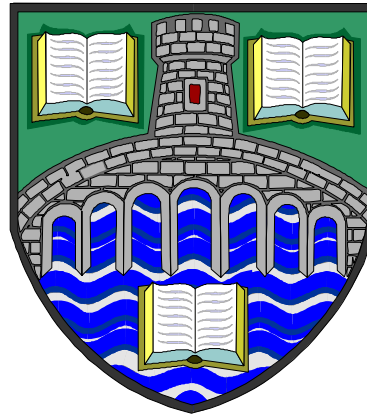


# ***From Persistence to Mobility***

Alan Dearle  
University of Stirling  
Scotland  
[al@cs.stir.ac.uk](mailto:al@cs.stir.ac.uk)



# ***Mobile users non-mobile applications***

- Many users make use of more than one machine:
  - e.g. at work, at home, between offices etc.
- Modes of working:
  - **Store (passive) data on a file-server**
    - » Communication cost fetching data, exploits local CPU, synchronization problems
  - **Work on a file/CPU server**
    - » Communication cost via user I/O, doesn't exploit local resource
  - **Work on a workstation/PC and move data using floppies etc.**
    - » Consistency problem
    - » Availability a problem
  - **Work on a portable**
    - » introduces a new problem - machine mobility.
    - » Availability a problem



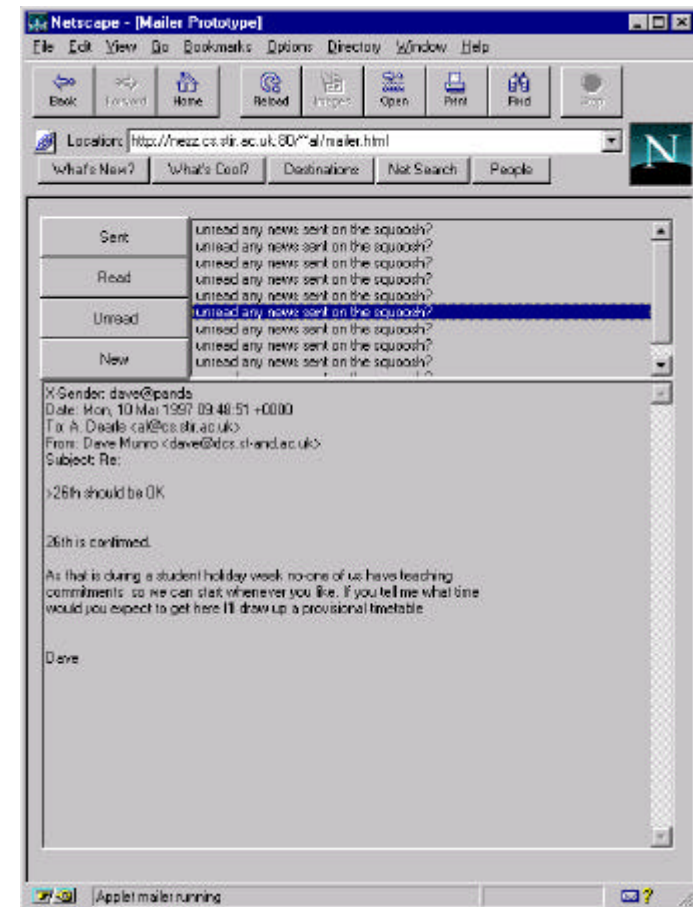
# *An example*

- Reading electronic mail:
- Common options:
  - Use a tool like *mailtool*, *elm* etc:
    - » This is a server solution, data is
      - read,
      - stored,
      - manipulated on the server.
  - Use a tool such as Eudora:
    - » This is a Workstation/PC solution
    - » Data is:
      - read,
      - stored,
      - manipulated on the workstation
  - But at a cost... the data gets *trapped* in the Eudora process..



# Can you have the best of both worlds?

- We built a ubiquitous mailer: *Granny mail*.
- Runs under a browser
- Written in Java
- Mailer asynchronously explicitly snapshots state back to a persistent server process machine running under Grasshopper
- Persistence is provided by Grasshopper
- Grasshopper cgi program runs pop and smtp
- All communication between client and server is via HTML - *the Web is the platform*
- When you log-on your last mailer session is restored
- Can you do this automatically with any process?



# *Objectives of Grasshopper*

- Conventional operating systems do not provide an ideal platform for the construction of persistent systems
- In Grasshopper we set out to provide an ideal environment for the construction of persistent application systems.
  - Support persistent objects as the basic abstraction.
  - Objects must be both stable and resilient.
- We assume standard hardware - only virtual memory management hardware is assumed.
- We wanted to make processes persistent to provide fault tolerance.
- We wanted to be able to use standard languages and compilers with no changes to existing compilers.
- We wanted applications to run in a Grasshopper environment with minimal changes.



# ***Grasshopper Basic Abstractions***

- The Grasshopper operating system is constructed using 3 basic abstractions:

## **Locus**

An abstraction over processes.

## **Container**

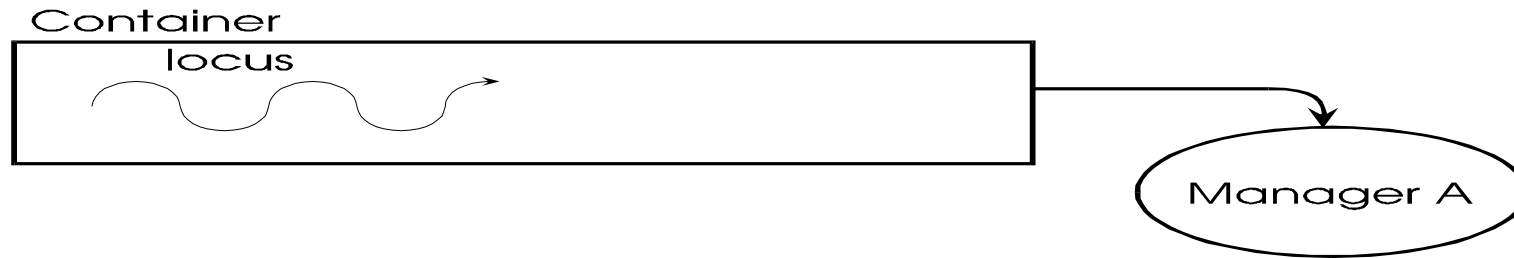
An abstraction over (all) memory.

## **Capability**

A naming & protection abstraction.



# Grasshopper Conceptual View



- Conceptually all loci execute in a container – their *host container*.
- Multiple loci may execute in a container
- Loci can only access data visible in their host container.
- Containers are persistent entities.
- Data in containers is supplied by managers.
- Managers maintain a consistent and recoverable copy of the data stored in containers



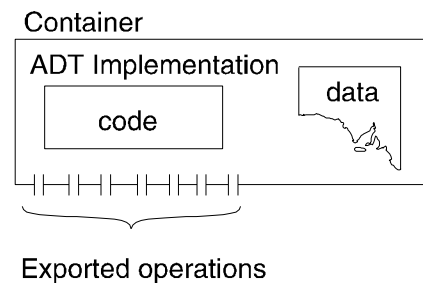
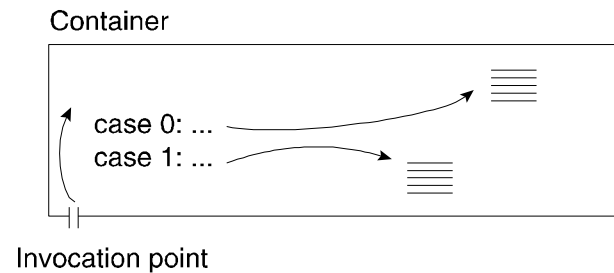
# *Loci*

- The Grasshopper computational model is procedure orientated.
- Loci are migratory, they move from container to container by **invoking** them.
- Each container may have a single entry point called an invocation point.
- When a locus invokes a container it begins execution at this point.
- A locus may invoke and return through many containers.
- Parameters consisting of data and capabilities may be passed on an invoke operation.



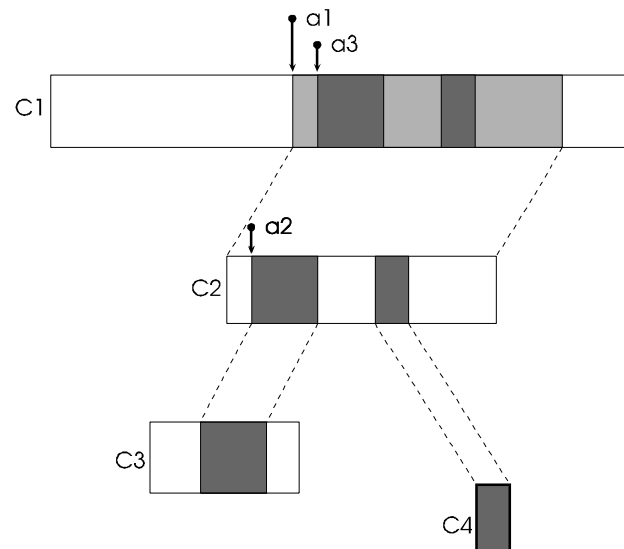
# Containers as ADTs

- The code executed on invocation is supplied by the container being invoked and the container can contain code and data.
- The invocation mechanism provides a building block for hardware protected ADTs.



# Mapping

- The purpose of mapping is to allow the data in a region of one container to appear as if it resides in another container.
- Any locus executing within a container onto which another container has been mapped perceives data from the mapped container within the mapped address range.
- Since any container may have another mapped into it, it is possible to construct a hierarchy of container mappings

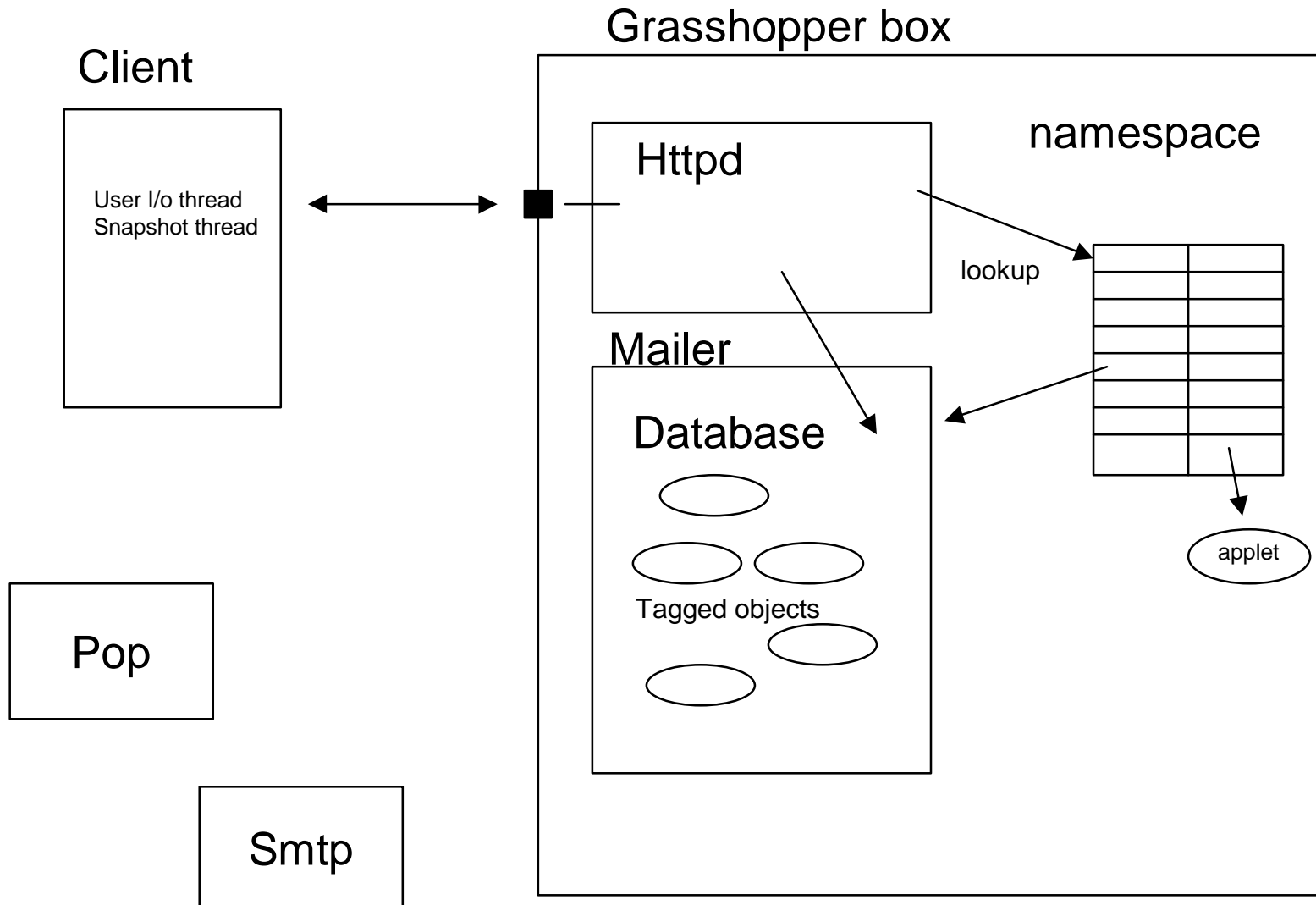


# Persistence

- Containers and managers provide the abstraction of stable persistent memory: orthogonal persistence provided by the operating system
- Managers are responsible for maintaining a consistent and recoverable stable copy of the data contained in the containers(s) they are managing.
- Each manager must provide a snapshot operation.
- Snapshot is locus based and is lazy.
- It involves making a self consistent copy of all modified data viewed by a locus since the last snapshot.
- Snapshot is orchestrated by the kernel and performed by managers.
- By themselves containers cannot maintain a system wide consistent state.
- The kernel uses a page base causality mechanism to ensure that global causal consistency is maintained.

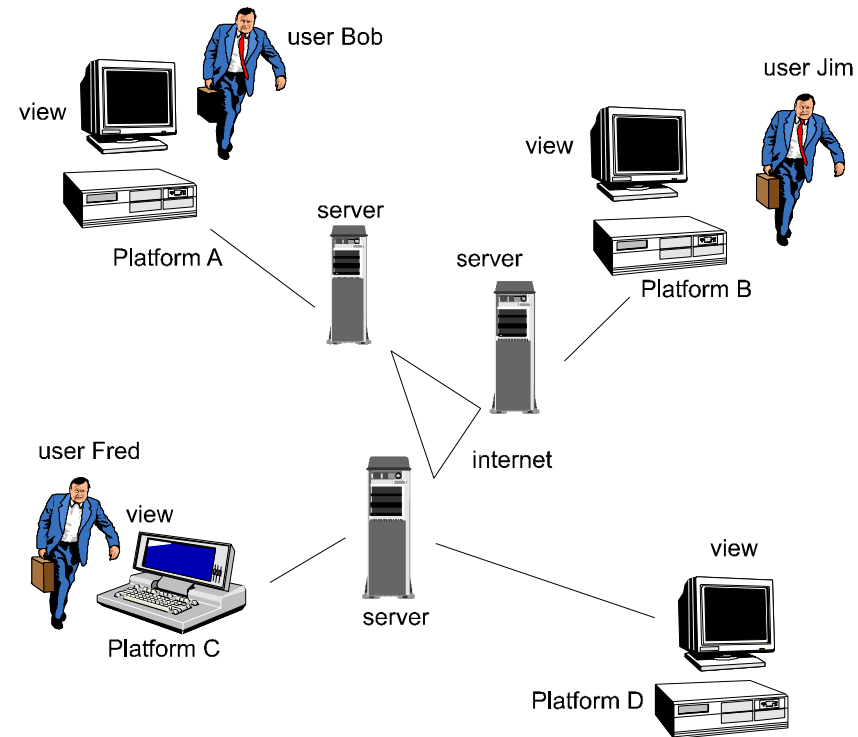


# Architecture of the mailer

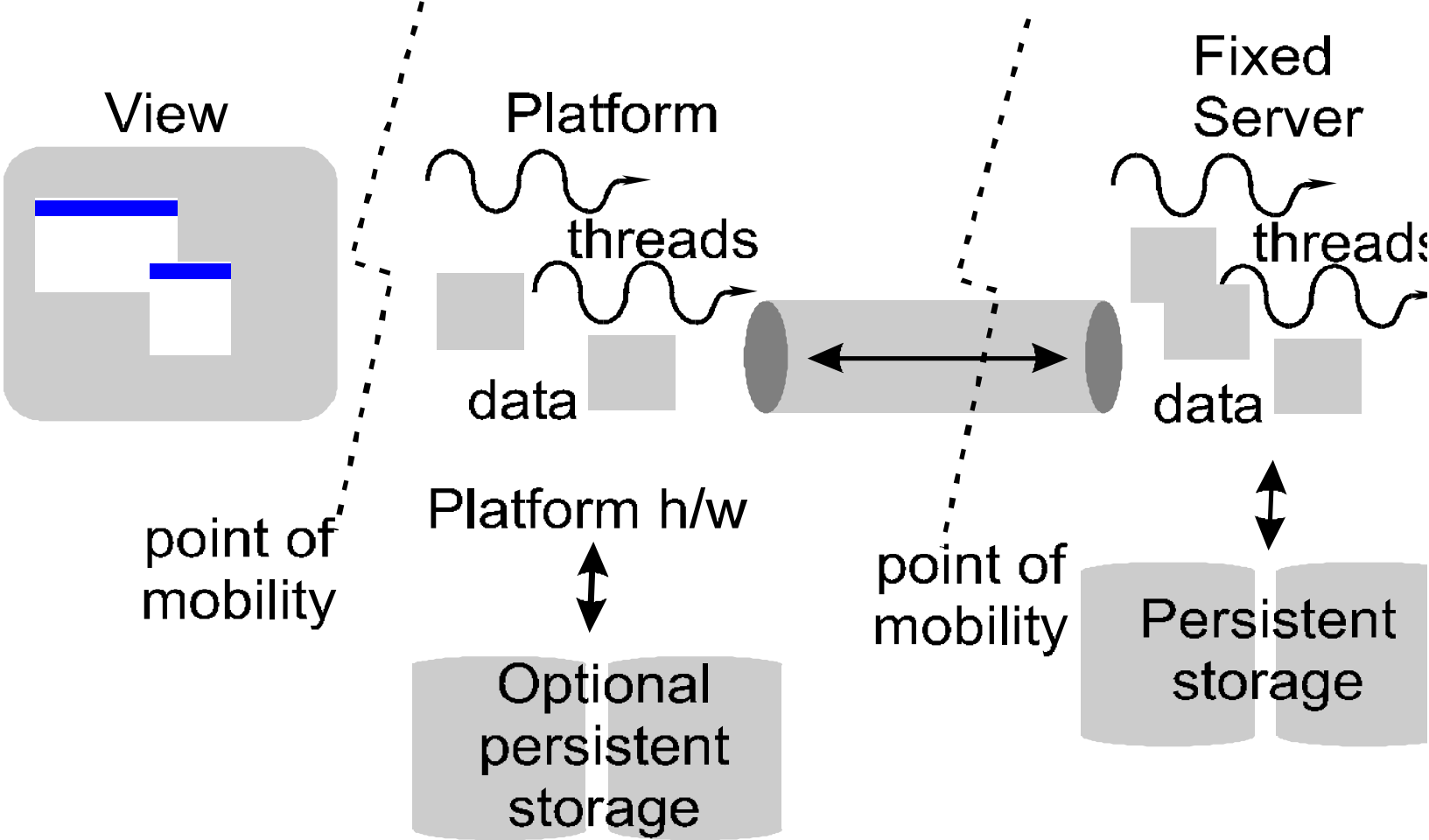


# *What would we like to be able to do?*

- Three classes of entity which may be mobile:
  - users
  - views
  - platforms



# Points of mobility



# *Requirement for User mobility*

- When the user moves, the user's view should also move, permitting the user to continue with whatever work was being performed at the last platform.
- Can either move:
  - threads implementing a view
  - or merely the view (Olivetti Teleporting approach)
- Trade off:
  - use of local CPU
  - network bandwidth
  - latency
  - complexity
- We assume the former.



# *View Migration*

- Migrating the view requires two forms of migration:

1. data migration, and,

2. thread migration,

or rephrased,

view mobility = thread mobility + data mobility.

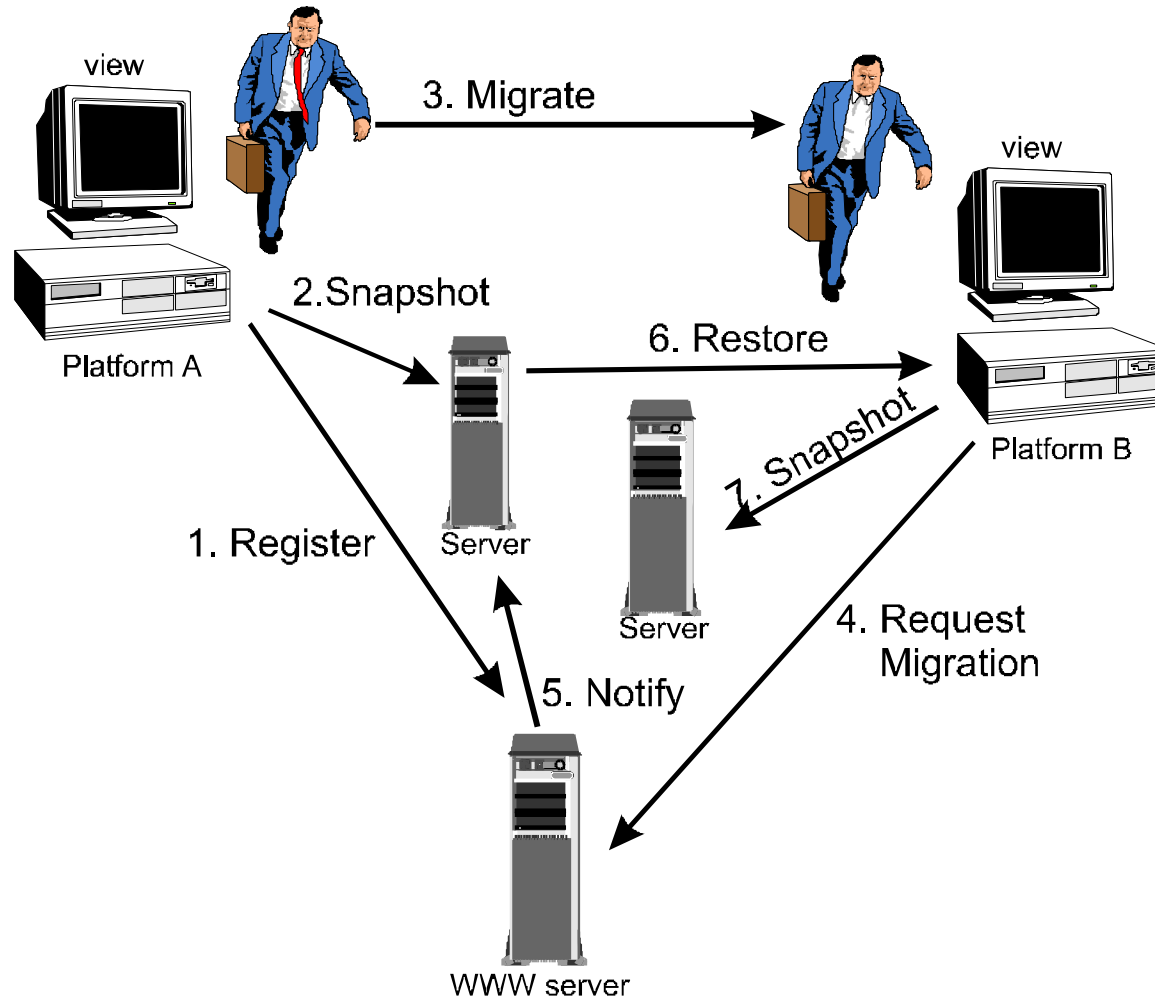


# *Requirements for Platform mobility*

- The mobility of platforms highlights some other problems, namely:
  1. Environment Mobility
    - » Environment mobility is concerned with bindings between threads and the external environment.
    - » A thread may make use of a printer or some input device. Mechanisms must be provided so that threads running on a mobile platform may bind to services which appear in their environment.
  2. Channel Mobility
    - A thread running on a platform may open a communications channel with a another thread running on another platform or server.
    - To make movement transparent, software that maintains the channel across movement must be provided. This may be achieved in one of two ways:
      1. Implementing software at both ends of the channel to manage the transparent connection/reconnection.
      2. Using a server as a connection proxy.



# Overall architecture



- Each user is assumed to have a home capable of recording the identity of the last hardware platform on which each view was last made.



# Platforms

- A platform must be capable of:
  1. authenticating a user,
  2. loading a view from the platform/server identified during the authentication sequence, and
  3. saving the state of the view to persistent storage provided by the platform or the server that supports it.
- The second and third activities are intimately related to each other and require a protocol that defines:
  1. how to identify the persistent state implementing a view,
  2. format of code,
  3. how persistent state is preserved \*\*\*
  4. what format the persistent data is in, and
  5. how to transport that state to and from persistent storage and between platforms and servers.



# ***Code formats***

- Java code is already machine independent and portable thus making it a good choice.
- Possible to perform *just in time* compilation.
- We are investigating intermediate low level '*RISC style*' languages which can be compiled *just in time*: Liquid Software from Arizona
- If this approach is followed need a bus-stop concept to identify state of computation(s) i.e. need some kind of symbolic PC



# Preserving State

- There are 4 approaches to saving state:
  1. Manually writing save and restore code in every application/applet,
    - » this is what Granny mail does - not general but works
  2. Perform saving and restoration using (Java) serialisation,
    - » Problems with not everything being saved
    - » Problems with granularity of snapshot
    - » Problems with threads and referential integrity
  3. Providing persistence at the (Java) virtual machine level, and
    - » Looks like the best good approach but some problems...
  4. Providing persistence at the address space level
    - » Approaches such as Wilson's page swizzling look promising



# *Problems of Preserving state (1)*

- Finding data on stacks, heaps and in registers.
- This is one level more difficult than required for garbage collection
- Not just pointers and non-pointers:
  - Reals
  - booleans
  - Pointers
  - byte order
- Approaches to stack analysis:
  - Tagged stack architecture (expensive)
  - Static map of stacks produced at compile/translation time
    - » Complexity of map
  - Produce code with knowledge of stack locations (N88/C approach)



# ***Problems of Preserving state (2)***

- Can this be done for an arbitrary language?
  - Object formats
  - Unions
  - Degree of type safety of source language
  - Degree of semantic information available at run-time



# *Problems of Preserving state (3)*

- Synchronous versus asynchronous checkpoint:
  - In systems supporting multiple threads how do you ensure a consistent snapshot
  - Need to either force threads to snapshot together
    - » This in turn limits approaches to snapshot
    - » e.g. hard to see how this fits with self saving code approach
  - or use some form of consistency regime
    - » adds considerable complexity to system

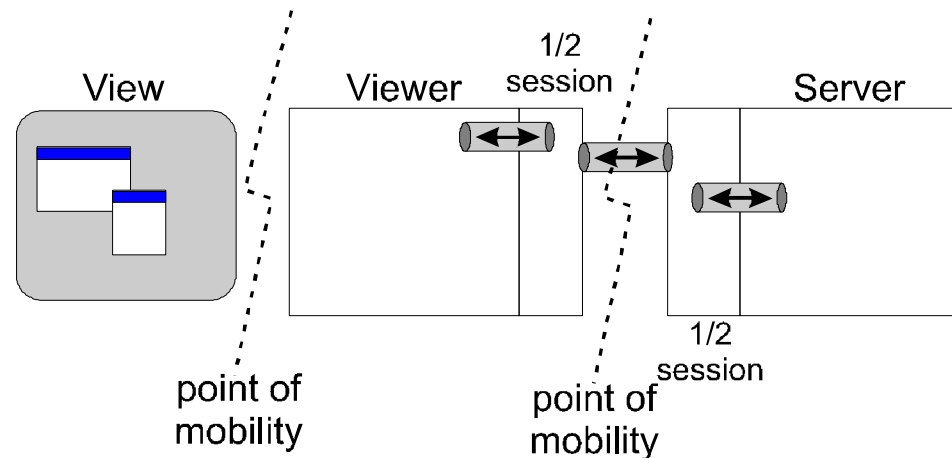


# *Servers*

- Servers are responsible for four tasks in the architecture:
  1. Implementing the home of users,
  2. Providing persistent storage for non-persistent client platforms,
  3. Providing channel proxies, and
  4. Provide caches for client platforms.



# Channel and Environment Mobility



- In order to accommodate the connection and reconnection of channels we introduce a new abstraction called a *half session*.
- The primary purpose of a half session is to implement a communication channel which provides a reliable stream abstraction that can be disconnected and reconnected to different platforms and servers.
- On each platform or server implementing a relocatable channel, a half session is used to manage the connection.
- Thus there is a half session managing each end of a relocatable channel.



# Channel Proxies

- Views may be required to communicate with legacy systems which do not implement the channel abstraction.
- This is the case where a user is interacting with a legacy application running on a Unix system, for example a shell.
- Since legacy code does not support the half session abstraction, some additional mechanism must be provided to permit the platform (hardware or software or both) to migrate.
- This may be achieved by the use of the server as a fixed proxy for communication.
- Using this scheme, the fixed server communicates with the remote party on behalf of the platform.
- This communication is achieved using a traditional socket interface.
- The platform in turn communicates with the server using the half session abstraction permitting the platform to be relocated without the knowledge of the remote party which is only aware of the server.



# *Binding to Services*

- A final aspect of mobility that must be addressed is binding to the external environment.
- Binding to external services may either be dynamic or static
- For example, an application running on a platform may wish to make use of a printer.
- In other circumstances, for example when a user wishes to make use of a file/object server, only the file server containing the user's files would be appropriate.
- In both cases the external services are provided by servers, the only issue is how platforms bind to the servers.
- Clearly, if either the hardware or software platform is permitted to migrate, some indication of the (re)binding regime must be specified when the binding to the service is initially established.



# *Conclusions*

- We aim to produce a ubiquitous environment that moves around with mobile users
- Aim to accommodate architectural heterogeneity
- Don't want programmers to have to write special code e.g. Aglet approach
- This will be based on:
  - Internet protocols
  - probably Java
  - persistent technologies
- We have also written a small demo program that uses Grasshopper to investigate the issues
- We have identified several problems
- We have identified several solutions
  
- A long way to go.....

