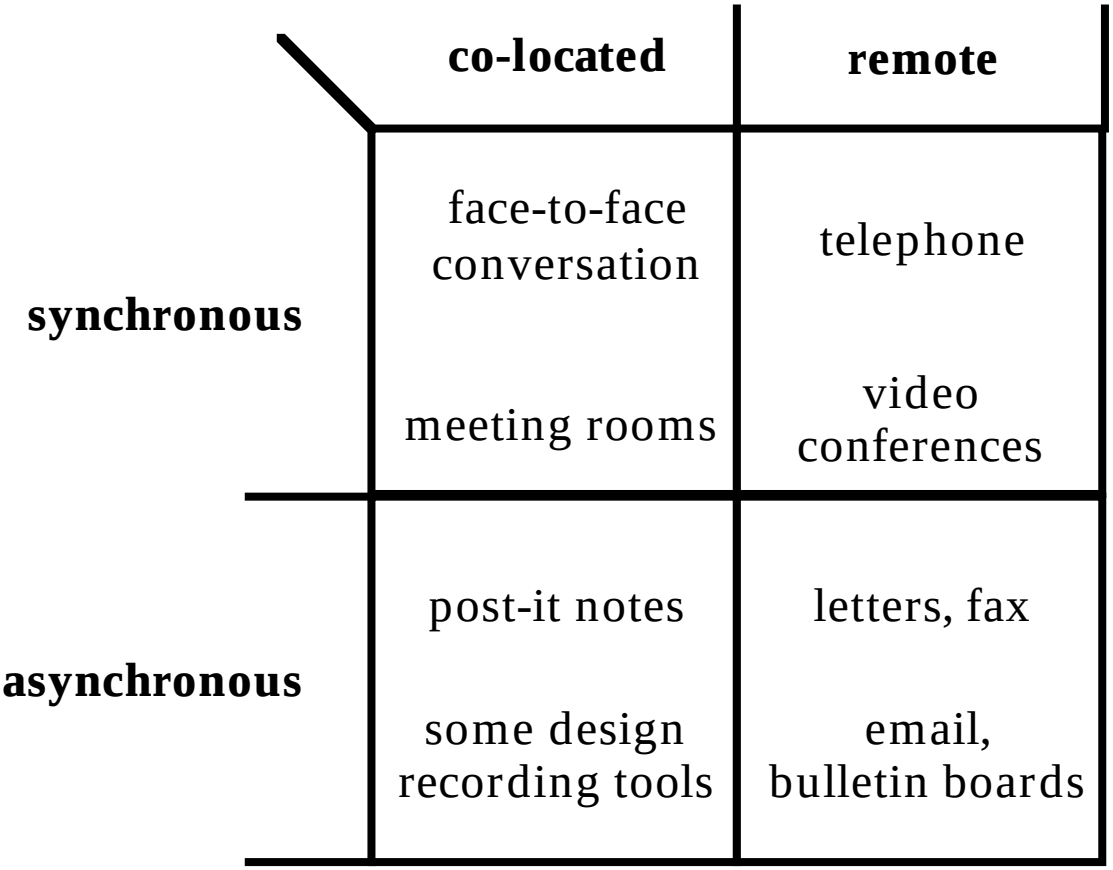
e.g., meeting rooms, design tools

Shared applications and artefacts

interacting with shared work objects

e.g., shared editing, diaries, databases

Time–space matrix



Architectures for Groupware

Groupware:

what

- user interface
- shared data
- control – locking, floor control

where

- distributed on different machines
- linked by a network

Software architecture:

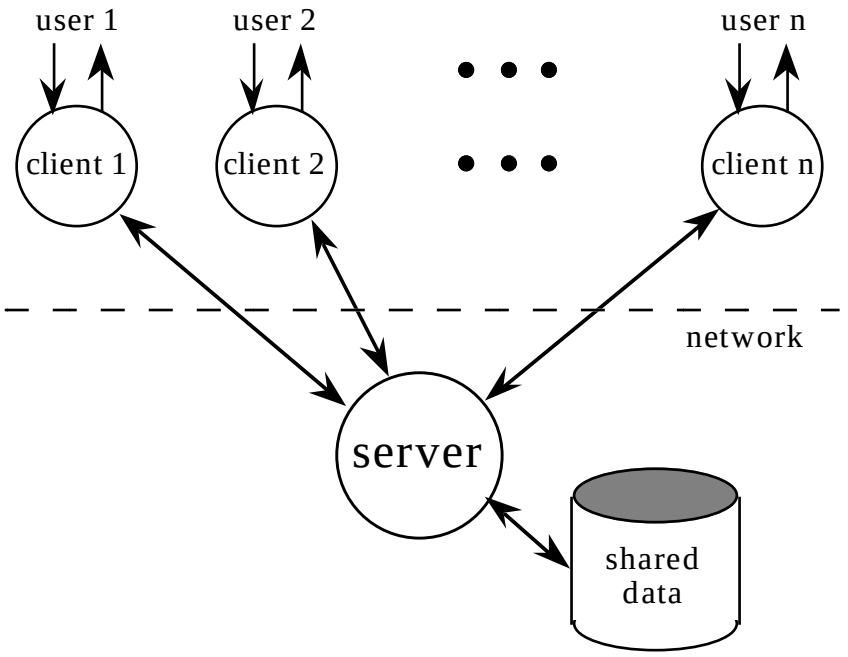
says what is put where

Choice effects:

- feedback and feedthrough
- network traffic
- complexity of implementation

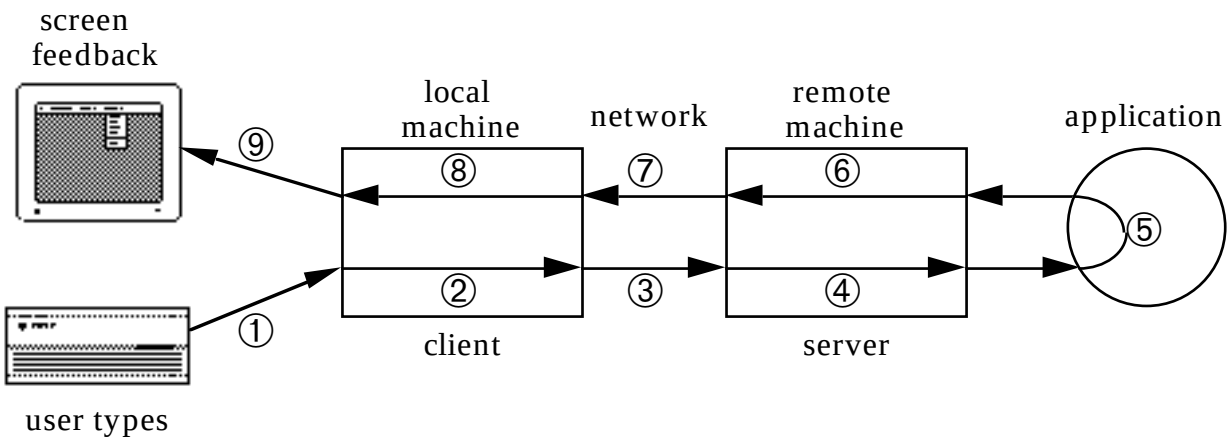
Centralised architecture

- single copy of application and data
  only user interface at each client
- client-server — simplest case
  N.B. opposite of X client/server



- master-slave — special case
  server merged with one client

Feedback delays



Round trip for remote application

at least

2 network message

\+ 4 context switches

in addition ...

- protocols
- network delays depend on load
- server may be multi-processing

Replicated architecture

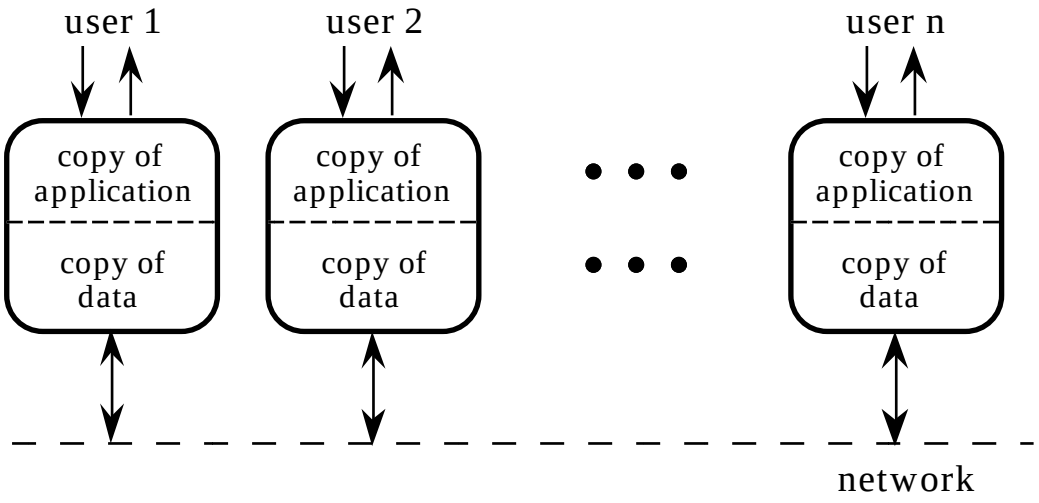
- also called peer-peer

- copy on each workstation

  special algorithms to keep data consistent

  ✔ local feedback

  ✘ race conditions



Hybrid approach

Each extreme has problems:

Centralised

- poor feedback
- bottleneck
- heavy network traffic

Replicated

- maintaining consistency
- new members joining

So, use ‘half-way’ architectures

- local copy of application
- central database
- local cache of data for fast feedback
- centralised support

  locking – explicit or implicit

  update propagation

Various balances possible

Feedthrough and chunking

Need to inform other clients of changes

Few networks allow broadcasts

n participants ⇒ n–1 network messages

Solution!

- increase granularity
- decrease frequency

  e.g., line-by-line rather than character update

But ...

poor feedthrough ⇒ loss of shared context

Trade-off: timeliness vs. network traffic

- can vary chunking between participants

  e.g., shared editing, other channels
- animation of changes

Shared window architecture

- non-collaboration aware application

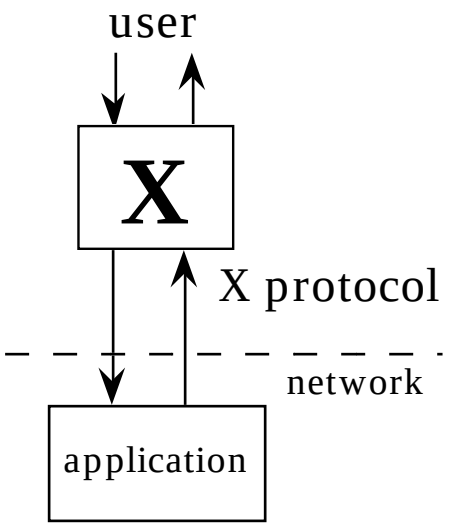
  $\Rightarrow$ one copy of application

  $\therefore$ client / server approach

  corresponding feedback / feedthrough problems

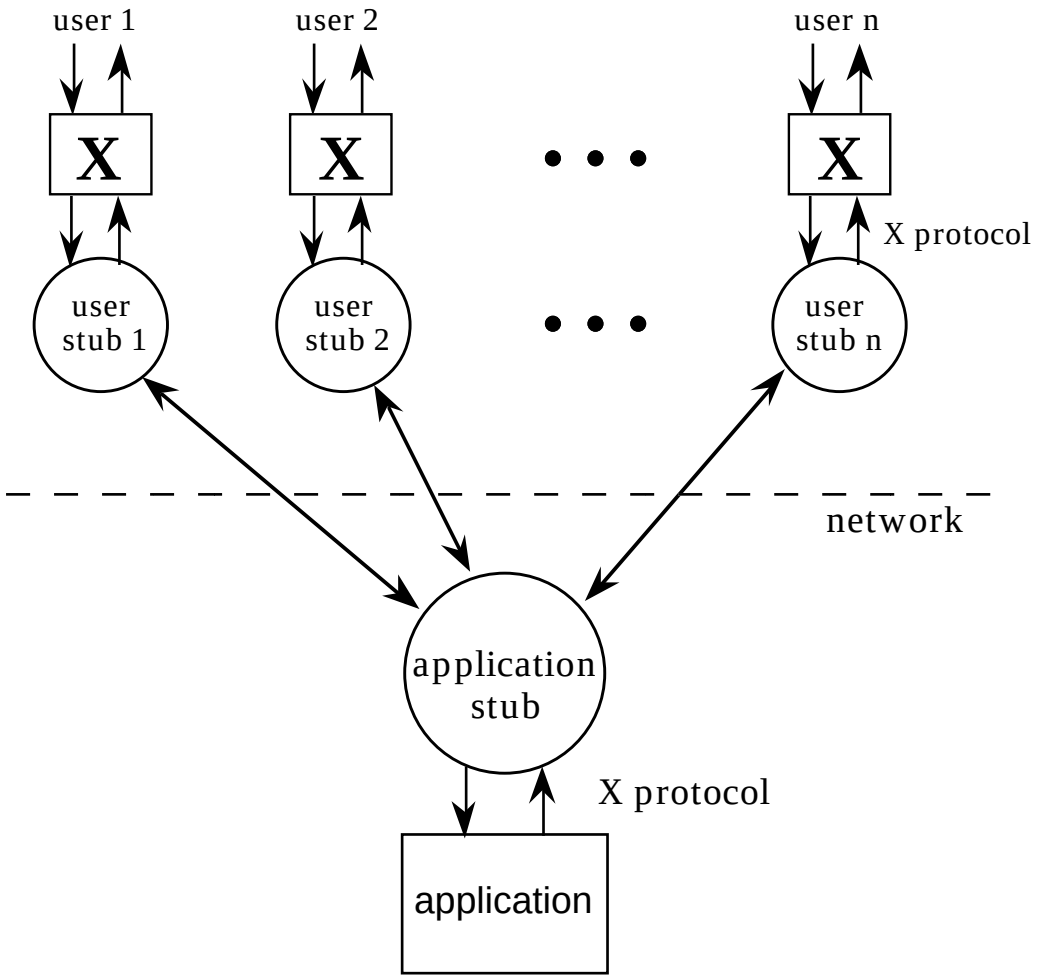
- no 'functionality' needed – all in the application but must handle floor control

- easy(ish) with X as it is already network based



Shared X

- merge user generated X Events
- copy application X lib requests



Toolkits

Why not use standard components?

- graphical interface toolkits
- shared network databases

NOT suitable for synchronous groupware

So, need special purpose

Graphical toolkits

Designed for *single* user interaction

many hidden assumptions

Problems include:

- pre-emptive widgets

  e.g., pop-up menus

  toolkit takes control during interaction.

  ⇒ cannot respond to other user's events

- over-packaged text

  single cursor, poor view control

  either:

  very limited small scale group interaction

  or

  resort to bitmap level graphics

Notification based toolkits help

local and remote events treated uniformly

Shared databases

Traditional approach:

- transparency

  you can't tell where something is
- isolation

  you think you're the only user
- locking

  preventing interference

BUT ... co-operation requires

- awareness

  knowing who else is around
- feedthrough

  seeing what they're doing

Standard databases not suitable

e.g., do not support notification when data changes

can be used for some asynchronous applications

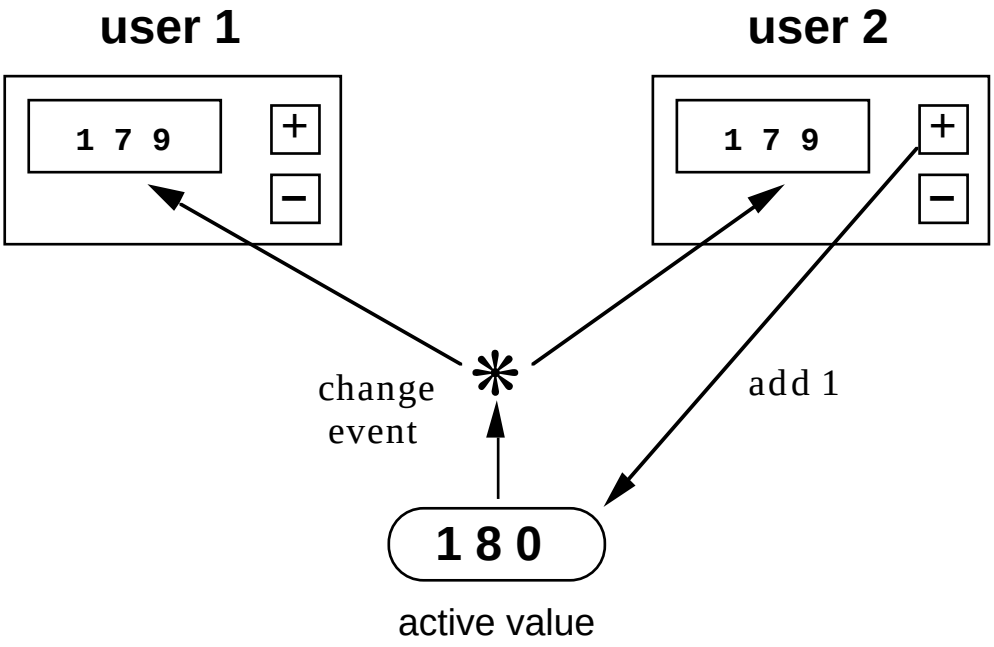
Groupware Toolkits

How to view and update shared objects?

Some solutions:

- Active values
  e.g., SUITE
- Constraints
  e.g., Rendezvous
- Distributed object stores
  e.g., MEAD
- Triggers

Active values

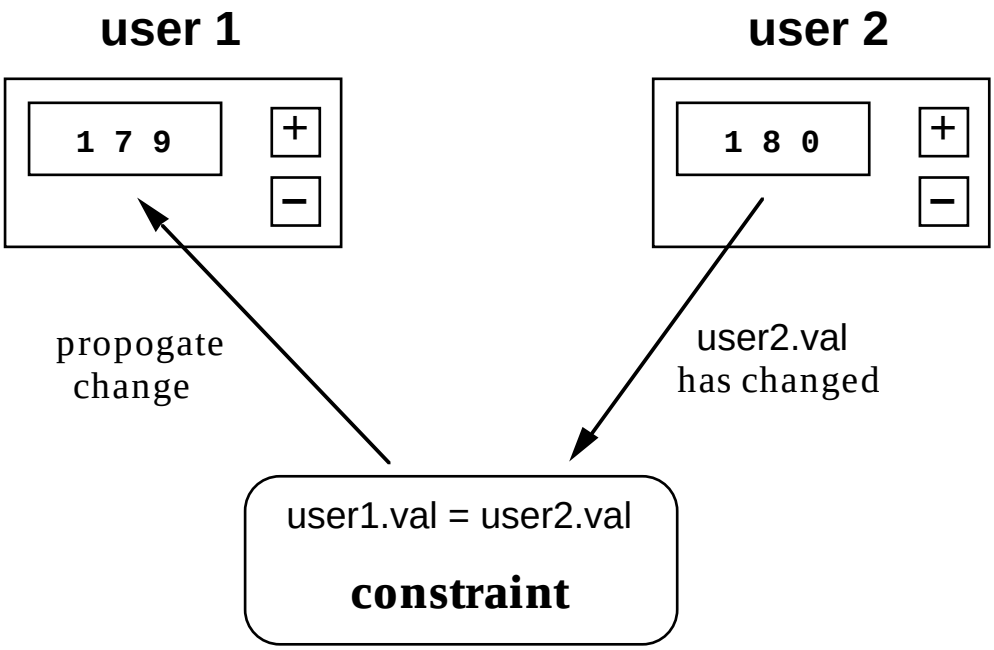


When active value is changed all interested interface elements are informed.

SUITE also supports

- variable update granularity
- predefined widgets

Constraints



Two types of constraint:

one way — one dependent value
two way — no ‘master’ object

Problems of predictability and control

Distributed object stores

- General purpose
  - distributed Smalltalk
  - some OO databases
- Groupware specific
  - MEAD – shared information displays

May support:

- network transparency
  - not necessarily a good idea!
- object migration
  - shifting locus of update
- object replication
  - improved feedback

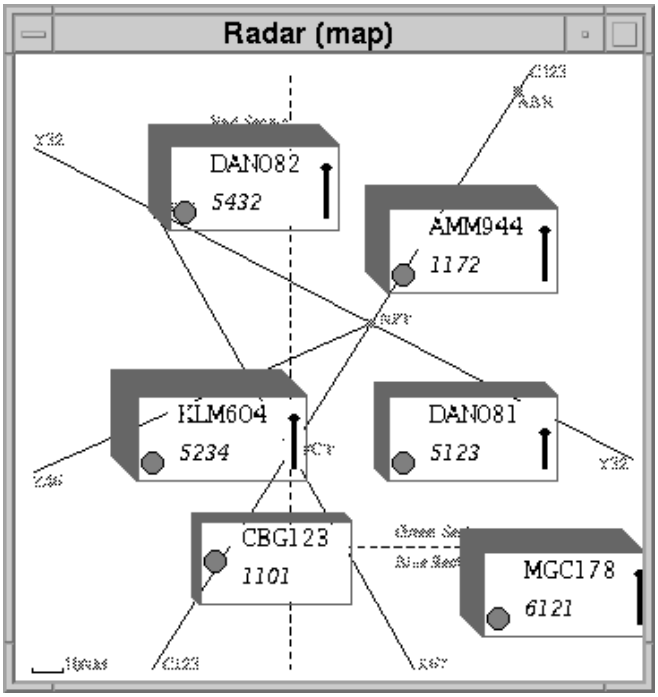
MEAD

Developed at:

Univ. of Lancaster by Richard Bentley

Domain:

Air Traffic Control (ATC) displays

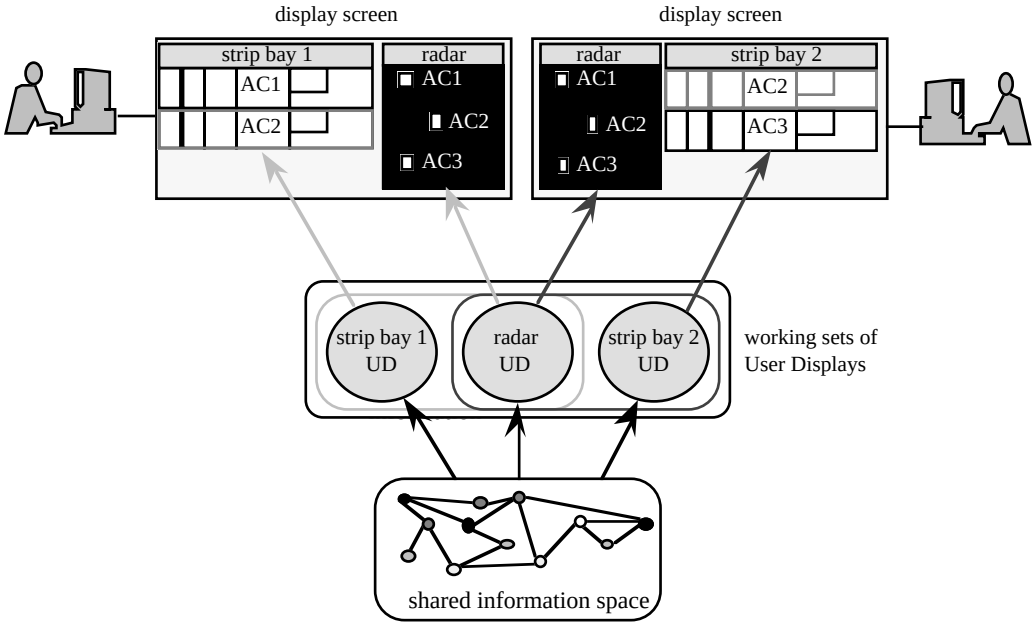


MEAD – 2

Two levels of shared objects:

- individual entities
  e.g., single aircraft

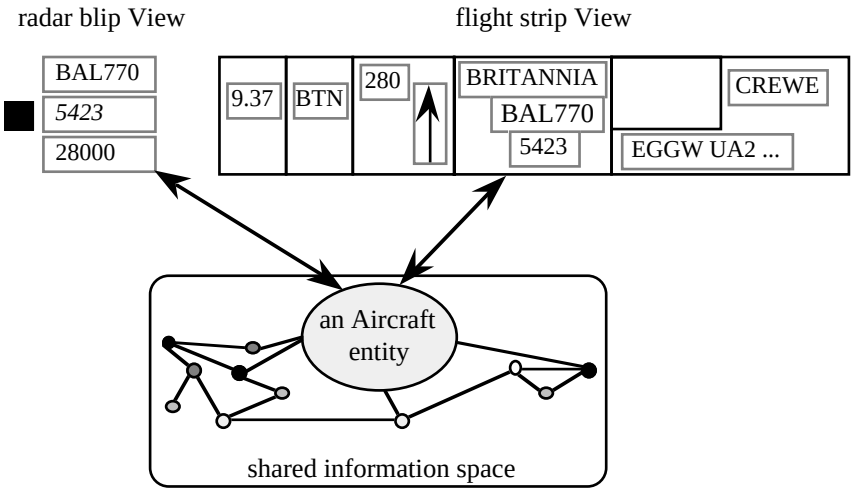
- collections – 'User Displays'
  e.g., radar display



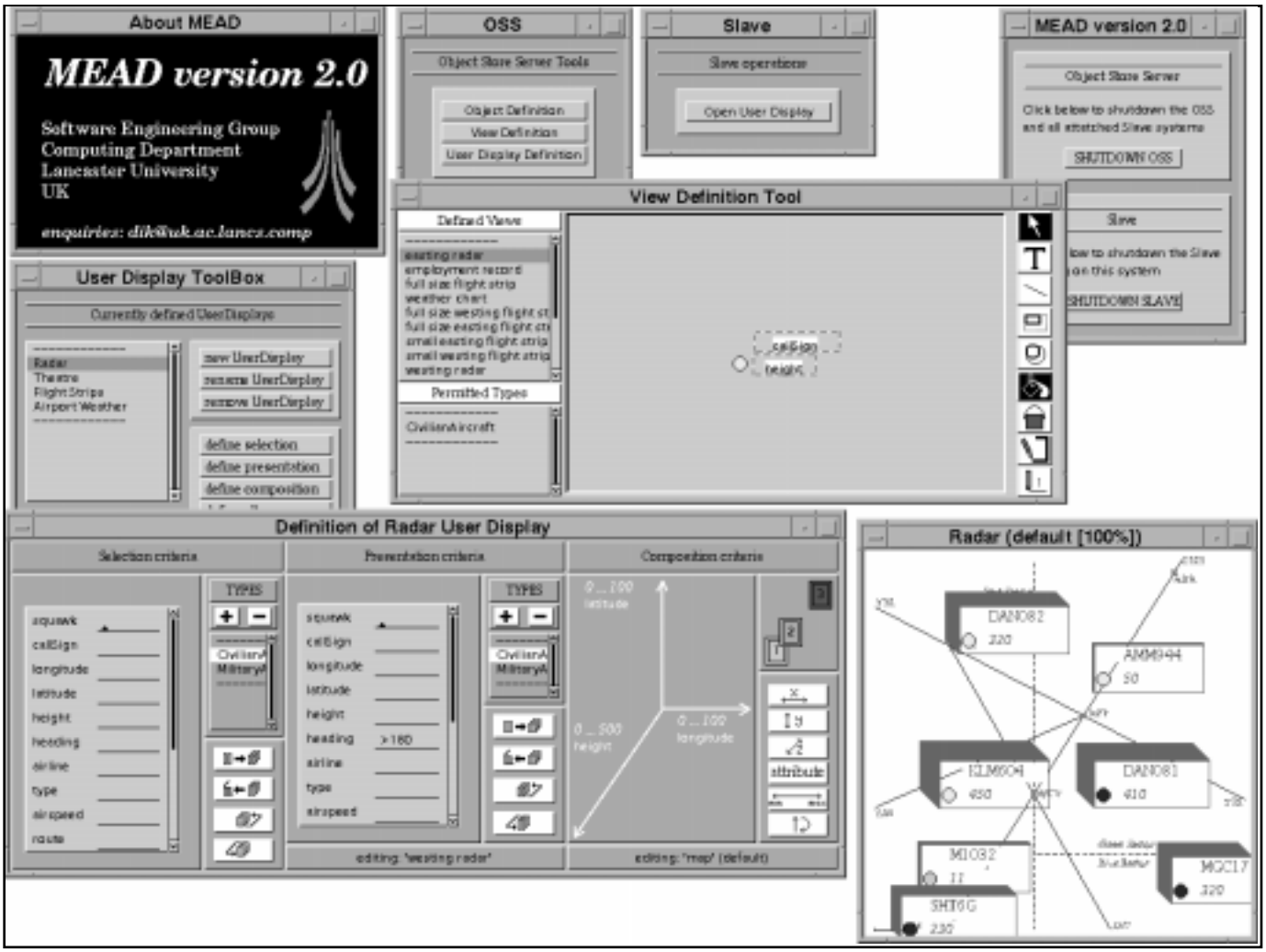
MEAD – 3

Other features:

- transparent object replication
- development environment
- multiple views of same objects

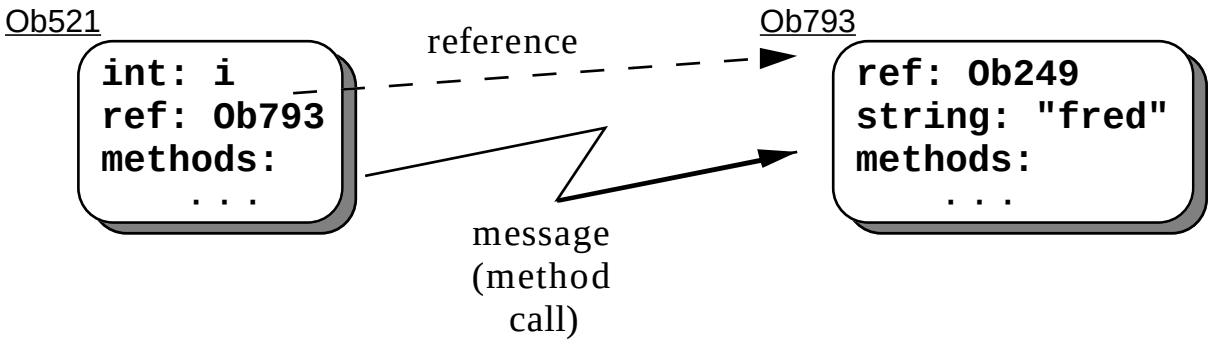


MEAD Development Environment

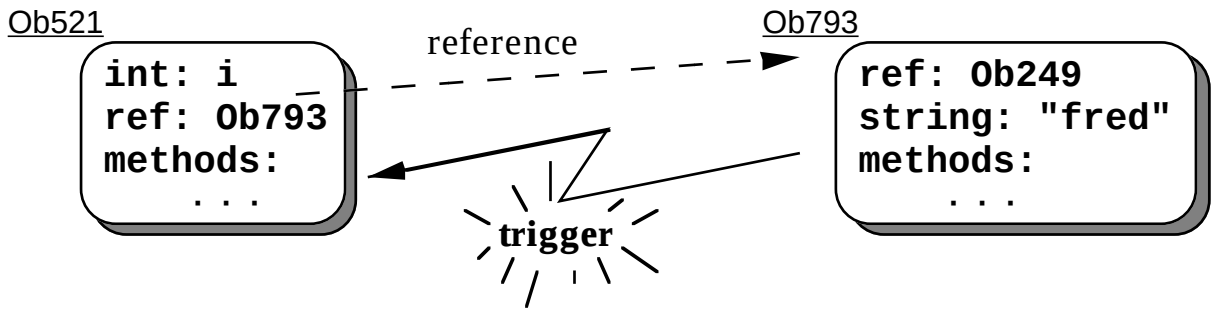


Triggers

Normal message passing



Trigger — opposite direction



Triggers – 2

Lower level mechanism

Can implement:

- Active Values

  trigger is ‘change event’

- Constraints

  constraint object listens for
  trigger on constrained values

More control ...

but, slightly more work

Good paradigm for packaging

groupware widgets

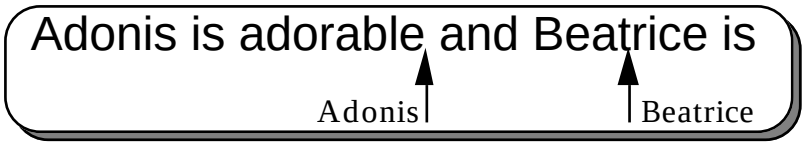
Shared text

The problem – simultaneous update

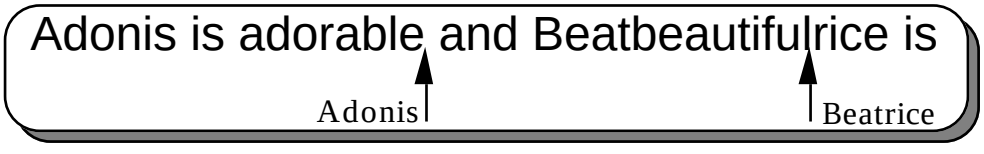
At the same time

- Adonis types "adorable"
- Beatrice types "beautiful"

But Adonis' input is processed first ...



Followed by Beatrice's input



Shared text – 2

Toolkits

- don't do it (surprise!)
- at best small fields
- no simultaneous update

Specialist shared editors

either lock areas of text

section, sentence, even cursor

or complex algorithms

Grove (Ellis and Gibbs)

- translates operations

after insert("adorable",10)

insert("beautiful",27) → insert("beautiful",35)

Dynamic pointers

- translates positions

insert("beautiful",*p*): *p*@27→ *p*@35

Programming Groupware

It is hard

- interface building + networking
- HCI people know about the former!!!

$\Rightarrow$ concentrate on networking

But it has to be good

- crash in single-user interface

  — one sad user
- crash in groupware

  — disaster!

$\Rightarrow$ concentrate on networking

Causes of failure

① hardware failures
② programming errors
③ unforeseen sequences of events
④ system does not scale

Large number of components

⇒ ① more frequent

Complexity of algorithms

⇒ ② more likely

Interleaving and delays

⇒ ③ difficult to debug

Limited testing conditions

⇒ ④ unexercised

Survival

Network or server failure

standard solutions

Client fails — three **R**s for server

- **robust**

  server should survive

  - never wait for response from client
  - non-blocking network I/O

- **reconfigure**

  detect and respond to failure

  - timeout or failure of I/O operations
  - reset internal data structures
  - inform other clients

- **resynchronise**

  catch up when client restarts

  - similar to new participant
  - N.B. client may not know (network)

Software faults

Defensive programming

- inconsistent client / server data structures

Use simple algorithms

- fixed sized structures – but check bounds!
- may conflict with scaleability – document

Verify

- close hand checks
- for production code – formal methods

Unforeseen sequences of events

- deadlock – never use blocking I/O
- never assume particular orders of events
- back-to-back messages

  network packet ≠ logical message

Debugging and testing

- logging – to reproduce failure
- random data – at interface or network
- undergraduates!!!

Network communications

Communication can be:

- Connectionless
  - ❍ address every message
    - ❋ like letters

    e.g., AppleEvents

- Connection based
  - ❍ use address to establish a fixed link
  - ❍ send each message using the link
    - ❋ like telephone

    e.g., UNIX sockets

N.B. both need an address

⇒ some sort of system address book

or, publicly known addresses

Network communications – 2

Other issues:

- Reliability
  - Do all messages arrive?
  - Do they arrive in the right order?

- Buffering
  - effects responsiveness
  - hides potential deadlock

- Messages or byte-stream
  - sent:
    - write 1 (len=26): “abcde....vwxyz”
    - write 2 (len=10): “0123456789”
  - received:
    - read 1 (len=20): “abcde....qrst”
    - read 2 (len=16): “uvwxyz012...89”

  $\Rightarrow$ fixed length messages or prefix with length

Real Code

① AppleEvents
- more later

② UNIX sockets
- two person ‘chat’
- strict turntaking
- simple-server.c & simple-client.c

③ UNIX sockets again
- multiparty conference
- free turntaking
- notification based
- server.c, client.c & libraries

AppleEvents

Addressed by:

zone name : *machine name* : *program name*

e.g., *:Alans'Duo:HyperCard

Address found using 'Chooser' style dialogue box:

```
answer the program
put it in myAddress
```

Sending

Connectionless

⇒ each message needs address

Syntax:

`send` *aMessage* `to program` *anAddress*

Receipt

uses normal HyperTalk event handling

```
on myEvent param1 param2
    do something
end myEvent
```

AppleEvents – 2

Sender waits for a reply – blocking

sender

```
send "tellUser hello"
¬
   to program anAddress
```

sender waits

recipient

```
on tellUser theWords
   answer theWords
   reply "GotIt"
end myEvent
```

sender continues

```
if the result is
"GotIt"
then   it was OK
else   something wrong
```

Also non-blocking form of send

sender

```
send "tellUser hello" ¬
   to program anAddress without
reply
```

sender continues immediately

UNIX sockets

Addressed by:

Internet machine address + port number

e.g.: `zeus@hud.ac.uk` port no.: `1573`

Connection based

⇒ more complex initialisation

① server establishes port number
② client 'finds out' what it is!
③ client requests connection to port
④ server accepts connection
⑤ client and server can talk

Plus ... a lot of C at each stage

Server Code

establish port

```
port_fd = inet_socket(host,port)
    /*  only done once  */
    /*  host must be this machine  */
```

wait for client to connect

```
client_fd = sock_accept(port_fd)
    /*  repeated for multiple clients  */
```

then talk to client

```
for(;;) {
    /*  wait for client's message  */
    len =
read(client_fd,buff,buf_len);
    buff[len] = '\0';
    printf("client says: %s\n",buff);
    /*  now it's our turn  */
    printf("speak: ");
    gets(buff);
    write(client_fd,buff,strlen(buff);
    }
```

N.B. strict turn taking

Client Code

request connection to server

```
server_fd = inet_connect(host,port)
    /*  waits for server to accept      */
    /*  returns NULL on failure         */
    /*  host is server's machine        */
```

then talk to server

```
for(;;) {
    /*  our turn first   */
    printf("speak: ");
    gets(buff);
    write(server_fd,buff,strlen(buff);
    /*  wait for server's message   */
    len =
read(server_fd,buff,buf_len);
    buff[len] = '\0';
    printf("server says: %s\n",buff);
    }
```

N.B. ① opposite turn order

② error checking needed

③ using simplified calls

More complex turn taking

In a full client server system ...

the server must

① monitor requests for connections
② listen for client messages
③ possibly have console control

the client must

① respond to user events
② listen for server messages

In short ...

... several things happen at once

Solutions

- Polling
  - bad for multiprocessing
- Add to event stream
  - network messages – just another event
  - e.g., AppleEvents
- UNIX select
  - rather complex!
    - (see implementation of inform.c)
- Notification based programs
  - callbacks
  - e.g., Windows and some X toolkits

Notification based server – 1

① **Initialisation**

```
main(...) {
        /*   establish port    */
    pd = inet_socket(host,port)
        /*   set-up callback for port    */
    inform_input(pd,accept_client,NULL)
;
        /*   give control to notifier    */
    inform_loop();
}
```

② **When client requests connection ...**

... notifier calls `accept_client`

```
accept_client(...) {
        /*   accept client's connection    */
    fd = sock_accept(port_fd);
        /*   record connection details    */
    client_fd[count] = fd;
        /*   set-up callback for client    */
    inform_input(fd,read_client,count);
        /*   keep track of number of clients    */
    count = count+1;
        /*   probably tell other clients also    */
}
```

Notification based server – 2

③ **When client sends message ...**

... notifier calls `read_client`

```
read_client( c_fd, id ) {
        /*  read client's message  */
    len = read(c_fd,buff,buf_len);
        /*  broadcast to other clients  */
    for( c=0; c<client_count; c++) {
        if ( client_fd == c_fd ) {
            /*  special reply for sender  */
        }
        else {
            /*  relay message to other clients
*/
        }
}
```

N.B. step ① performed <u>once</u> at initialisation
steps ② & ③ happen any number of times ...
... in any order

- Client code similar, but without 'accept' stage.

Notification based client – 1

① **Initialisation**

```
main(...) {
        /*   request connection to server    */
    sd = inet_connect(host,port)
        /*   set-up callback for server    */
    inform_input(sd,read_socket,NULL);
        /*   set-up callbacks for interface    */
    ...
        /*   give control to notifier    */
    inform_loop();
}
```

② **When server sends message …**

… notifier calls `read_socket`

```
read_socket( sd, ... ) {
        /*   read server's message    */
    len = read(sd,buff,buf_len);
        /*   process message           */
        /*   probably update interface   */
}
```

Notification based client – 2

③ **When user does something ...**
... notifier calls appropriate callback

```
process_mouse_click( event, ... ) {
        /*   process interface event   */
    ...
        /*   possibly send message to server   */
    if ( event.region == send_button )
{
        write(sd,mess,mess_len);
    }
}
```

Again step ① once at initialisation
steps ② & ③ any number of times in any order

Simple X based server

Simply open lots of displays ...

① open the displays

```
for( i=0; i<Nos_displays; i++ ) {
    d[i] = XOpenDisplay(display_name[i]);
    root = XRootWindow(d[i]);
    win[i] =
XCreateWindow(d[i],root,… );
    }
```

② everytime you read an event

```
XNextEvent(some_display,event);
```

③ you perform the feedback on every display

```
for( i=0; i<Nos_displays; i++ ) {
    XDrawLine( d[i], … )
    }
```

NO client code at all

although in X terms the server is a client anyway!!!

N.B. All displays need to be checked at stage ②

⇒ need select or toolkit callbacks

Synchronous Groupware

Issues

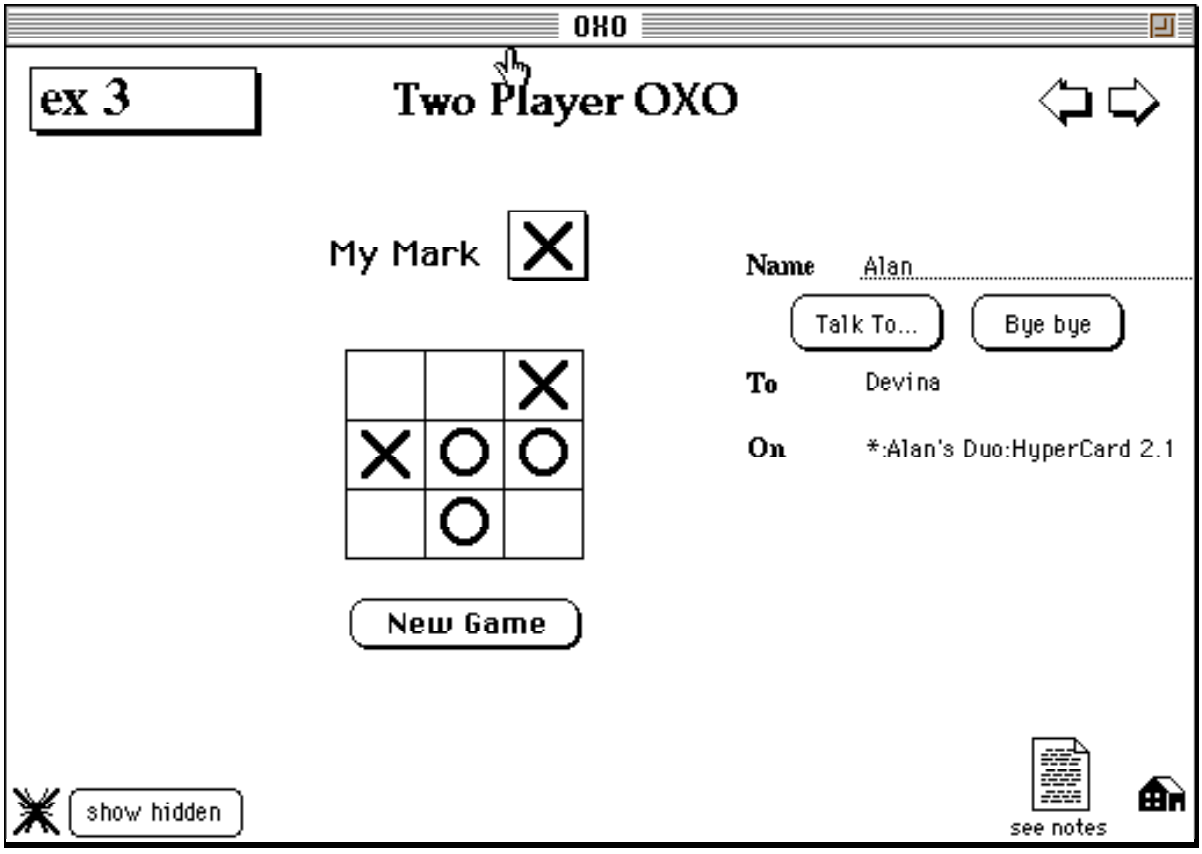
- architecture choice
- feedback
- feedthrough
  - granularity and pace
- consistency
- robustness
  - race conditions etc.

Four examples:

- Noughts and crosses (tic-tac-toe)
- Two-person talk
- Multi-way talk (simple conference)
- Electronic post-it notes

OXO – a two-player game

architecture	peer – peer
feedback	local
feedthrough	each move

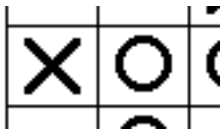


OXO – Phases of play

① establish connection

- either player selects 

② normal play – a player either:

- makes a move 

- selects 

- changes play symbol 

③ termination

- either player selects 

OXO – Structure of program

Typical pattern:

```
respond to user event
        send message to other program
```

```
respond to message
        update  screen etc.
```

For example:

```
on doTalkTo
        find other machine
        send "PleasePlayOxo " to other program  & wait
```

```
on PleasePlayOxo name
        ask user if OK and reply
```

```
on makeMove
        if legal ... update my board
        send "remoteMove" to other
program
```

```
on remoteMove loc
        update board
```

```
on sayByeBye
        confirm with user
        send "ByeBye" to other program
        tidy up board
```

```
on ByeBye
        tell user
        tidy up board
```

OXO – Problems

✘ No game rules

- rely on social protocols
- add more code ...

✘ Deadlock during connection

- more complex code

✘ Race conditions

e.g., both players hit same square

- rely on players turn-taking
- more complex code

✘ Both players need to 'logon'

⇒ need to know passwords etc.

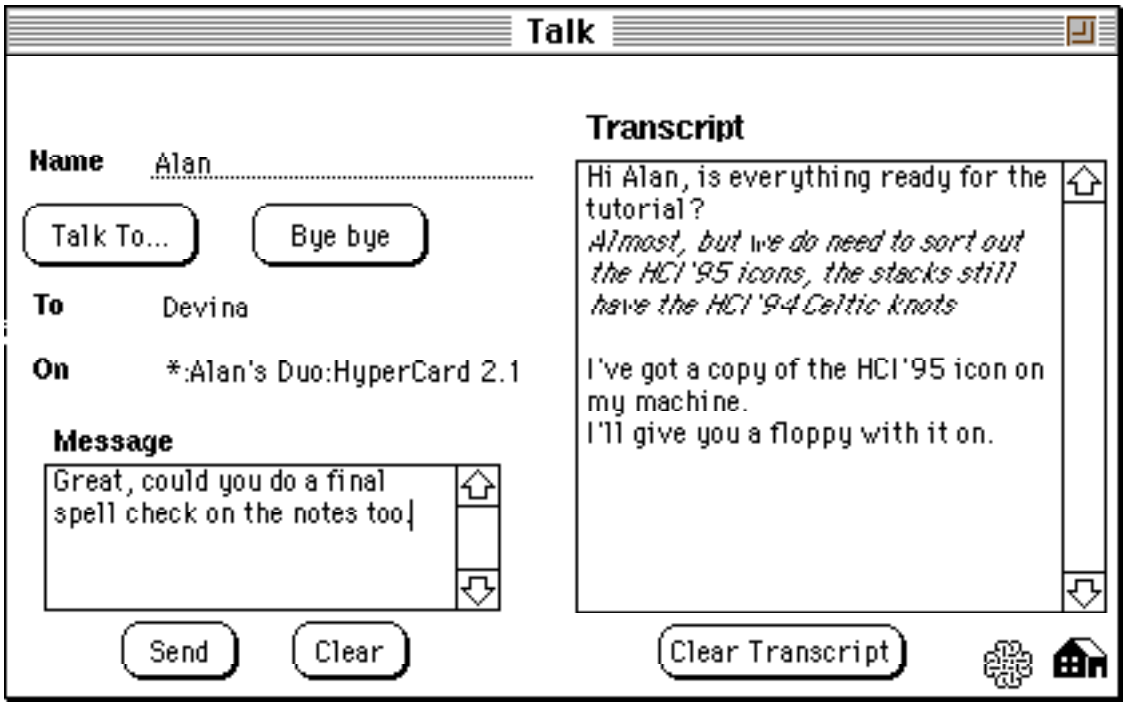
✘ nature of AppleEvents

Two-person Talk

N.B. similar to OXO

architecture	peer – peer
feedback	all local
feedthrough	a ‘message’

✔ allows simultaneous composition



Talk – Problems

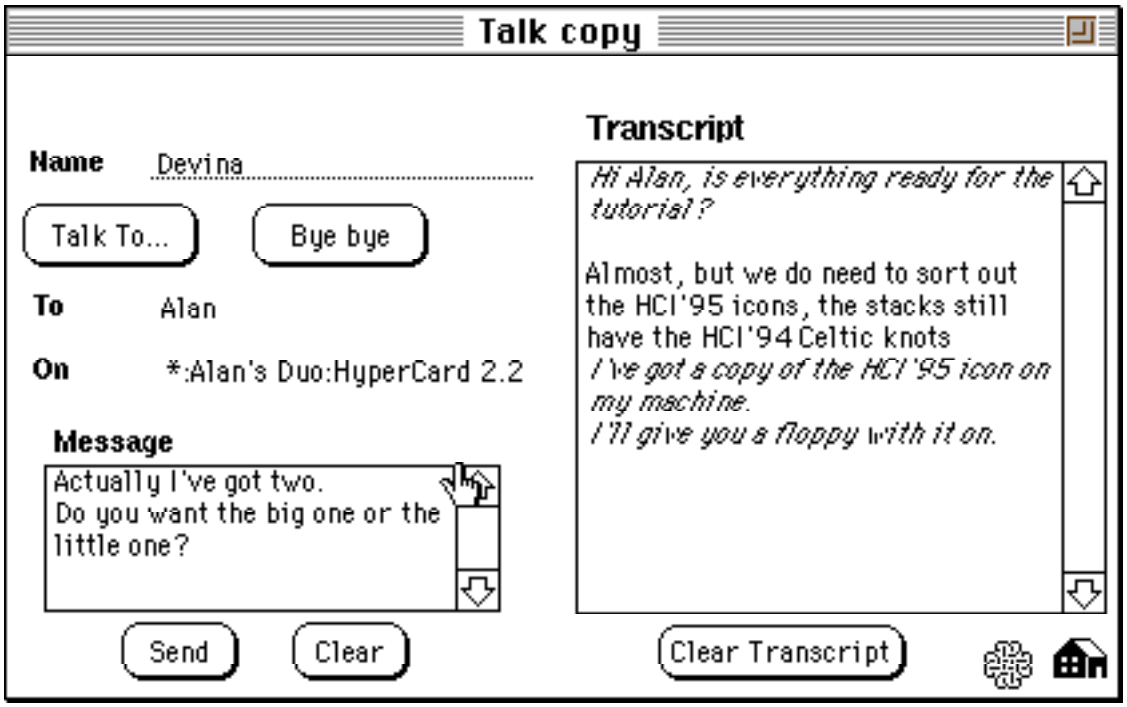
✘ Consistency – race conditions

- simultaneous message sends
  participants see different order (cf. email)
- need some global ordering ...

✘ Text handling

- need to 'pack' text for send

✘ nature of HyperCard 'send'



Multi-party Talk

architecture master – slave

function of master:

① keep track of who is connected

② broadcast messages

③ impose global order

feedback:

- local – remove composed message
- semantic – message appears in transcript

feedthrough:

- granularity – a 'message'
- notification – direct from server

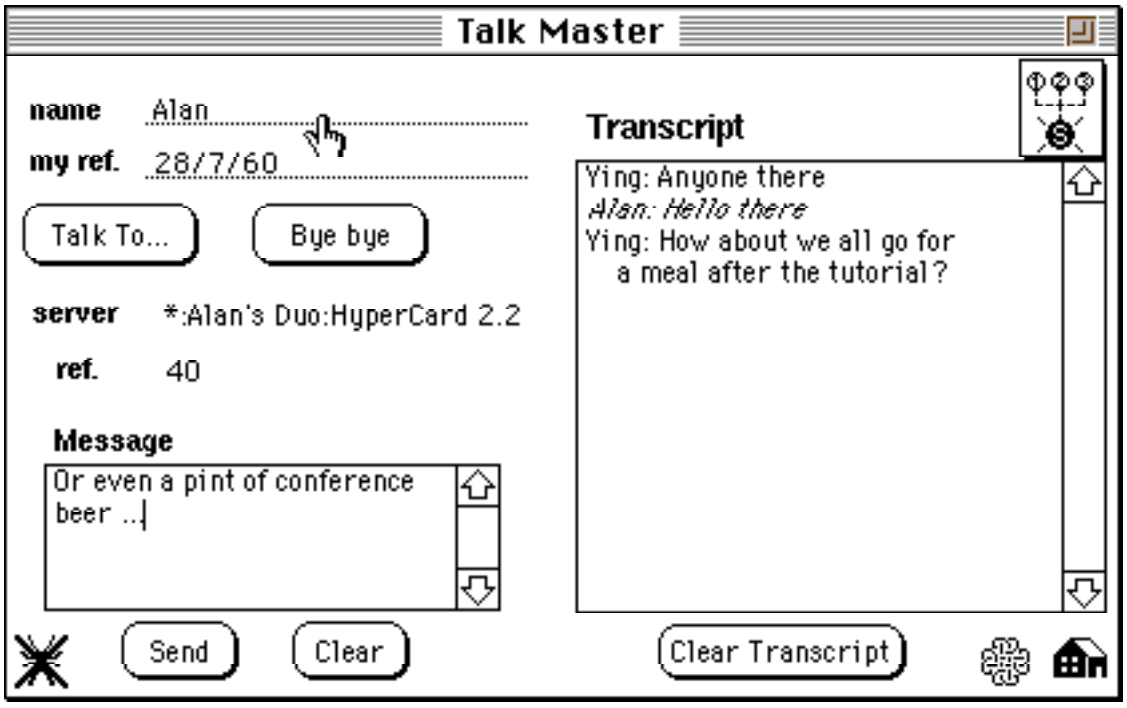
consistency:

✔ maintained because of global order

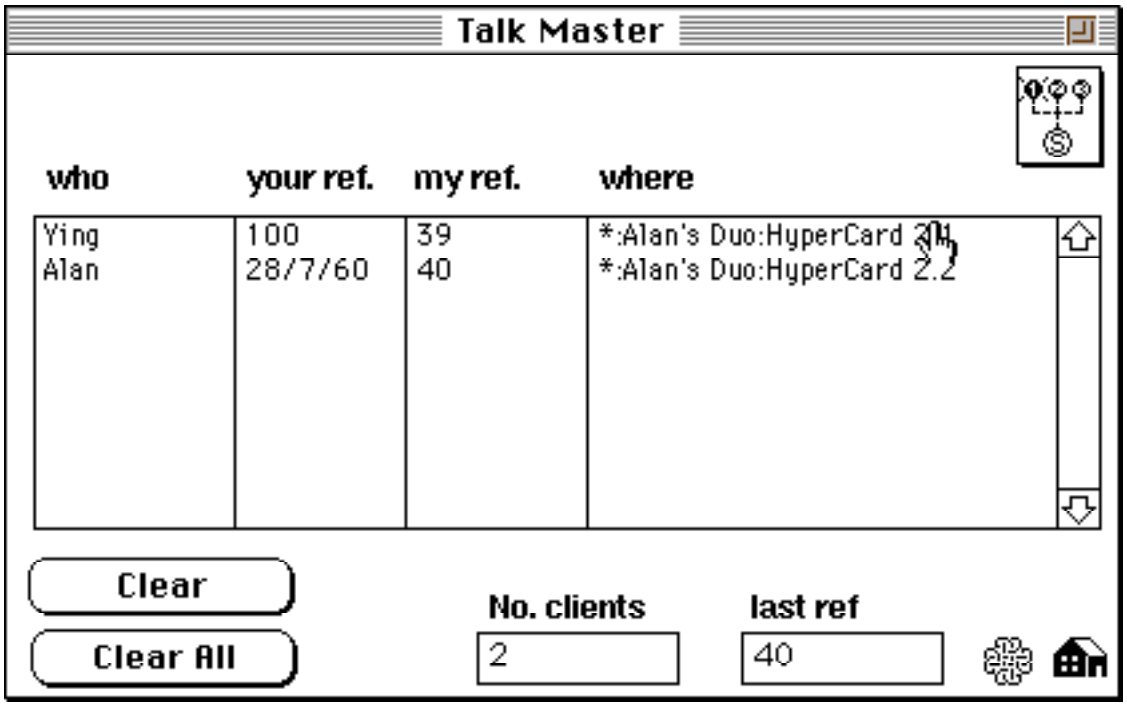
✘ data decentralised – glorified peer-peer

Multi-party Talk – 2

User interface



Master control panel



Multi-party Talk – 3

Debugging and instruction

✔ data structures visible

e.g., master's connection table

✔ optional message tracing

✳ shows all received AppleEvents

Problems

✘ no catching up

- because no golden copy of data

✘ master needs to 'logon'

❋❋ very serious ❋❋

- nature of AppleEvents ...
  ... only fix is to poll for changes

Post-it notes

architecture client – server

function of server:

① keep track of who is connected

② keep central copy of data

③ locking policy

feedback:

- creation – local
- selection – wait for server lock
  - \+ immediate shadowing of note
- editing – local (once selected)

feedthrough:

- granularity – complete action (edit etc.)
- notification – direct from server

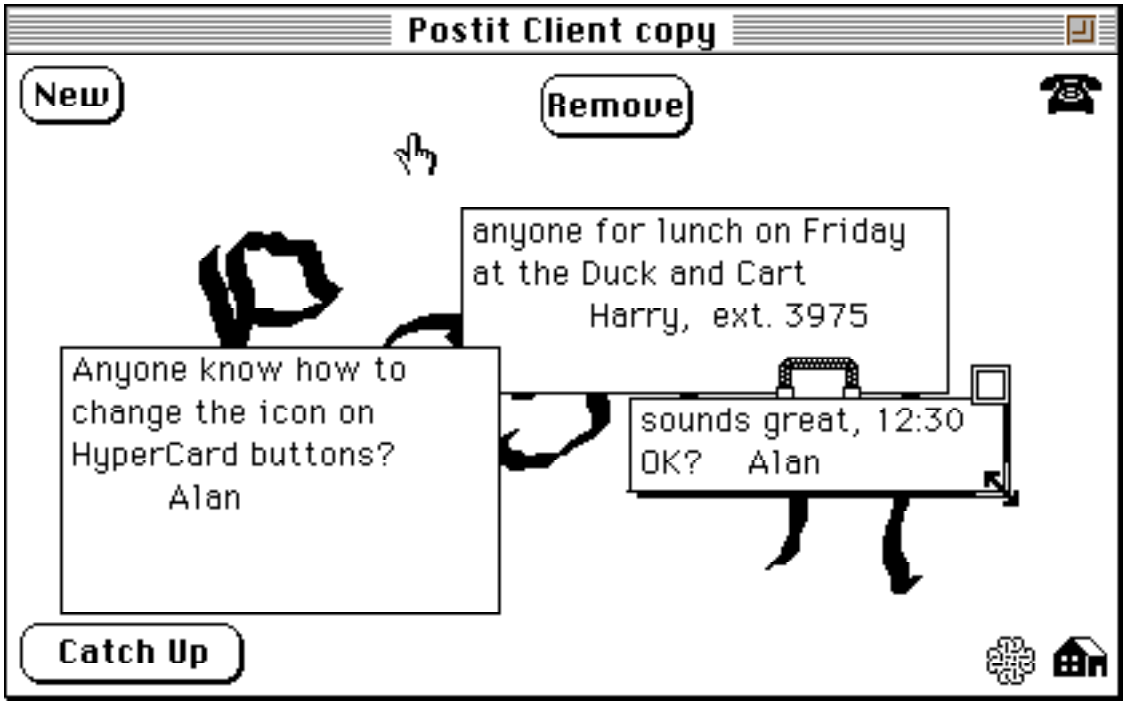
consistency:

✔ maintained by server

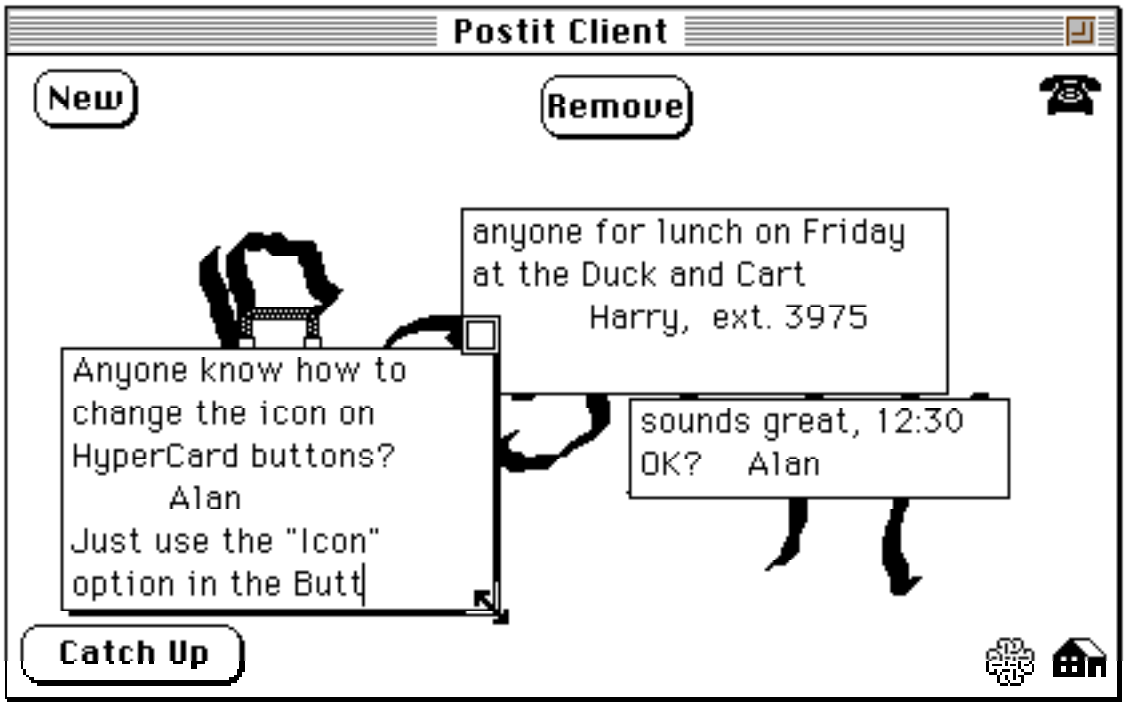
✔ catch up – using central data

Post-it – client screens

Alan editing a new note

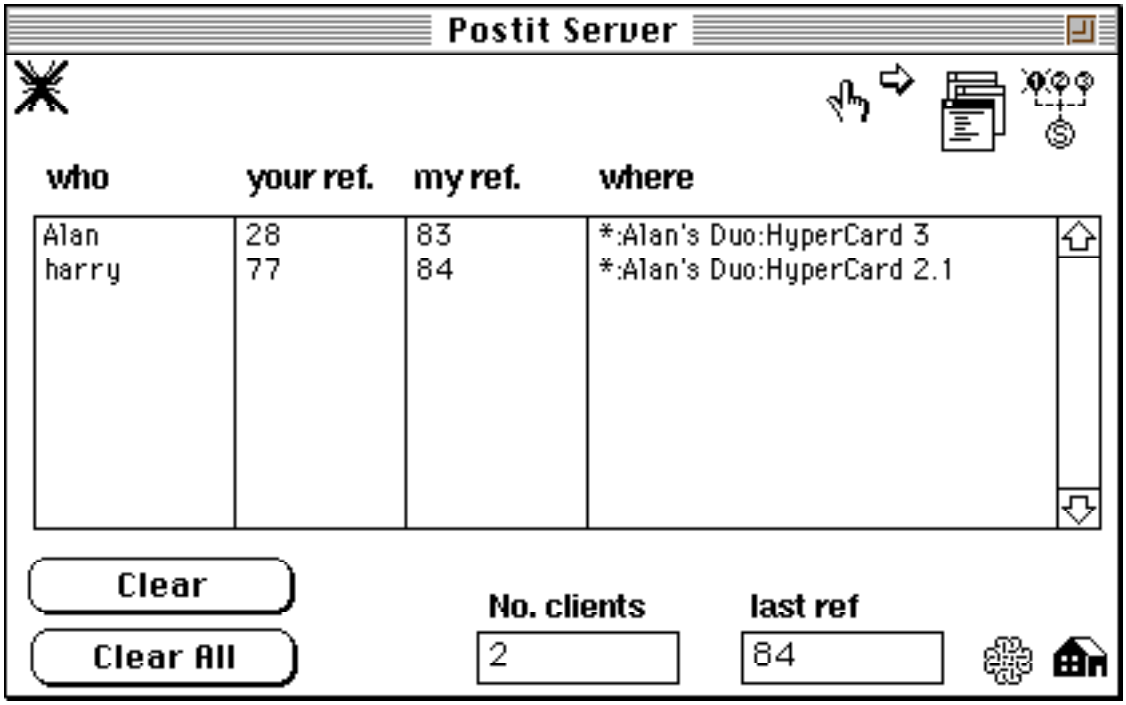


it appears on Harry's screen – he updates another note

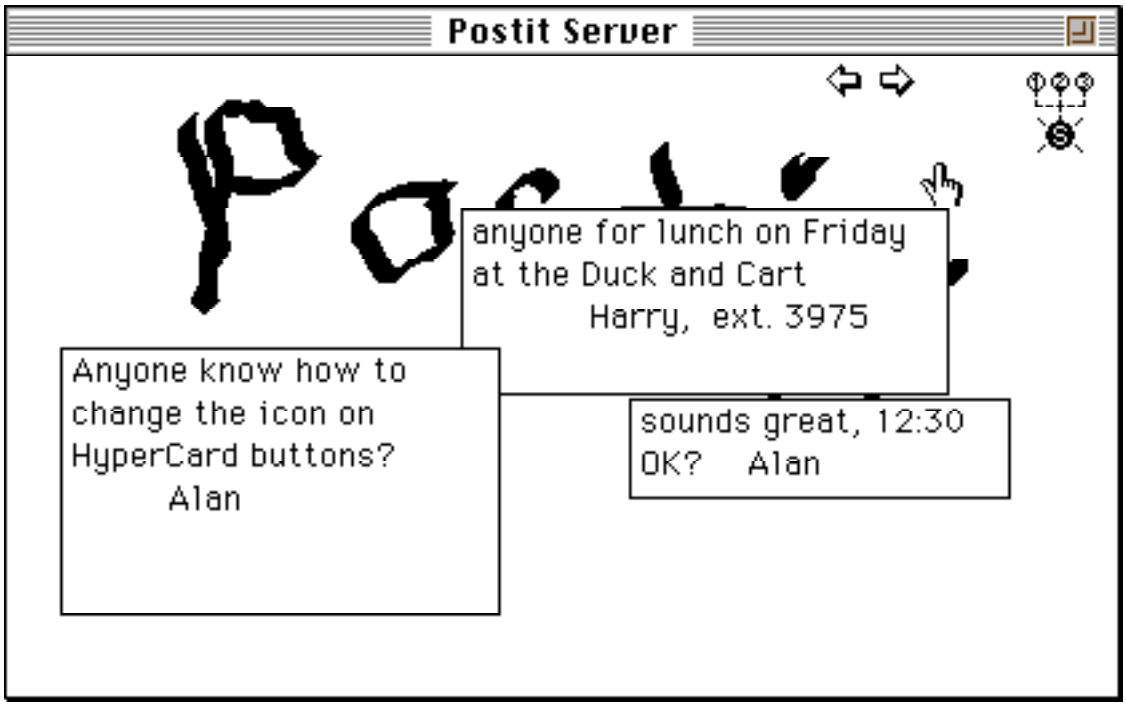


Post-it – server screens

connections – like multi-person talk



the golden copy of the data

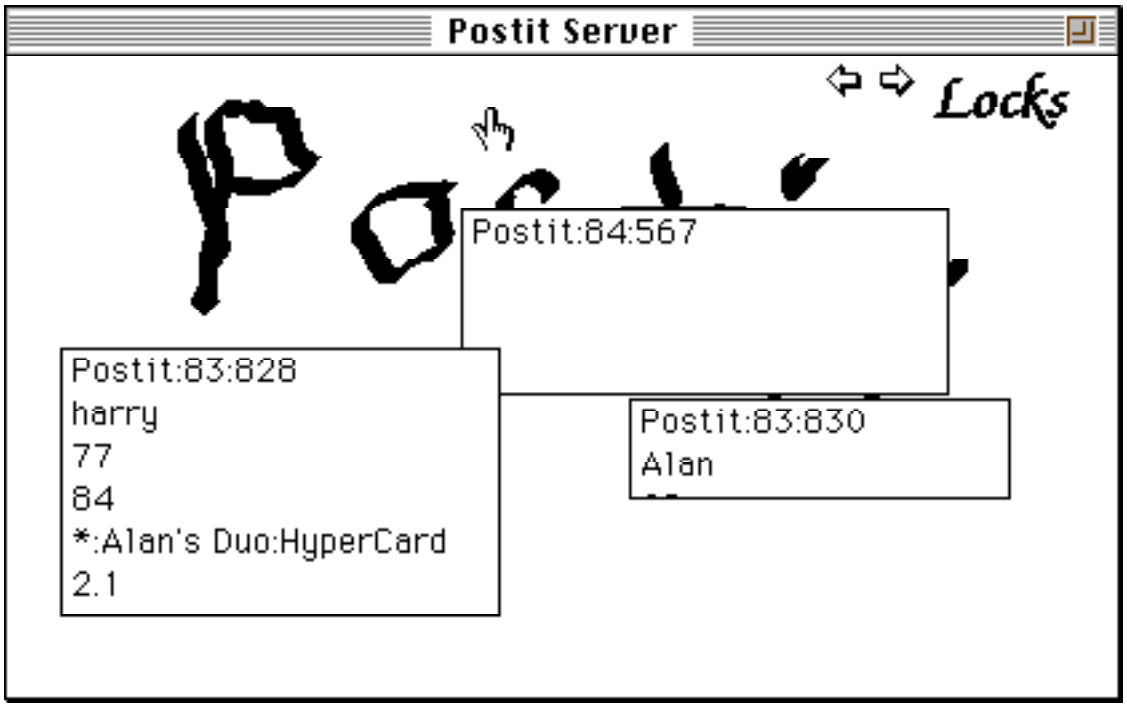


Post-it – server locks

server = connections + data + locks

✔ locks also visible in server stack

- lock fields mirror the real notes
- useful for debugging and instruction



Post-it – problems

- ✘ Speed!!
  - ✘ nature of HyperCard
    field and card creation very slow
- ✘ Lost user selections
  - ✘ HyperTalk behaves 'like a user'!
  - ✔ workarounds possible – more code
- ✘ Post-it card must be visible
  - ? general problem – where to put the script
  - • put more into the stack script
- ✘ Server needs dedicated machine
  - ✘ multi-tasking in Mac/OS is poor
- ✘ Complicated client code
  - ✘ nature of HyperCard
    - • direct manipulation messy
    - • creation of fields hard & ugly
- ✘ Complicated server code
  - • partly HyperCard's limitations
  - + locking etc. is hard

Asynchronous Groupware

Main types

- messaging systems

  email, structured mail, workflow etc.

- shared information

  shared diaries, Liveware, Lotus Notes

Issues

- feedthrough

  when does the user see changes

- consistency

  concurrent update

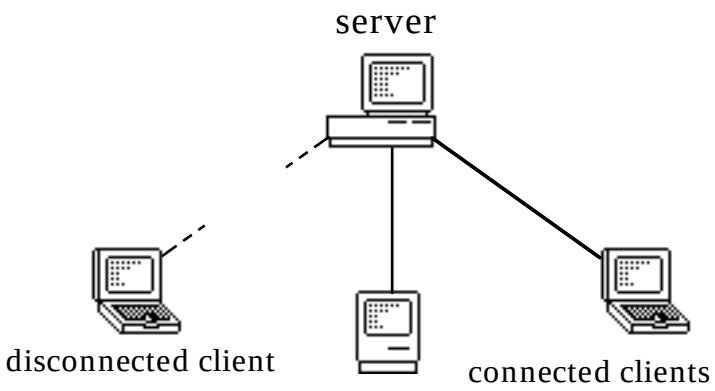
  times and dates

Two examples:

- simple email system
- shared appointment database

Architecture and coding

- both client–server
- user controlled connection



- clients always poll server
  - no explicit server notification
  - avoids server logon problem

- working when disconnected
  - simple email ✘
  - shared database ✔

Simple Mail

- single mail server – no routing

  operates on single AppleTalk anyway

- server functionality:

  ① receives messages from clients

  ② stores messages in stack

  ③ delivers when recipient connected

- tricks

  ❍ names cards by recipient
  (HyperCard can find them easily)

  ❍ single line messages
  (let clients worry!)

- very general

  message passing engine

  could be used by many applications

- client functionality:

  N.B. only operates when connected

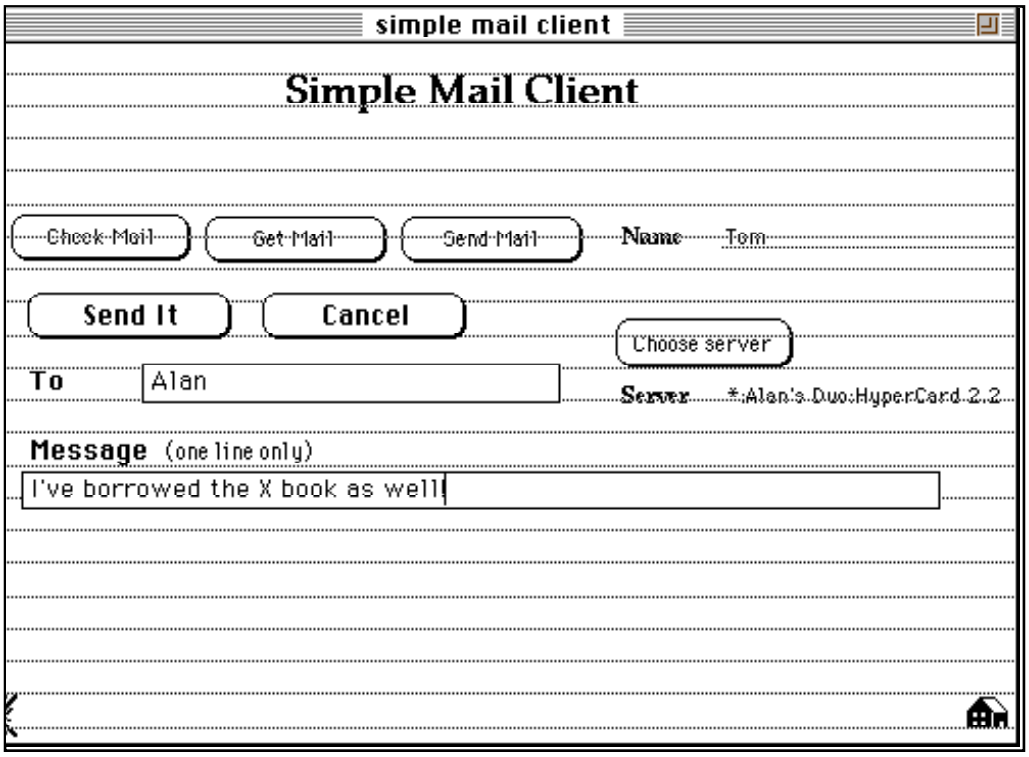
  ① sends messages for user

  ② displays number of waiting messages

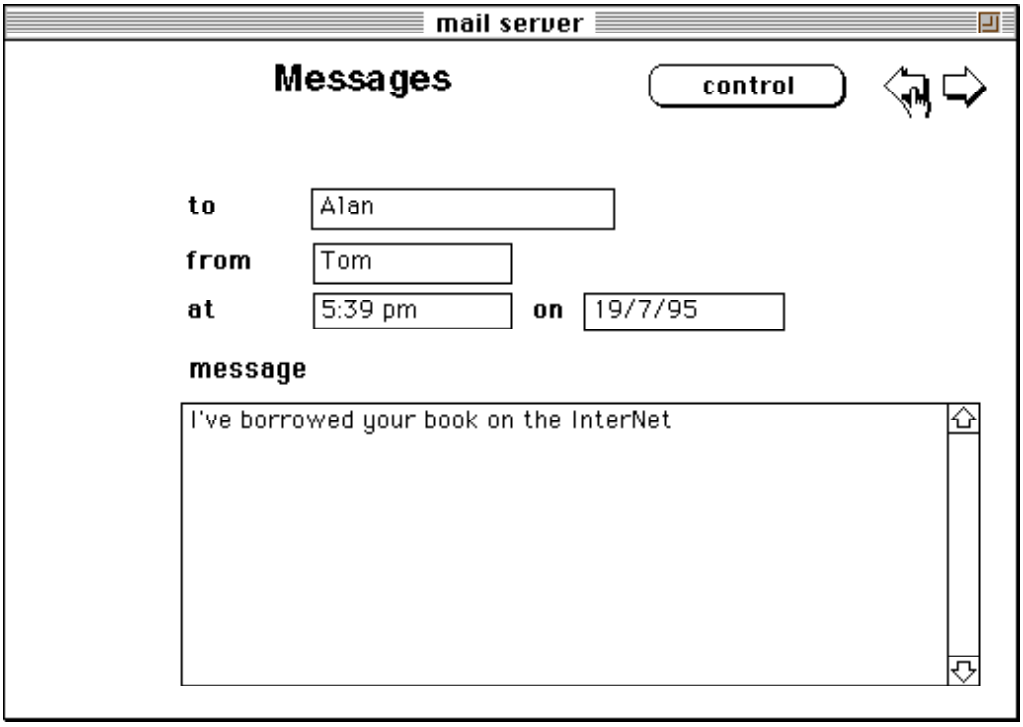
  ③ receives messages for user

Simple Mail – screens

client sends mail



messages stored on server



Shared Database

- disconnected operation
  - clients can update when disconnected
  - explicit resynchronisation
- merging
  - granularity – record
  - most recent record is preserved
  - based on virtual time stamp
- notification of changes
  - for program – on request
  - for user '**new' flag
    (could be used for changes listing)
- server
  - accepts simultaneous clients ...
    ... but care needed in coding
  - can connect as client to other servers
    (allowing peer–peer updates)

Shared database – problems

✘ Slow

- largely due to text packing
  (candidate for XCMD)
- ✔ does have progress indicators

✘ Deletes

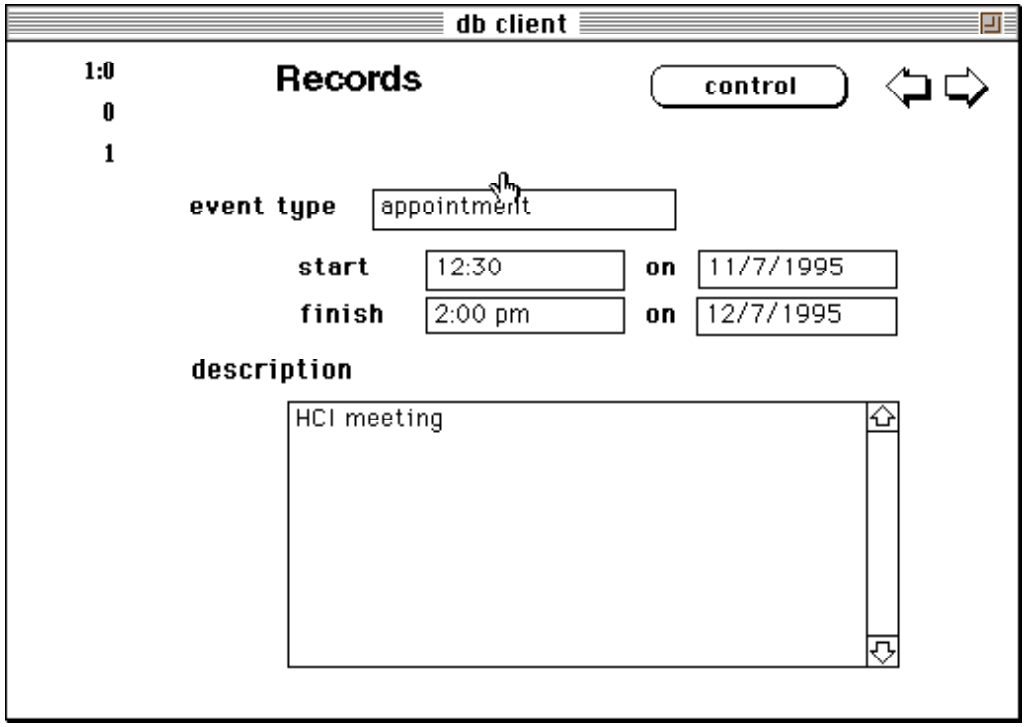
- gradual accumulation of garbage
- ✘ solution difficult

✘ concurrent update of single record

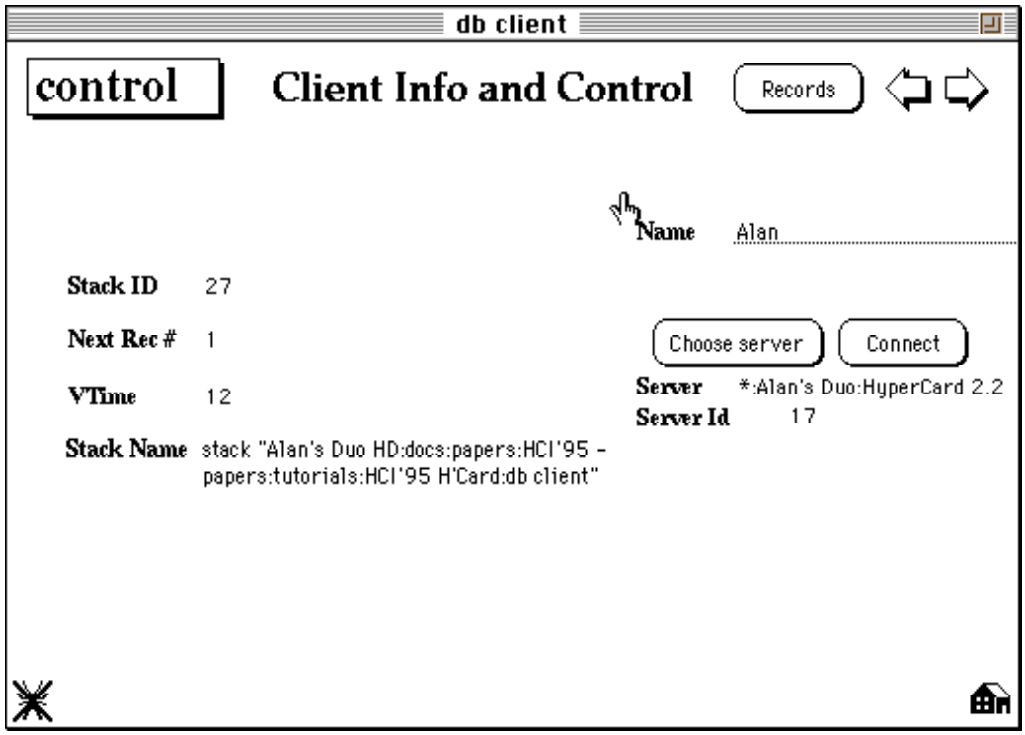
- ✘ **NOT** detected
- possible given data stored

Shared database – screens

user updates records when disconnected



later connects to server to merge updates



Shared database – coding

- when disconnected

  user updates database

- when connected client:

  ① increments virtual time (vtime)

  ② synchronises vtime with server

  ③ sends all changed records

  ④ gets list of changed records
  (and new vtime)

  ⑤ reads changes one by one

? which records are changed
client stores a last sync vtime for each server

? why use 'virtual time'
doesn't rely on accurate clocks

? how to tell if two records are 'the same'
records identified by stack id and counter
(N.B. relies on unique numbering of clients)

Rapid Prototyping Platforms

Really is fast!

Talks + Postit	—	1 week
inc. learning curve		
OXO	—	1 hour
simple mail	—	$\frac{1}{2}$ day
shared database	—	1 day

N.B. experienced hacker !

c.f. conferencer (similar to Talk+Postit)

- developed at York in C/UNIX
- TCP/IP network
- Presenter for graphics

— 2–3 months coding !!!

HyperCard pros & cons

- ✔ interface development easy ...
  - ✘ ... but only if you do what it likes
- HyperTalk acts ‘like a user’
  - ✔ simplifies some programming
  - ✘ ... but odd side effects !
- visible data structures
  - ✔ good for debugging
  - ✘ poor for complex algorithms
- performance
  - ❍ **OK** if you do easy things
  - ✘ but sometimes a real pain
- AppleEvents
  - ✔ simple mechanism
  - ✘ awkward packing problems
  - ✘ ‘logon’ sometimes a problem

Alternative platforms

- TCP/IP
  - ❍ X/UNIX – not really rapid!
  - ❍ HyperCard – real hacking
  - ❍ H'Card UI client + UNIX server
    – good evolutionary path

- PC/Windows
  - ❍ WinSock (TCP/IP) – similar to UNIX
  - ❍ V Basic + Windows for Workgroups
    maildrops easy, but need polling

- Network databases
  - ❍ must poll for changes
  - ✔ use with bespoke groupware server

- Lotus Notes
  - ✔ good for workflow etc.
  - ✘ spreadsheet like macro language

Bibliography

This is not an exhaustive bibliography, merely an entry point into the literature. For a more extensive bibliography, see also the electronic CSCW bibliography maintained at the Department of Computer Science, University of Calgary.

General Introduction

- A. Dix, J. Finlay, G. Abowd and R. Beale, *Human–Computer Interaction*, Chapters 13 and14, Prentice Hall International, UK, 1994.

Journals

There are now two journals specialising in CSCW:

- *Computer Supported Cooperative Work,* Kluwer.
- *Collaborative Computing*, Chapman and Hall.

The latter takes a more technological stand and is thus somewhat closer in spirit to this tutorial, but has only just published its third issue! The Kluwer journal is slightly older but still only in its third year.

Other journals have had special editions on CSCW, including *International Journal of Man-Machine Studies* (as it then was called!), *Communications of the ACM* and *Interacting with Computers.* The papers from the two special issues of IJMMS were later published as a book:

- S. Greenberg (ed.). Computer Supported Cooperative Work and Groupware, Academic Press, London, 1991.

In addition, older papers on CSCW can be found in a variety of journals including ACM TOIS, Comm. ACM, IEEE Computer and all the general HCI journals.

Conferences

There are two major biennial conferences on CSCW which meet in the autumn of alternate years.

- CSCW — sponsored by the ACM, 1986, ‘88, ‘90, ‘92. ‘94
- ECSCW — the European CSCW conference, 1989, ‘91, ‘93.

The ACM CSCW conference proceedings are published by ACM Press, but early ECSCW conference proceedings are hard to come by. A selection of ECSCW’89 papers were also published as:

- J.M. Bowers and S.D. Benford (eds.). *Studies in Computer Supported Cooperative Work: Theory, Practice and Design,* North-Holland, 1991.

CSCW issues and theory

The tutorial has not concentrated on the theoretical issues surrounding the human oriented analysis and design of cooperative systems. But, so that you don't come away thinking that all the issues are technological, here is a small selection of articles considering these broader issues. It is just a taster and for a balanced picture browse some of the above journals and conference proceedings.

- J. Grudin. Why CSCW applications fail: Problems in the design and evaluation of organisational interfaces, *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88)*, pp. 85-93, ACM Press, 1988.

  Examines, based on case studies, some of the social and organisational issues which determine the successful adoption of groupware. Many of the issues which are now part of the accepted vocabulary of CSCW are found here.

Ethnography

Ethnography has become a major force in CSCW and Suchman's book probably contributed more than any other factor to the popularisation (if not acceptance!) of ethnography within HCI. Heath and Luff is regarded as a classic example of ethnographic analysis within CSCW and Bentley et. al. shows how such analysis can be linked with traditional computer systems design.

- L. Suchman. *Plans and Situated Actions: The Problem of Human-Machine Communication*, Cambridge university press, 1987.
- C. Heath and P. Luff. Collaborative activity and technological design: task coordination in London Underground control rooms, Proceedings of ECSCW'91, L. Bannon, M. Robinson, K. Schmidt (eds.), Amsterdam, pp. 65–80, 1991.
- R. Bentley, J. Hughes, D. Randall, T. Rodden, P. Sawyer, D. Shapiro and I. Sommerville. Ethnographically -informed systems design for air traffic control, *Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '92)*, J. Turner and R. Kraut (eds.), pp. 123–130, ACM Press, 1992.

Other approaches

- T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*, Ablex. Norwood, New Jersey, 1986.

  Strangely enough, Winograd and Flores argue, rather like Suchman, the importance of context. Based on work by Searle, they are proponents of speech-act theory, which declares that the importance of language is in that does something — language is action. However, despite the similarity of many of their arguments to those of Suchman, speech-act theory is used as the basis of Coordinator the (in)famous highly structured message system, which has been attacked primarily for its rigidity.

- H.H. Clark, and S.E. Brennan. Grounding in communication, *Socially Shared Cognition*, L.B. Resnick, J. Levine and S.D. Behreno (eds.), American Psychology Association, Washington, 1991.

  This takes the view that the meaning of language is negotiated and that one of the purposes of communication is establishing a common ground — a shared understanding — that can be used to further cooperative endeavour.

Groupware overviews

Two survey articles. They are a few years old now, but the categories of groupware still stand.

- C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some issues and experiences, *Communications of the ACM*, 34(1), pp. 38-58, January 1991.
- T. Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3), 1991.

Specific applications and systems

Computer mediated Communication

See also the description of Coordinator in Winograd and Flores.

- E.A, Dykstra, and R.P. Carasik. Structure and support in cooperative environments: The Amsterdam Conversation Environment, *International Journal of Man Machine Studies*, 34(3), pp. 419–434 March 1991.

  A message system which took the opposite approach to Coordinator. It supplied a free hypertext style conversation system, relying on the participants to establish their own local structure, as opposed to a global structure imposed by the system.

- T.W. Malone, K.R. Grant, K. Lai, R. Rao and D. Rosenblitt. Semi-structured messages are surprisingly useful for computer supported coordination, *ACM Transactions on Office Systems*, 5(2), pp. 115–131, 1987.

  Describes Information Lens, which has semi-structured messages based around an extensible class hierarchy of message types. Users are free to override and extend the message types, and can use information in named message fields to automatically filter, sort and take action. A commercial version, OPAL, is also available.

Meeting rooms, design and decision support

- M. Stefik, D.G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multi-user interfaces, *ACM Transactions on Office Information Systems*, 5(2), pp 147-167 April 1987.

  Describes Colab, one of the earlier small group meeting rooms. The paper explores some of the software design issues faced — especially the realisation that general design principles need to be re-examined in the context of use.

P. Cook, C. Ellis, M. Graf, G. Rein, and T. Smith. Project NICK: Meetings augmentation and analysis, *ACM Transactions on Office Information Systems*, 5(2), pp. 132-146, April 1987.

Another early meeting room. Alternatively see the proceedings of CSCW'86 which has a paper of the same name (with an additional author!).

•• J. Conklin. "gIBIS: A hypertext tool for exploratory policy discussion." Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '88), pp. 140–152, ACM Press, 1988.

An hypertext like tool for capturing design rationale — the issues faced and reasons for particular design decisions. gIBIS only supported asynchronous collaboration, but a latter tool rIBIS allowed synchronous use.

• H. Ishii and N. Miyake. Towards an open shared workspace: computer and video fusion approach of TeamWorkStation, *Communications of the ACM*, 34(12), pp. 37-50, December 1991.

TeamWorkStation is an example of a combined video and electronic drawing board. Video images of the participants faces and desktops are fused in various combinations with computer based images.

Shared applications and artefacts

• C.M. Neuwirth, D.S. Kaufer, R. Chandhok, and J. Morris. Issues in the design of computer support for co-authoring and commenting, *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '90)*, ACM Press, 1990.

• J.S. Olson, G.M. Olson, L.A. Mack, and P. Wellner. Concurrent editing: The group's interface, *Proceedings of Interact '90*, D. Diaper (ed.), Elsevier Science, pp. 834-840, 1990.

Note the difference in titles between these two papers. Each describe their own systems, but in the context of general analysis. However, a co-authoring system may or may not include shared editing. Editing concentrates on the actual generation and revision of text, whereas co-authoring may include other aspects of the authoring process such as brain storming, idea organisation, negotiation and different participant roles.

• I.H. Witten, H.W. Thimbleby, G. Coulouris, and S. Greenberg. Liveware: A new approach to sharing data in social networks, *International Journal of Man Machine Studies*, 34(3), pp. 337-348, March 1991.

Liveware is the only serious attempt I have seen which addresses the problem of simultaneous update when participants have no network connection.

Architectures and toolkits

- T. Rodden, J. Mariani and G. Blair, Supporting cooperative applications, *Computer Supported Cooperative Work*, 1, pp. 41-67, 1992.

  Review article of issues surrounding the technological support for groupware.

- A. Schill, *Cooperative Office Systems*, Prentice Hall, 1995.

  Describes international standards in distributed computing, databases and document description. Aimed principally, but not solely, at asynchronous groupware.

- R.D. Hill. The Rendezvous Constraint Management System, *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'93)*, ACM Press, pp. 225–234, 1993.

  The Rendezvous language and architecture supports one-way constraints and was designed for cooperative applications. This paper describes some of the technical details and compares it with other (single user) constraint based architectures.

- P. Dewan and R. Choudhary. Primitives for programming multi-user interfaces, *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'91)*, ACM Press, pp. 69–78, 1991.

  Describes SUITE a multi-user application toolkit based on active values.

- R. Bentley, *Supporting Multi-User Interface Development for Cooperative Systems*, Ph.D. Thesis, University of Lancaster, UK, 1994.

  Describes MEAD's architecture and development environment in detail. It also includes a general review of groupware implementation and toolkits. See also articles in CSCW'92 about the context in which MEAD was developed and in IEEE Computer, 27(5), for a shorter description of MEAD.

- C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems, *Proceedings of SIGMOD'89*, ACM Press, pp. 399-407, 1989.

  Describes the algorithms used in the Grove synchronous editor. This allows unrestricted (no locking) fine grain shared editing with a replicated architecture — no mean feat.

- G.D. Abowd and A.J. Dix. Giving undo attention, *Interacting with Computers*, 4(3), pp. 317-342, 1992.

  Discusses undo in a groupware context. One persons actions may depend on the previous actions of another participant, thus making the undo of the earlier action problematical.

AppleEvents HyperTalk Examples

The Stacks

This section describes the operation and HyperTalk code of the 'OXO' and 'Talk' stacks which are on the example disk.

The OXO stack is part of stack of three cards. The first is a solitaire version of the game and the second a simple message send card. These put collect the ingredients used in the final card – the game itself. The script is only a few screen fulls of code. The stack has online documentation, so consult that also.

The Talk stack consists of only one card! It implements a two-person phone-call-like conversation. It is similar to the conversation card in the 'AppleEvent Primer' stack distributed with HyperCard, but adds a few flourishes and is of course separated from the tutorial material.

Do look at 'AppleEvent Primer' for other examples of the use of AppleEvents.

Advantages and disadvantages of AppleEvents

Using HyperCard to generate prototype groupware means that one can make a reasonably respectable user interface. Compare the Talk stack with the similar UNIX based code. The UNIX equivalent is purely terminal based. To add a graphical interface, one would have to add vast amounts of window manager code. However, the restricted data types available in HyperCard mean that it is very difficult to program the complex algorithms one needs for robust groupware.

Of course, AppleEvents are not limited to HyperTalk programs and any Macintosh application can send or receive them. So, one development option would be to use HyperCard clients and a C or Pascal server. This means that each operates where it is best suited: HyperCard for the user interface and C for the complex server algorithms.

In short, HyperCard with AppleEvents makes a good platform for early prototyping, but would normally need something else for production software.

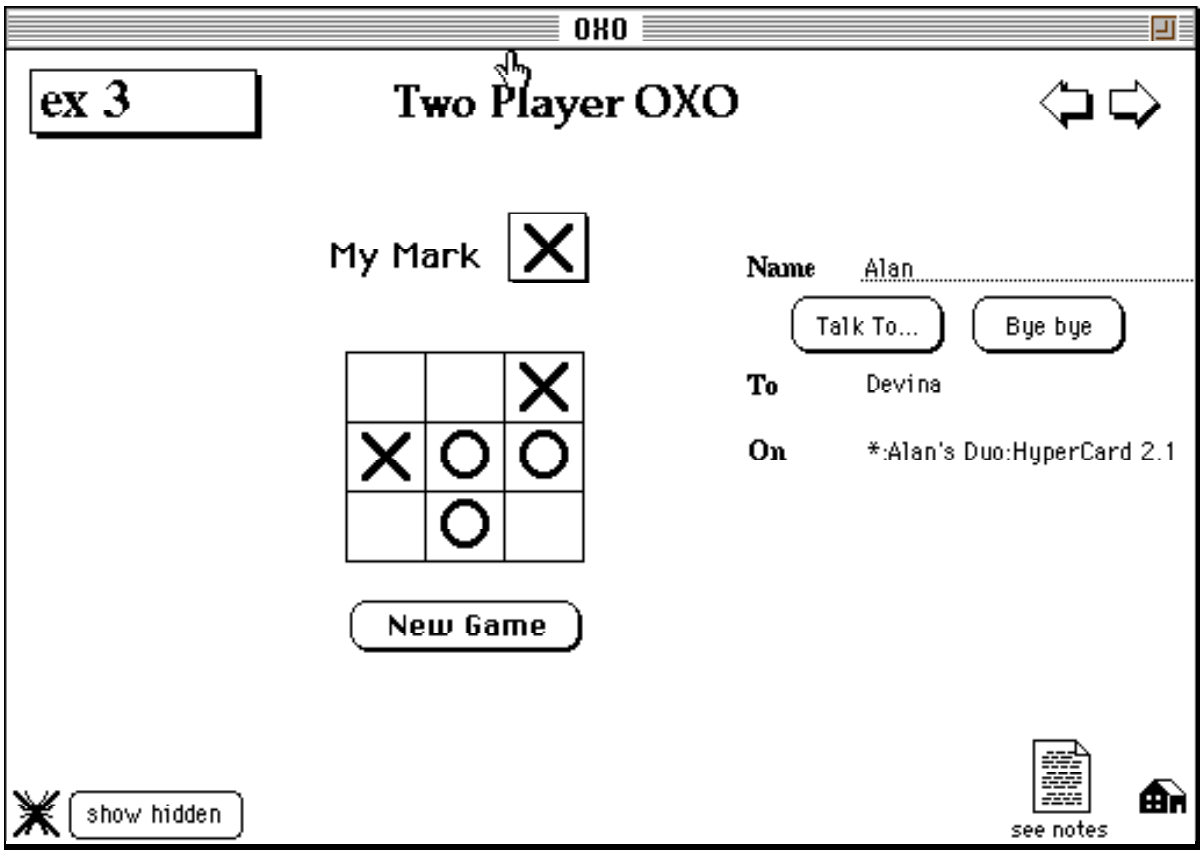
Using the OXO stack

The OXO stack has three exercise cards each with associated on-line notes. The third card is the networked game of noughts and crosses.

The card has two main areas:

- The playing board and associated functions in the middle.
- A communications area to setup the play on te right

In addition there are some debugging buttons on the botton left.



Starting a game

On entering the stack, you may find a name there from last time it is used. If not, or if it is not your name, enter it before starting a game. If you don't you'll be known by the wrong name, or be 'Anon'.

You next have to find someone to play with ... press the 'Talk To...' button. This enters a chooser-style dialogue from which you can select a machine and program

(another copy of HyperCard) to whom to talk. Unfortunately at this stage you can't find the name of who you're going to talk to!

If the person at the other end is running the OXO stack, has the right card up and wants to talk to you, the game can begin. If not, you get an error message. If all is well, the name and address of your conversant are now displayed.

On the other end ...

Alternatively, someone else may try to play with you (using their 'Talk To...' button) while you have the OXO stack open. You will get a dialogue box asking you whether you want to play with the person. This time you do get told who they are (if they entered their name right). If you agree, the play commences.

Playing

From here on it doesn't matter who started the game, both sides are equal and can play at any time. The program does not enforce proper turn taking – it is up to the players whether they cheat or not. You play by clicking on a square on the board.

At any time (presumably when the game is won or lost) either player can press the "New Game" button to clear both borads and start afresh.

Alos at any time (but presumably at the start of a game!) either player can click in theor "My Move" icon to change their play symbol. The other player's symbol will automatically get changed correspondingly.

Finishing a game session

Finally, to finish completely you press the 'Bye Bye' button. The other player gets a dialogue box to say they have been cut off.

Interruptions

It may happen that someone tries to contact you while you are playing with someone else. Alternatively, you may press 'Talk To...' before finishing off your last conversation. If you agree to a new request to play, or initiate a new game yourself, your original partner is not told – the results are predictable and chaotic!

Debugging

The 'show hidden' button reveals various hidden buttons that have been used to implement the game. The 'bug' button turns on and off the reporting of AppleEvents. When debugging is on you will get a dialogue box every time an AppleEvent arrives from another machine. You can see the messages fly ...

Implementation

The buttons each call a single HyperTalk handler in the card script:

button	script
	doTalkTo
	sayByeBye
	newGame
My Move 	toggleSymbol
(board play) 	makeMove

AppleEvent protocol

The stack sends three types of AppleEvent:

PleaseTalk **myName**

Sent by doTalkTo requesting that a conversation be started.
MyName is the name of the sender (from the name field).
The remote stack can send one of two replies:

LetsPlay **otherName**	—	OK start a game
GetLost	—	other user is busy.

(The user interface is less frank!)
They only need to exchange names as the sender's AppleTalk address is automatically transmitted as part of any AppleEvent.

remoteMove **loc**

Sent by makeMove to say where a move has been made.

remoteSymbolO and remoteSymbolX

Sent by toggleSymbol to tell remote HyperCard to change play symbol

remoteNewGame

Sent by newGame to tell the remote HyperCard to clear the board and start a new game.

ByeBye

Sent by sayByeBye telling the remote stack that the play is over.

The Script

The handlers in the script are organised into three sections:

Sending — handlers which send messages to the remote stack
most invoked directly by a user interface button

```
doTalkTo
sayByeBye
newGame
toggleSymbol
SendSymbol
makeMove
```

Receiving — messages received from remote stack

```
PleasePlayOxo  name
ByeBye
remoteNewGame
remoteSymbolO
remoteSymbolX
remoteMove  loc
```

Utilities — other minor handlers and utility functions

communication fields

```
startPlay  name,  address
endPlay
```

board and play symbols

```
clearBoard
setSymbolO
setSymbolX
```

field access functions

```
function  getMyName
function  getRemoteName
function  getRemoteHyperCard
```

Sending

> Handlers which send messages to the remote HyperCard.
> These are invoked by buttons in the user interface

doTalkTo

> handler for "Talk To..." button
> gets name of other stack and establishes conversation
> remote stack may give a "Busy" reply or may
> not be an "OXO" stack, in which case the reply
> is not recognised and an error given

```
on doTalkTo
```

> The HyperTalk command: answer program ""
> puts up a chooser-style dailogue box whereby you can
> search for a program across the AppleTalk network.

```
    answer program ""
    if it is empty then exit to HyperCard
    put it into remoteHyperCard
```

> now check the other HyperCard is running OXO
> and if so whether it wants to play

```
    send "PleasePlayOxo" && quote & getMyName() & quote ¬
            to program remoteHyperCard
    put the result into theResult
    if word 1 of theResult is "GetLost" then
        answer "I think you're friend is busy at the moment"
        put empty into remoteHyperCard
        exit to HyperCard
    end if
    if word 1 of theResult is not "LetsPlay" then
        answer "I don't think your friend has OXO open" ¬
            & return & "The error message was: " & theResult
        put empty into remoteHyperCard
        exit to HyperCard
    end if
```

> theResult should now be "LetsPlay machine-name"

```
    delete word 1 of theResult
    startPlay theResult, remoteHyperCard
    SendSymbol
```

```
end doTalkTo
```

sayByeBye

> handler for ByeBye button
> confirms that user wants to finish
> and then informs remote stack

```
on sayByeBye
    get getRemoteHyperCard()
    put it into remoteHyperCard
    if it is not empty
    then
        answer "Do you really want to finish the game" ¬
                with "finish game" or "keep playing"
        if it is "keep playing" then exit to HyperCard
        send "ByeBye" to program remoteHyperCard
    end if
    endPlay
end sayByeBye
```

newGame

> handler for NewGame button
> clears board and passes request to remote program

```
on newGame
    clearBoard
    get getRemoteHyperCard()
    put it into remoteHyperCard
    send "remoteNewGame" to program remoteHyperCard without reply
end newGame
```

toggleSymbol

> handler for mySymbol button
> toggles the choice of "O" or "X"

```
on toggleSymbol
    if the icon of cd btn "mySymbol" = the icon of cd btn "X"
    then
        setSymbolO
        get getRemoteHyperCard()
        put it into remoteHyperCard
        send "remoteSymbolO" to program remoteHyperCard without reply
    else
```

```
        setSymbolX
        get getRemoteHyperCard()
        put it into remoteHyperCard
        send "remoteSymbolX" to program remoteHyperCard without reply
    end if
end toggleSymbol
```

SendSymbol

> handler to send the current symbol to the other player at the beginning of a game

```
on SendSymbol
    if the icon of cd btn "mySymbol" = the icon of cd btn "X"
    then
        get getRemoteHyperCard()
        put it into remoteHyperCard
        send "remoteSymbolX" to program remoteHyperCard without reply
    else
        get getRemoteHyperCard()
        put it into remoteHyperCard
        send "remoteSymbolO" to program remoteHyperCard without reply
    end if
end SendSymbol
```

makeMove

> handler for board buttons

```
on makeMove
    if the icon of the target is 0              -- empty
    then
        set the icon of the target to the icon of card button "mySymbol"
        get getRemoteHyperCard()
        put it into remoteHyperCard
        send "remoteMove" && the short name of the target ¬
                to program remoteHyperCard without reply
    else
        play "boing"
    end if
end makeMove
```

Receiving

> Messages received from remote HyperCard. These are invoked on a one-to-one basis for each type of AppleEvent sent.

PleasePlayOxo

> request to play
> needs to confirm the owner wants to play

```
on PleasePlayOxo name
    request AppleEvent sender
    put it into theSender
```

> the name of the stack who sent this message

```
    answer "Do you want to play with " & name & return & " on " & theSender ¬
           with "Play" or "Busy"
    if it is "Busy"
    then
        reply "GetLost"
        exit to HyperCard
    end if

    reply "LetsPlay" && getMyName()
    startPlay name, theSender
end PleasePlayOxo
```

ByeBye

> remote request to end game

```
on ByeBye
    answer "Your friend says Bye Bye" & return & "the conversation is finished"
    endPlay
end ByeBye
```

remoteNewGame

> remote request for new game

```
on remoteNewGame
    clearBoard
end remoteNewGame
```

remoteSymbolO / X

> remote choice of "O" or ""
> note that the name of the message is the other players choice,
> this player gets the opposite.

```
on remoteSymbolO
    setSymbolX
end remoteSymbolO

on remoteSymbolX
    setSymbolO
end remoteSymbolX
```

remoteMove

> remote move made by the other player

```
on remoteMove loc
```

> if icon is 0 the square is empty

```
    if the icon of card button loc is 0
    then
        set the icon of card button loc to the icon of card button "otherSymbol"
    else
        play "boing"
    end if
end remoteMove
```

Utilities

communication fields

Set and clear opponents name and address

```
on startPlay name, address
    put name into card field "To Value"
    put address into card field "On Value"
end startPlay

on endPlay
    put empty into card field "To Value"
    put empty into card field "On Value"
end endPlay
```

clearBoard

clear the playing board

```
on clearBoard
    repeat with n = 1 to 9
        set the icon of cd btn ("oxo" & n) to empty
    end repeat
end clearBoard
```

setSymbolO / X

set this players symbol to "O"or "X"

```
on setSymbolO
    set the icon of cd btn "mySymbol" to "O"
    set the icon of cd btn "otherSymbol" to "X"
end setSymbolO

on setSymbolX
    set the icon of cd btn "mySymbol" to "X"
    set the icon of cd btn "otherSymbol" to "O"
end setSymbolX
```

field access functions

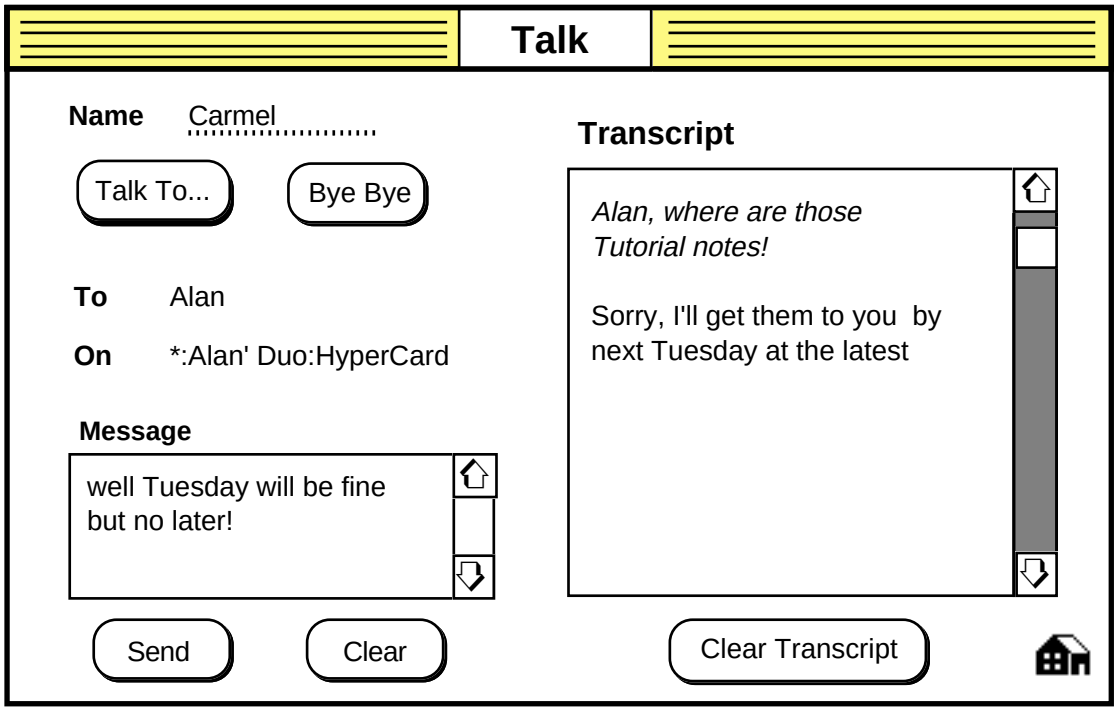
These functions simply access certain text fields.
They are included to insulate the rest of the code from the names of particular fields.
The first function getMyName also defaults the name to "Anon" if none has been entered.

```
function getMyName
    get card field "My Name"
    if it is empty
    then
        return "Anon"
    else
        return it
    end if
end getMyName

function getRemoteName
    return card field "To Value"
end getRemoteName

function getRemoteHyperCard
    return card field "On Value"
end getRemoteHyperCard
```

Using the Talk stack

As you can see the card has three main areas:

- An information area to the upper left, where you enter your name and where your conversant's name and address (electronic!) are displayed.
- A message area to the lower left where you enter your messages.
- A transcript to the right where a record of the conversation is displayed and can be reviewed.



Starting a conversation

On entering the stack, you may find a name there from last time it is used. If not, or if it is not your name, enter it before starting any conversations. If you don't you'll be known by the wrong name, or be 'Anon'.

You next have to find someone to talk to ... press the 'Talk To...' button. This enters a chooser-style dialogue from which you can select a machine and program (another copy of HyperCard) to whom to talk. Unfortunately at this stage you can't find the name of who you're going to talk to!

Talk

If the person at the other end is running the Talk stack and wants to talk to you, the conversation has begun. If not, you get an error message. If all is well, the name and address of your conversant are now displayed.

On the other end ...

Alternatively, you may be on the other end of a call. If someone else tries to get in touch with you (using their 'Talk To...' button) and you have the Talk stack open, you get a dialogue box asking you whether you want to talk to the person. This time you do get told who they are (if they entered their name right). If you agree, the conversation commences.

Talking

From here on it doesn't matter who started the conversation, both sides are equal and can enter messages at any time — no turn taking.

Once you are in a conversation either you or your conversant (who has an identical screen) can type into your own message fields. When you hit the 'Send' button the message is added to your transcript and to that of your conversant.

Your own messages appear in italic in the transcript and your conversant's appear in plain type. On your conversant's screen they will be the other way round (i.e., your messages will be plain type). This is the same convention as is used in the 'AppleEvent Primer' stack.

Finishing a conversation

Finally, to finish a conversation, you press the 'Bye Bye' button. The other conversant gets a dialogue box to say they have been cut off.

Interruptions

It may happen that someone tries to contact you while you are talking to someone else. Alternatively, you may press 'Talk To...' before finishing off your last conversation. In this case, you are asked to confirm that you want to finish the existing conversation. If you do, the other conversant will be informed (as if you had hit 'the Bye Bye' button), otherwise the interruption will be cancelled.

Trivia

The 'Clear' buttons clear the corresponding fields, and the home button goes home closing any existing conversation on the way.

Implementation

Most of the HyperTalk code is in the card script (it really should be in the background script, but in a one card script ...). The buttons each call a single handler in the card script:

button	script
Talk To...	findRemoteHyperCard
Bye Bye	sayByeBye
Send	sendMessage
Clear	clearMessage
Clear Transcript	clearTranscript

AppleEvent protocol

The stack sends three types of AppleEvent:

PleaseTalk **myName**

Sent by findRemoteHyperCard requesting that a conversation be started. MyName is the name of the sender (from the name field). The remote stack can send one of two replies:

LetsTalk **otherName** — OK start a conversation

GetLost — other user is busy.

(The user interface is less frank!)

They only need to exchange names as the sender's AppleTalk address is automatically transmitted as part of any AppleEvent.

TalkMessage **message**

Sent by sendMessage where message is a single line of text.. HyperTalk AppleEvents cannot easily have multi-line messages hence the user's message box is sent line by line to the remote stack. There is no reply, but the code does check for successful receipt. It really ought to be non-blocking (remember the rules for robustness) — try sending messages simultaneously in both directions can you crash it?

ByeBye

Sent by sayByeBye telling the remote stack that the chat is over.

The Script

The handlers in the script are organised into three sections:

Receiving — messages received from remote stack

PleaseTalk name
TalkMessage textSent
ByeBye

Sending — handlers which send messages to the remote stack

findRemoteHyperCard
sendMessage
sayByeBye
function checkExistingConversation

Utilities — other minor handlers and utility functions

housekeeping

newConversation name, address
endConversation
closeCard

clear particular fields

clearMessage
clearTranscript

text processing

function stripBlankLines text

field access functions

function getMyName
function getRemoteName
function getRemoteHyperCard
function getMyName

Receiving

Messages received from remote HyperCard. These are invoked on a one-to-one basis for each type of AppleEvent sent.

The stack also contains the following commented out handler, which can be used to monitor AppleEvents during debugging. It captures each incoming event and puts up a dialogue box saying what it is and where it has come from. It then passes the event on for normal processing. Uncomment this in order to get a feeling for the passing to and fro of AppleEvents.

```
on AppleEvent, class, id, sender
    request AppleEvent data
    answer "AppleEvent" && class && id & return¬
                    & "from" && sender & return & it
    pass AppleEvent
end AppleEvent
```

PleaseTalk

Request to start a new conversation. It needs to confirm the owner wants a new chat and warn that any existing conversation will be lost.

```
on PleaseTalk name

    request AppleEvent sender  -- the stack who sent this message
    put it into theSender

    answer "Do you want to talk to " & name & return ¬
                    & "     on " & theSender  with "Talk" or "Busy"
    if it is "Busy"
    then
        reply "GetLost"
        exit to HyperCard
    end if

    if checkExistingConversation() is "Cancel"
    then
        reply "GetLost"
        exit to HyperCard
```

```
    end if
    reply "LetsTalk" && getMyName()
    newConversation name, theSender
end PleaseTalk
```

TalkMessage

> Receive a line of text and add it to transcript.

```
on TalkMessage textSent
    put textSent & return after cd fld "Transcript"
    set scroll of cd fld "Transcript" to 32000
end TalkMessage
```

ByeBye

> Remote request to end conversation.
> Warn user and tidy up.

```
on ByeBye
    answer "Your friend says Bye Bye" & return & "the conversation is finished"
    endConversation
end ByeBye
```

Sending

Handlers which send messages to the remote HyperCard.
These are invoked by buttons in the user interface

findRemoteHyperCard

Handler for "Talk To..." button gets name of other stack and establishes conversation. The remote stack may give a "Busy" reply or may not be a "Talk" stack, in which case the reply is not recognised and an error is given

```
on findRemoteHyperCard
```

The handler check4aeSupport is defined in the AppleEvent Primer stack. It checks to see whether the version of HyperCard and of the operating system are suitable to manage AppleEvents. The code for check4aeSupport is installed in the stack script, but is not reproduced here.

```
    check4aeSupport

    if checkExistingConversation() is "Cancel"
    then
        exit to HyperCard
    end if
```

The HyperTalk command: answer program ""
puts up a chooser-style dailogue box whereby you can search for a program across the AppleTalk network.

```
    answer program ""
    if it is empty then exit to HyperCard
    put it into remoteHyperCard
```

Now check the other HyperCard is running Talk and if so whether it wants a chat. Assuming Talk is running, the remote user will get a dialogue box with the option of accepting the conversation, or replying “Busy”.

```
    send "PleaseTalk" && quote & getMyName() & quote to program remoteHyperCard
    put the result into theResult
```

> If the AppleEvent returns "GetLost" then the remote user has answered “Busy” to the dialogue

```
if word 1 of theResult is "GetLost" then
    answer "I think you're friend is busy at the moment"
    put empty into remoteHyperCard
    exit to HyperCard
end if
```

> Having dealt with that case, the reply should be: LetsTalk
> If it is not, the remote stack cannot be “Talk” or some network error has occured.

```
if word 1 of theResult is not "LetsTalk" then
    answer "I don’t think your friend has Talk open"
                & return & "The error message was: " & theResult
    put empty into remoteHyperCard
    exit to HyperCard
end if
```

> The reply is of the form: LetsTalk **name**
> The first word is deleted leaving the name to be displayed in the appropriate field.

```
delete word 1 of theResult
newConversation theResult, remoteHyperCard

end findRemoteHyperCard
```

sendMessage

> Handler for Send button, which sends a message to the remote stack.
> The message is sent line by line as we can not have multi-line arguments to the ‘send’ command. It also adds the message to its own transcript in italic.

```
on sendMessage
    check4aeSupport
```

> Check whether we are in a conversation.
> N.B. we could have disabled the "Send " button but even if we had, it would be good defensive programming to check

```
    get getRemoteHyperCard()
    put it into remoteHyperCard
    if remoteHyperCard is empty then
        answer "You need to choose your friend’s HyperCard program first." ¬
                                                    with "Cancel" or "Talk To…"
```

```
        if it is "Cancel" then exit sendMessage
        else if it is "Talk To..." then send mouseUp to cd btn "Talk To..."
    end if
    if remoteHyperCard is empty then exit to HyperCard

    put stripBlankLines(card field "Message") into theMessage
    put theMessage into toSend
```

> Send message line by line to remote stack

```
    repeat until toSend is empty
        put line 1 of toSend into thisMess
        send "TalkMessage"  && quote & thisMess & quote ¬
                                                to program remoteHyperCard
        if the result is not empty then
            answer "Unable to send to" && remoteHyperCard & "." & ¬
                                return & "The error reported was:" && the result
            exit to HyperCard
        end if
        delete line 1 of toSend
    end repeat
```

> Add message to our transcript

```
    put (number of lines in cd fld "Transcript") into lineCount
    put theMessage & return & return after cd fld "Transcript"
    clearMessage
    put (number of lines in cd fld "Transcript") into newLineCount
    set textstyle of line LineCount + 1 to ¬
                        newLineCount of cd fld "Transcript" to italic
    set scroll of cd fld "Transcript" to 32000

end sendMessage
```

sayByeBye

> Handler for ByeBye button confirms that user wants to finish and then informs the remote stack.

```
on sayByeBye
    get getRemoteHyperCard()
    put it into remoteHyperCard
    if it is not empty
    then
        answer "Do you really want to stop talking" ¬
                                with  "keep talking" or "say bye bye"
        if it is "keep talking" then exit to HyperCard
        send "ByeBye" to program remoteHyperCard
    end if
    endConversation
end sayByeBye
```

checkExistingConversation

Called by any handler which would conflict with an existing conversation. If there is one the user is asked to confirm whether or not it should be abandoned returns: OK — if the user agrees or there is none Cancel — if the user wants to continue

```
function checkExistingConversation
    get getRemoteHyperCard()
    put it into remoteHyperCard
    if remoteHyperCard is empty
    then
        return "OK"
    end if
    answer "this means finishing your chat to" && getRemoteName() & return ¬
            & "    on " & remoteHyperCard  with "keep talking" or "finish chat"
    if it is "keep talking"
    then
        return "Cancel"
    else
        send "ByeBye" to program remoteHyperCard
        endConversation
        return "OK"
    end if
end checkExistingConversation
```

Utilities

housekeeping

> Miscelaneous handlers to set fields and do tidying up.

```
on newConversation name, address
    put name into card field "To Value"
    put address into card field "On Value"
end newConversation

on endConversation
    put empty into card field "To Value"
    put empty into card field "On Value"
end endConversation

on closeCard
    set scroll of cd fld "Transcript" to 0
    pass closeCard
end closeCard
```

clear particular fields

```
on clearMessage
    put empty into card field "Message"
end clearMessage

on clearTranscript
    put empty into card field "Transcript"
end clearTranscript
```

stripBlankLines

> Text processing function. Removes leading and trailing blank lines from its text argument.

```
function stripBlankLines text
    repeat while the first line of text is empty
        delete the first line of text
    end repeat
    repeat while the last line of text is empty
        delete the last line of text
    end repeat
    return text
end stripBlankLines
```

field access functions

These functions simply access certain text fields.
They are included to insulate the rest of the code from the names of particular fields.
The first function getMyName also defaults the name to "Anon" if none has been entered.

```
function getMyName
    get card field "My Name"
    if it is empty
    then
        return "Anon"
    else
        return it
    end if
end getMyName

function getRemoteName
    return card field "To Value"
end getRemoteName

function getRemoteHyperCard
    return card field "On Value"
end getRemoteHyperCard
```

UNIX sockets
C Program Listings

The following are listings of the programs described in the text and some of the library code which supports them.

The code is divided into 4 sections:

Two person 'talk' programs
- simple-server.c
- simple-client.c

Basic library
- sock.h
- sock.c

Multi-party client/server programs
- server.c
- client.c

Full library include files
- inform.h
- line_by_line.h
- mess.h
- monitor.h

The code is also available on the examples disk but in places differs slightly from the printed code. In particular, the example disk code is written so as to be compatible with different versions of C (K&R, ANSI and C++) but the printed versions are the ANSI C version.

In the server and client programs, are '#define'ed constants HOST and PORTNO. These must be set to the same value, and HOST must be the machine that ther server is running on. In fact, the version of 'socket.c' supplied on the examples disk does have the means to set the host from the command line, but this is omitted from the printed versions to simplify the code.

Only the headers are given for most of the library files as these are assumed to be 'given'. The exception is sock.c which is included so you can see raw UNIX socket code. The notification library 'inform.c' and 'line_by_line.c', packages up UNIX select, and you can use this as an example (on disk) if you want to use select directly.

Two person ‘talk’

Contents

The following programs operate a simple ‘talk’ like program.
One participant runs the server and one the client.

The turntaking is strict, each program waits when it is not its turn. The user may not be aware of the behaviour becouse of keyboard buffering. This is obviously a very crude form of communication, but means that the program is far simpler to understand.

Both programs consist of a single ‘`main`’ procedure which is of the form:

```
initialise
loop
        client → server message
        server → client message
```

The program exits when either participant enters an ‘end of file’ (usually ctrl-D).

The network part of programs are written using low-level I/O (read and write) rather than stdio calls. In principle, one can attach a network file descriptor to a stdio file stream, but often one will want to use binary coding of the messages and also have some control of the internal buffering. If you’re network programming, you can’t really avoid getting your fingers dirty!

See the slides and header page of the next section ‘Basic library’ for a short description of the socket function calls.

```
/****************************************/
/*                                      */
/*      simple-server.c                 */
/*                                      */
/*      Alan Dix                        */
/*                                      */
/*      turntaking server               */
/*      HCI'94 Tutorial                 */
/*                                      */
/*      26th July 1994                  */
/*                                      */
/****************************************/

#include <stdio.h>
#include <string.h>
#include "sock.h"


#define PORTNO  6789
#define HOST   "gamma"
                  /*  needs to be set to server's machine  */

#define BUFF_LEN 200
char buff[BUFF_LEN];
int buf_len = BUFF_LEN-1;
                     /*  allow room for terminating '\0'  */


main(argc,argv)
    int argc;
    char **argv;
{
    int    port_fd,  client_fd;
    int    len;

  /*  establish port  */
    port_fd = inet_socket(HOST,PORTNO);
  /*  negative result on failure     */
    if ( port_fd < 0 ) { perror("socket"); exit(1); }
    printf("start up complete\n");

  /*  wait for client to connect  */
    client_fd = sock_accept(port_fd);

  /*  only want one client, so close port_fd  */
    close(port_fd);
```

```
38    /*  talk to client  */
39      for(;;) {
40        /*  wait for client's message  */
41          len = read(client_fd,buff,buf_len);
42          if (len == 0) {
43              printf("client finished the conversation\n");
44              break;
45              }
46          buff[len] = '\0';
47          printf("client says: %s\n",buff);
48        /*  now it's our turn  */
49          printf("speak: ");
50          if ( gets(buff) == NULL ) {
51                                    /* user typed end of file  */
52              close(client_fd);
53              printf("bye bye\n",id);
54              break;
55              }
56          write(client_fd,buff,strlen(buff);
57      }
58      exit(0);
59  }
```

```
1   /****************************************/
2   /*                                      */
3   /*      simple-client.c                 */
4   /*                                      */
5   /*      Alan Dix                        */
6   /*                                      */
7   /*      turntaking client               */
8   /*      HCI'94 Tutorial                 */
9   /*                                      */
10  /*      26th July 1994                  */
11  /*                                      */
12  /****************************************/

13  #include <stdio.h>
14  #include <string.h>
15  #include "sock.h"


16  #define PORTNO  6789
17  #define HOST   "gamma"   /*  set to serve4's machine  */

18  #define BUFF_LEN 200
19  char buff[BUFF_LEN];
20  int buf_len = BUFF_LEN-1;
21                        /*  allow room for terminating '\0'  */

22  main(argc,argv)
23      int argc;
24      char **argv;
25  {
26      int  server_fd;
27    /*  request connection to server   */
28      server_fd = inet_connect(HOST,PORTNO);
29    /*  waits for server to accept     */
30    /*  host is server's machine       */
31    /*  negative result on failure     */
32      if ( server_fd < 0 ) { perror("socket"); exit(1); }
33      printf("You can send now\n");

34    /*  talk to server  */
35      for(;;) {
36        /*  our turn first  */
37          printf("speak: ");
38          if ( gets(buff) == NULL ) {
39                                  /* user typed end of file  */
40              close(server_fd);
41              printf("bye bye\n");
42              break;
43              }
44          write(server_fd,buff,strlen(buff);


45        /*  wait for server's message  */
46          len = read(server_fd,buff,buf_len);
47          if (len == 0) {
```

```
48              printf("client finished the conversation\n");
49              break;
50              }
51          buff[len] = '\0';
52          printf("server says: %s\n",buff);
53      }
54      exit(0);
55  }
```

Basic library

Contents

sock.h
sock.c

These package up the UNIX socket calls and give a simplified (although less powerful) programmers interface. The version on disk also has support for communication via UNIX named pipes, but the printed version is for Internet sockets only.

The full implementation (sock.c) is given as well as the header. This is so you can see the standard UNIX operating system calls if you wish.

Three routines supplied are:

`inet_socket( hostname, portno )`
called by the server to establish an Internet port

`inet_connect( hostname, portno )`
called by the client to request a connection

`sock_accept( sock )`
called by the server to wait for and accept a client connection

Each returns a UNIX file descriptor. The one returned by `inet_socket` is not used directly to communicate with the client, but is used when establishing the connection. It is passed as a parameter to `sock_accept` and it is the file descriptor which is returned by this which is connected to the clients descriptor (returned by `inet_connect`). The resulting connection is two way. The connection is closed implicitly when the file descriptor is closed or when either program exits. In standard UNIX fashion this becomes apparent to the other end of the connection when a subsequent `read` system call returns nothing (end of file). The programmer is of course free to establish a more graceful close down, but should always be prepared for a broken connection if the other program crashes!

```
1   /****************************************/
2   /*                                      */
3   /*      sock.h                          */
4   /*                                      */
5   /*      Alan Dix                        */
6   /*                                      */
7   /*      package up socket garbage       */
8   /*                                      */
9   /*      5th January 1990                */
10  /*                                      */
11  /****************************************/


12    /*  all return negative number on failure  */

13  int  inet_socket(char *hostname,int portno);
14  int  inet_connect(char *hostname,int portno);
15  int  sock_accept(int sock);
```

```
1   /*****************************************/
2   /*                                       */
3   /*      sock.c                           */
4   /*                                       */
5   /*      Alan Dix                         */
6   /*                                       */
7   /*      package up socket garbage        */
8   /*                                       */
9   /*      5th January 1990                 */
10  /*                                       */
11  /*****************************************/

12  #include <sys/types.h>
13  #include <sys/socket.h>
14  #include <netinet/in.h>
15  #include <netdb.h>
16  #include <arpa/inet.h>
17  #include <stdio.h>
18  #include <string.h>
19  #include "sock.h"

20  static void check_inet_host(hostname)
21      char * hostname;
22  {
23      char real_hostname[32];
24      gethostname(real_hostname,32);
25      if ( strncmp(hostname,real_hostname) == 0 ) return;
26      mess( "program running on %s, must run on host %s\n",
27                              real_hostname,         hostname  );
28      exit(1);
29  }

30  int  inet_socket(hostname,portno)
31      char * hostname;
32      int    portno;
33  {
34      int     sd, code;
35      struct  sockaddr_in bind_addr;
36      check_inet_host(hostname);
37                      /* do this to by-pass bug in INET sockets*/
38      bind_addr.sin_family = AF_INET;
39      bind_addr.sin_addr.s_addr = 0;
40      bzero(bind_addr.sin_zero, 8);
41      bind_addr.sin_port = portno;
42      sd = socket(AF_INET, SOCK_STREAM,0);
43      if ( sd < 0 ) return sd;
44      code = bind(sd, &bind_addr, sizeof(bind_addr) );
45      if ( code < 0 ) { close(sd); return code; }
46      code = listen(sd, 1);
47      if ( code < 0 ) { close(sd); return code; }
48      return sd;
49  }

50  int  inet_connect(hostname,portno)
51      char * hostname;
```

```
52      int    portno;
53  {
54      int     sd, code;
55      struct  sockaddr_in bind_addr;
56      struct hostent *host;

57      host = gethostbyname(hostname);
58      if (host == NULL ) return -1;
59      bind_addr.sin_family = PF_INET;
60      bind_addr.sin_addr = *((struct in_addr *) (host->h_addr));
61      bind_addr.sin_port = portno;
62      sd = socket(AF_INET, SOCK_STREAM, 0);
63      if ( sd < 0 ) return sd;
64      code = connect(sd, &bind_addr, sizeof(bind_addr) );
65      if ( code < 0 ) { close(sd); return code; }
66      return sd;
67  }

68  int  sock_accept(sock)
69      int sock;
70  {
71      int     sd;
72      struct  sockaddr bind_addr;
73      int len=sizeof(bind_addr);
74      sd = accept(sock, &bind_addr, &len);
75      return sd;
76  }
```

Multi-party conference

Contents

server.c
client.c

These implement a multi-party conference using a full client server approach and using a notification based programming paradigm. A single server is run on a central machine and then several clients can be started. Any line of text typed on one participant's terminal will be echoed at all the rest.

The user interface is still exceedingly crude. When any participant types a line, the other participants receive a line like the following:

```
client n says :'my message'
```

where n is the number that the server has allocated to the speaking client. In a real application the participants would at some stage enter their name and this would be transmitted to the other participants. Also, the messages would have effects other than printing text! However, it gives the general flavour of a multi-party application.

The server does not act as a participant itself, but has a simple user interface to act as a monitor. As it is set up, it only responds to two commands 'help' and 'exit' which closes down the server. However, more commands, for example listing the participants, can be added by modifying the table `monitor_tab`.

Server.c

As well as the main routine, this has four callback functions:

`accept_client`	—	called when a new client requests a connection
`read_client`	—	called when a client sends a message
`term_line`	—	called when a line is entered at the server's terminal
`term_eof`	—	called when an 'end of file' is typed there

Client.c

This has four callback functions:

`read_socket`	—	client receives message from server
`term_line`	—	user types a line (message to send)
`term_eof`	—	user types 'end of file' (leave conference)

Note that neither client nor server put message lengths at the beginning of their network transmissions. The programs are thus subject to end-to-end message problems as described in the slides. Correcting this complicates the programs somewhat — try it as an exercise!!

```
/****************************************/
/*                                      */
/*      server.c                        */
/*                                      */
/*      Alan Dix                        */
/*                                      */
/*      electronic conference server    */
/*      HCI'94 Tutorial                 */
/*                                      */
/*      26th July 1994                  */
/*                                      */
/****************************************/

#include <stdio.h>
#include <string.h>
#include "mess.h"
#include "sock.h"
#include "inform.h"
#include "monitor.h"
#include "line_by_line.h"


#define  PORTNO   6789
#define  HOST    "gamma"

#define MAXCLIENTS      10

int   clients = 0;     /* count of active clients   */
int   ns[MAXCLIENTS];  /* file descriptor for each  */
                       /*            active client  */

int  read_client( int nd, int id );
int  accept_client( int sd, void *id );
term_line( int fd, void *id, char *buff );
term_eof( int fd, void *id );


main(argc,argv)
    int argc;
    char **argv;
{
    int  sd;

    sd = inet_socket(HOST,PORTNO);
    if ( sd < 0 ) { perror("socket"); exit(1); }
    inform_line_by_line(0,term_line,term_eof,NULL);
    inform_input(sd,accept_client,NULL);
    mess("start up complete\n");
    inform_loop();
    exit(0);
}



int  read_client(nd,id)
```

```
44      int  nd;
45      int  id;
46  {
47      int len, i;
48      char buf[256];
49      len = read(nd, buf, sizeof(buf)-1);
50      if (len == 0) {
51          mess("client %d leaving the conversation\n",id);
52          inform_input(nd,NULL,NULL);
53          ns[id] = 0;
54          }
55      else {
56          char mess_buff[256];
57          buf[len] = '\0';
58          mess("client %d says :'%s'\n", id, buf);
59          sprintf(mess_buff,"client %d says :'%s'", id, buf);
60          for ( i=0; i<clients; i++ ) {
61              if ( ns[i] > 0 ) {
62                  write(ns[i],mess_buff,strlen(mess_buff));
63                  }
64              }
65          }
66      return 0;
67  }


68  int  accept_client(sd,id)
69      int   sd;
70      void *id;
71  {
72      ns[clients] = sock_accept(sd);
73      if ( ns[clients] < 0 ) {
74          perror("accept");
75          return -1;
76          }
77      inform_input(ns[clients],read_client,clients);
78      mess("new client %d\n",clients);
79      ++clients;
80      if ( clients >= MAXCLIENTS ) {
81          mess("server saturated\n");
82          inform_input(sd,NULL,NULL);
83          }
84      return 0;
85  }
```

```
86
87  /* the following table is used by the library call       */
88  /*                                        'process_line'  */
89  /* to despatch user commands, the table has the form:    */
90  /*   { id, command_name, command_function, help_string } */
91  /* process_line is called with a line the user has typed */
92  /* and then uses the first word in that line to look up  */
93  /* in the table, and call the appropriate function       */
94  /* additional functions can be added to this table as    */
95  /* an aid to debugging                                   */
96  struct mon_tab_struct monitor_tab[] = {
97      { 0, "exit", inform_done, "close down server" },
98      { 0, 0, 0, 0 }
99  };


100 term_line(fd,id,buff)
101     int   fd;
102     void *id;
103     char *buff;
104 {
105     mess("command {%s}\n",buff);
106     perform_line(buff);
107     return 0;
108 }


109 term_eof(fd,id)
110     int   fd;
111     void *id;
112 {
113     inform_done();
114     mess("lost control terminal\n");
115     return 0;
116 }
```

```
1   /****************************************/
2   /*                                      */
3   /*      client.c                        */
4   /*                                      */
5   /*      Alan Dix                        */
6   /*                                      */
7   /*      electronic conference client    */
8   /*      HCI'94 Tutorial                 */
9   /*                                      */
10  /*      26th July 1994                  */
11  /*                                      */
12  /****************************************/

13  #include <stdio.h>
14  #include <string.h>
15  #include "mess.h"
16  #include "sock.h"
17  #include "inform.h"
18  #include "line_by_line.h"


19  #define  PORTNO  6789
20  #define  HOST    "gamma"


21  char in_buff[256], out_buff[256], line_buff[256];
22  int  line_len = 0;
23  int sd;  /* socket file descriptor */


24  read_socket( int fd, void *id);
25  term_line( int fd,  void *id, char *buff );
26  term_eof( int fd, void *id );


27  main(argc,argv)
28      int argc;
29      char **argv;
30  {
31      sd = inet_connect(HOST,PORTNO);
32      if ( sd < 0 ) { perror("socket"); exit(1); }
33      printf("You can send now\n");
34      inform_line_by_line(0,term_line,term_eof,NULL);
35      inform_input(sd,read_socket,NULL);
36      inform_loop();
37      exit(0);
38  }
```

```
39
40  read_socket(fd,id)
41      int   fd;
42      void *id;
43  {
44      int n = read(fd,in_buff,255);
45      if ( n==0 ) {
46          inform_done();
47          fprintf(stderr,"socket closed\n");
48          return 0;
49          }
50      in_buff[n] = '\0';
51      printf("received %s\n",in_buff);
52      fflush(stdout);
53      return 0;
54  }


55  term_line(fd,id,buff)
56      int   fd;
57      void *id;
58      char *buff;
59  {
60      mess("sending {%s}\n",buff);
61      write(sd,buff,strlen(buff));
62      return 0;
63  }


64  term_eof(fd,id)
65      int   fd;
66      void *id;
67  {
68      inform_done();
69      mess("client done\n");
70      return 0;
71  }
```

Library headers

Contents

Inform.h

This is the most important file giving the programmer' interface to a notification based set of routines. This will normally not be necessary in the client as this is likely to be running under an event-driven window manager. However, these routines allow simple text based clients to be written and also allow servers to be written in an event-driven paradigm.

`inform_input` — establishes a callback on a file descriptor. It is invoked every time the file can be read without blocking – that is, whenever input is ready.

`inform_output` — similarly, invokes the callback whenever one can output the file descriptor. Note that the client and server programs included do not use this, but instead assume that they can always do a `write` without it blocking. This will work most of the time, but may fail if operating system buffers become full.

`inform_slack` — invokes callback when nothing else is happening. Useful for animation (like HyperCard on idle).

`inform_loop` — this is called by the program when it has set up initial callbacks. It gives control to the notification based loop and the program is event driven from then on.

`inform_done` — program calls this when it wants to finish. It makes the `inform_loop` return and allows a graceful exit.

In addition, there are calls to establish timed periodic callbacks.

line_by_line.h

Allows the user to set up callbacks which are called on each line of user input. This is contrast to `inform_input` which calls the program's callback whenever a *buffer* full of input is ready. There is one principle call:

inform_line_by_line

which establishes 2 callbacks. One is called for each line input and the other when an end of file is encountered. There are also calls to turn off the callbacks.

mess.h — debugging and logging for server, optionally copies to a log file.

monitor.h —simple table driven command interpreter used by server to implement a simple console monitor.

```
/****************************************/
/*                                      */
/*      inform.h                        */
/*                                      */
/*      Alan Dix                        */
/*                                      */
/*      notifier like I/O multiplexer   */
/*                                      */
/*      5th January 1990                */
/*                                      */
/****************************************/


#include <sys/time.h>

extern int inform_finished;
extern int inform_tick_count;
extern int inform_tick_id;

  /*  type declarations for callbacks  */
typedef void *inform_id;
typedef (*inform_fun)( int fd, inform_id id );

  /*  establishing input/output callbacks  */
int    inform_input( int fd, inform_fun f, inform_id id );
int    inform_output( int fd, inform_fun f, inform_id id );
int    inform_exception( int fd, inform_fun f, inform_id id );

  /*  establishing callbacks for signals and idle time */
int    inform_signal( int signo, inform_fun f, inform_id id );
int    inform_slack( inform_fun f, inform_id id );

  /*  starting and finishing event driven behaviour  */
int    inform_done(void);
int    inform_one_loop(long t);
int    inform_loop(void);

  /*  establishing callbacks for periodic processing   */
int    inform_itimer_times( long p_s, long p_us,
                            long v_s, long v_us,
                            inform_fun f, inform_id id );
int    inform_itimer( struct itimerval *value,
                      inform_fun f, inform_id id );


#define FATAL_ERROR        -151
#define TICK_ALARM_MISSED  -150
```

```
/****************************************/
/*                                      */
/*      line_by_line.c                  */
/*                                      */
/*      Alan Dix                        */
/*                                      */
/*      line input protocol             */
/*                                      */
/*      8th January 1990                */
/*                                      */
/****************************************/


  /*  type declarations for callbacks  */
typedef void * id_type;
typedef void (*line_fun)( int fd, id_type id, char*buff );
typedef void (*eof_fun)( int fd, id_type id );


int inform_line_by_line( int fd,  line_fun line_f,
                         eof_fun eof_f, id_type id    );
   /*  establishes two callbacks                          */
   /*     line_f  is called with each line of input on fd  */
   /*     eof_f   is called when fd reaches end of file    */

int eof_line_by_line(int fd);
   /*  finishes callbacks flushing any remaining input. */
   /*  that is it calls the line callback on any        */
   /*  remaining characters and then the eof callback   */

int cancel_line_by_line(int fd);
   /*  finishes without flushing  */
```

```
/****************************************/
/*                                      */
/*      mess.h                          */
/*                                      */
/*      Alan Dix                        */
/*                                      */
/*      print to tty and transcript     */
/*                                      */
/*      8th January 1990                */
/*                                      */
/****************************************/


void  init_mess( char *transcript );
    /*  sets up the file names by transcript    */
    /*  as the log file for debugging messages  */


void  mess( char *fmt, ... );
    /*  a printf style call                       */
    /*  prints the message on standard error and  */
    /*  also copies to the log file if one has    */
    /*  been set up                               */
```

```
1   /****************************************/
2   /*                                      */
3   /*      monitor.h                       */
4   /*                                      */
5   /*      Alan Dix                        */
6   /*                                      */
7   /*      command interpretter to         */
8   /*      monitor conference server       */
9   /*                                      */
10  /*      11th January 1990               */
11  /*                                      */
12  /****************************************/


13  /*  the programmer sets up a table called 'monitor_tab'    */
14  /*  which is a list of command names and corresponding     */
15  /*  functions.  When perform_line is called it treats      */
16  /*  the line it is given as a command line and             */
17  /*  interprets the first word using the table.             */
18  /*  The appropriate function is called with the rest       */
19  /*  of the line and an identifier (from the table)         */
20  /*  as arguments.  See server.c for an example of its use  */


21  typedef int (*monitor_fun)(int id, char*command, char* buff);

22  struct  mon_tab_struct {
23      int   id;
24      char *name;
25      monitor_fun f;
26      char *help;
27      };

28  extern  struct mon_tab_struct monitor_tab[];


29  extern perform_line(char *line);
```