Providing Flexibility in Distributed Applications Using a Mobile Component Framework

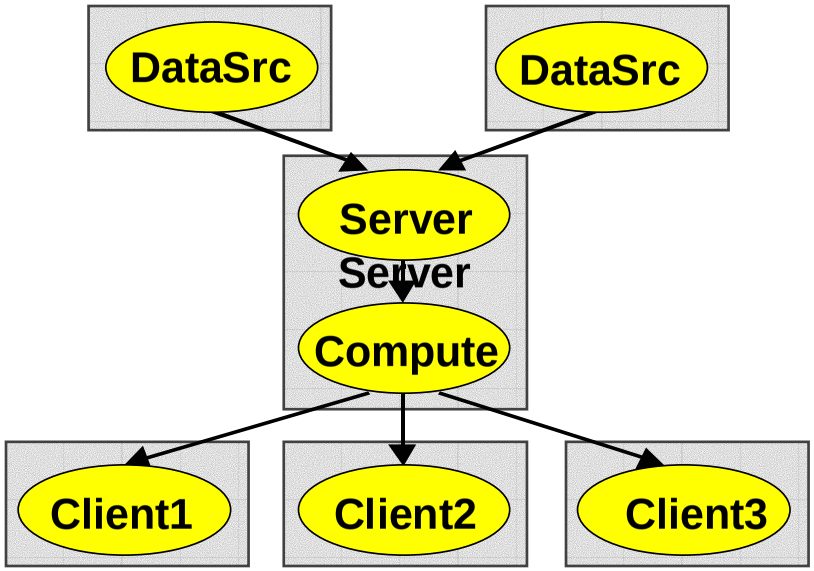
Radu Litiu

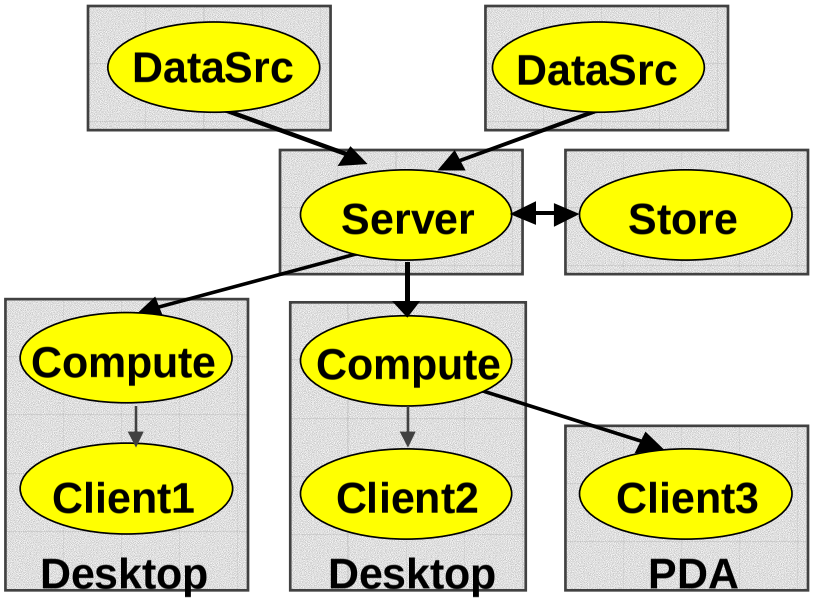
University of Michigan
Electrical Engineering and Computer Science

Outline

- Motivation and Design Goals
- Related Work
- System Architecture
- Component Mobility
- Dynamic Application Reconfiguration
- Building Adaptive Applications
- Summary and Future Work

Reconfiguration Need

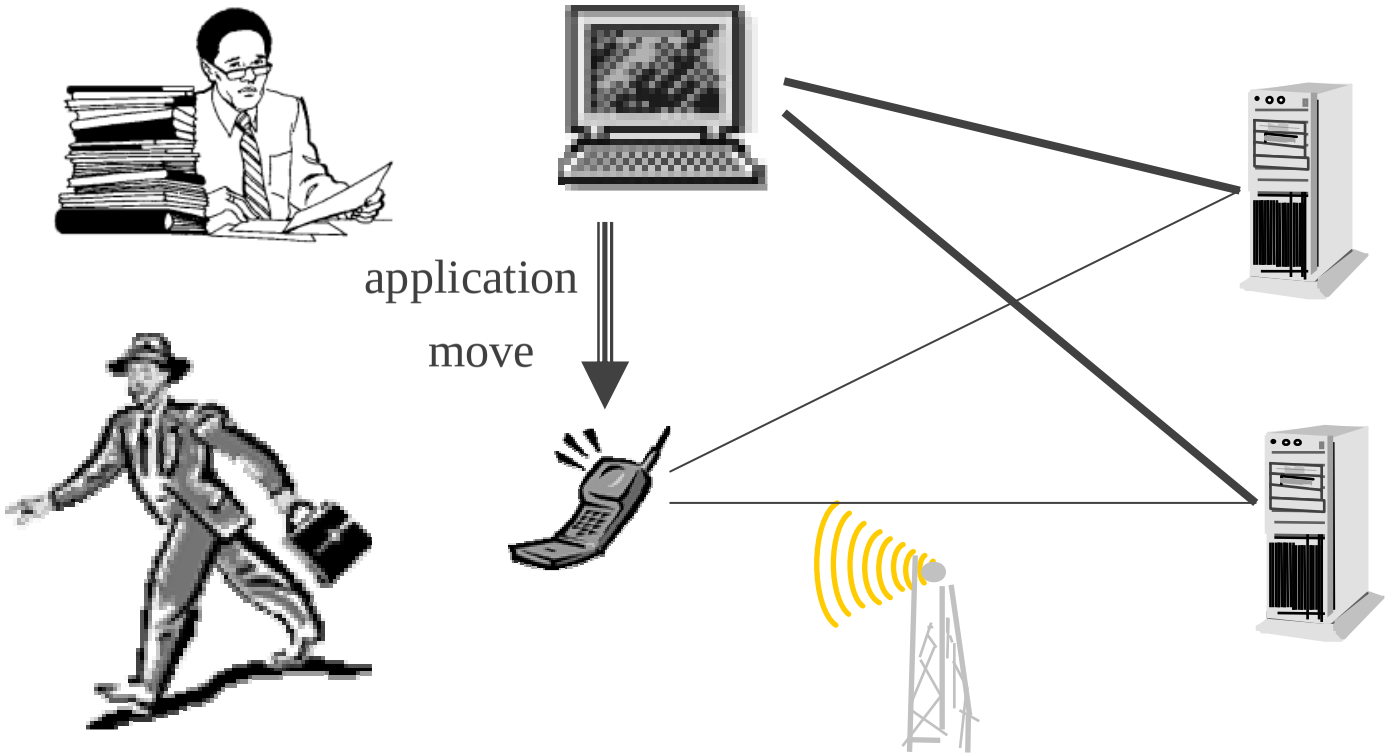




- Variable load
- Variable resources
- Latency important

- Change the architecture of the application
- Do not know a priori which architecture is better

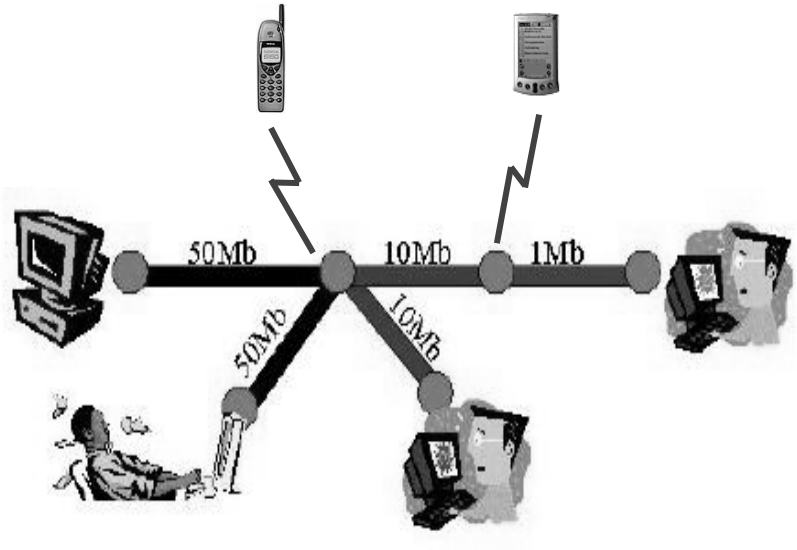
Application and User Mobility



- Device heterogeneity
- Persistent connectivity during application moves

Goals

- Variability and heterogeneity
  - User and application demands
  - Hardware and network variability
- Continuous operation



- Performance
- Mobility
- Cost of application maintenance and upgrade

Component-based framework that supports:

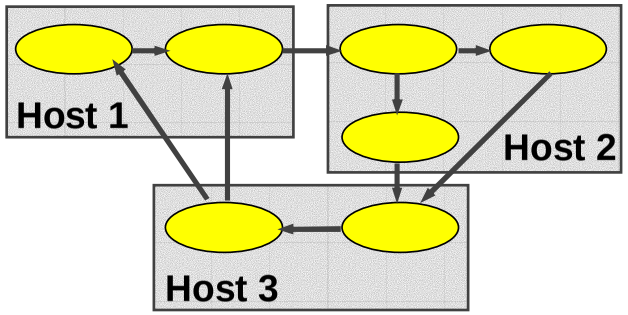
- Dynamic reconfiguration
- Component mobility and persistent connectivity

Related Work

- Modularity and application decomposition: Horus, Ensemble, x-kernel, Coyote, Scout
- Distributed component architectures: CORBA, DCOM, Rover, FarGo
- Dynamic application configuration: configuration programming (Darwin, Conic, Rex – Kramer, Magee et. al.), PCL – version description
- Code mobility & mobile agents: Telescript, Obliq, Sumatra, Tacoma, Mole, Ajanta, Aglets (IBM), Odyssey (General Magic), Voyager (ObjectSpace)
- Adaptive systems: Odyssey, Daedalus/BARWAN/MASH, Conductor

Contributions

DACIA* features:



- Component-based framework
- Dynamic application reconfiguration
- Component mobility
- Persistent connectivity between components
- Support for offline operation
- Benefits of modularity without significant performance degradation

*Dynamic Adjustment of Component InterActions

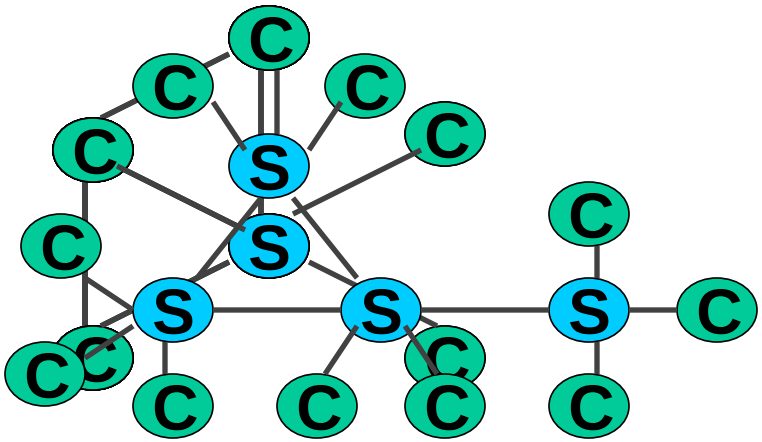
Outline

- ✓Motivation and Design Goals
- ✓Related Work
- System Architecture
- Component Mobility
- Dynamic Application Reconfiguration
- Building Adaptive Applications
- Summary and Future Work

An Adaptive Application

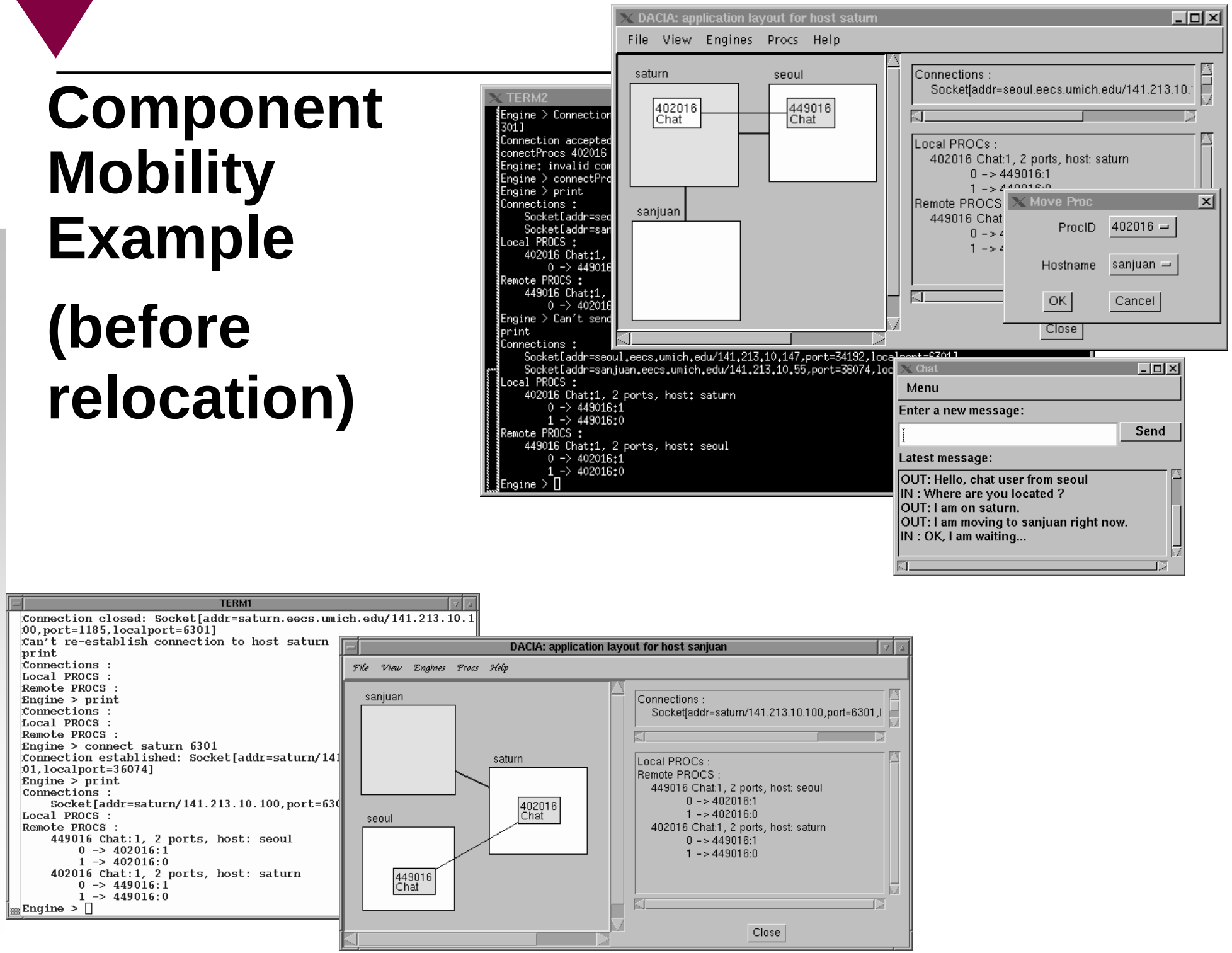
Multi-party communication (implemented in DACIA)

- Application dynamics
- Scalability
- Proximity of clients and servers
- Automated reconfiguration

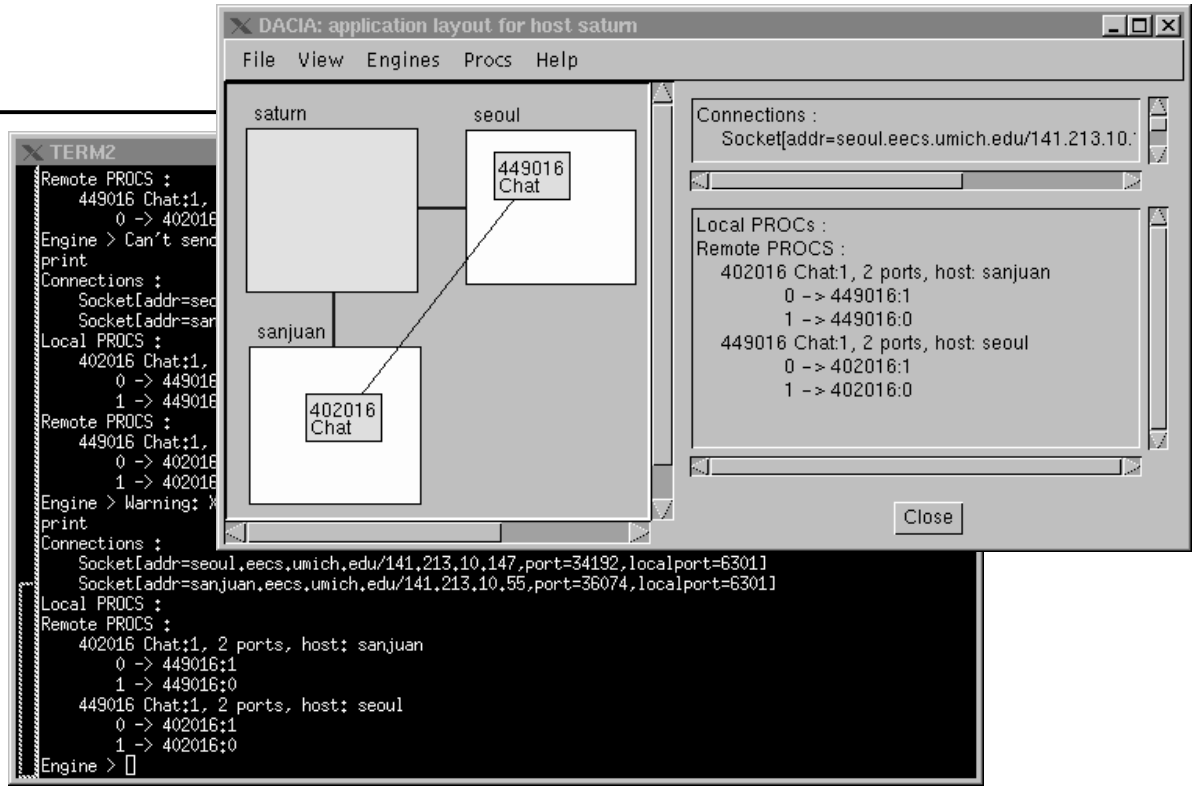


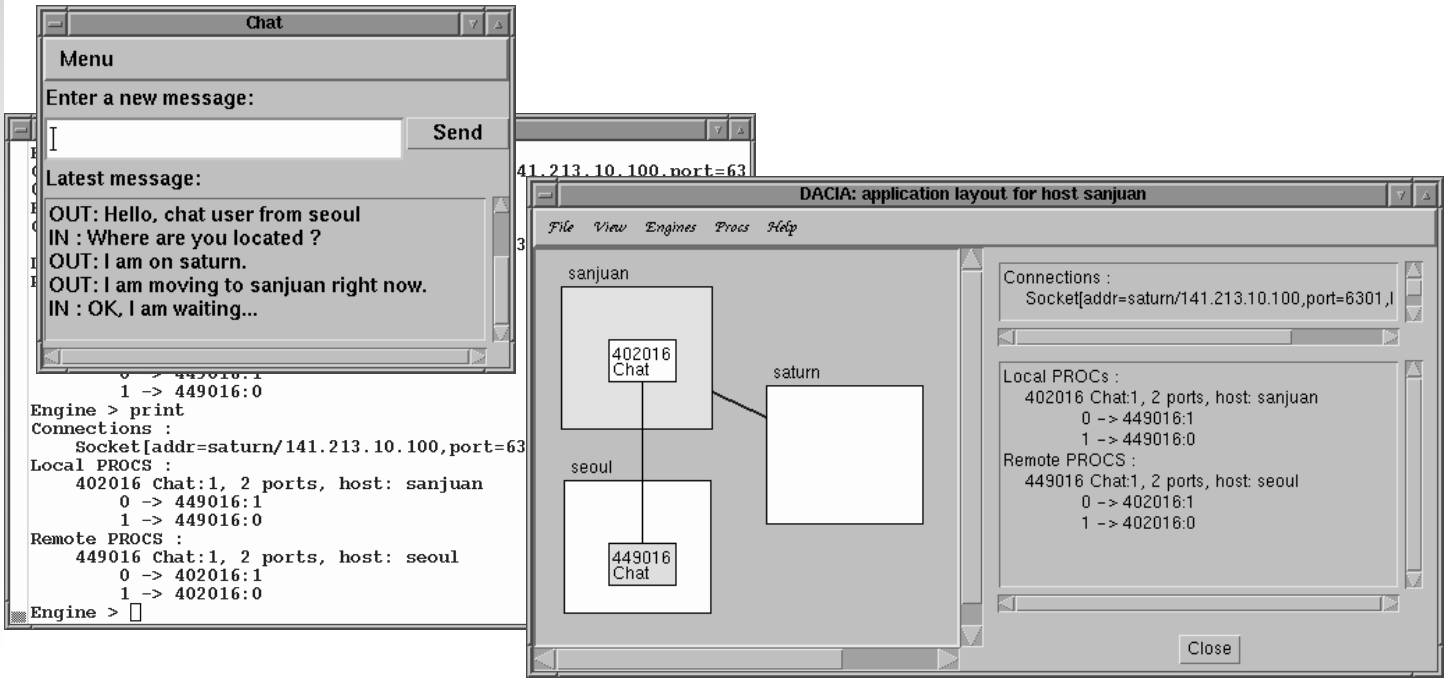
Component Mobility Example

(before relocation)

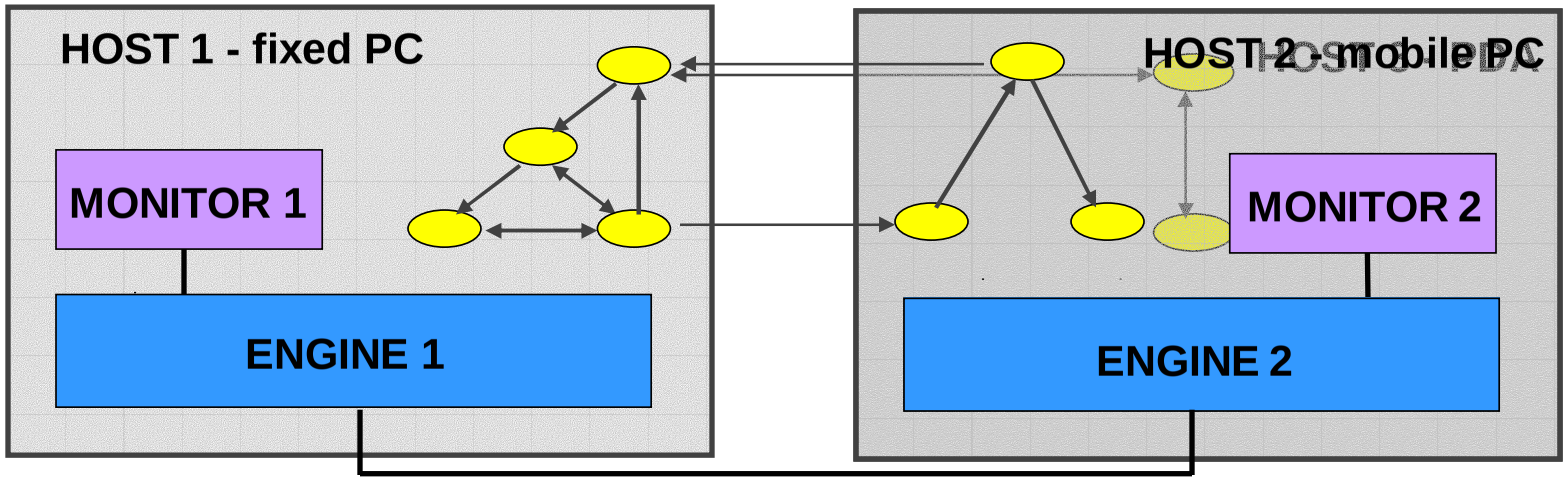


Component Mobility Example (after relocation)





DACIA Architecture



Engine (mechanism)

- Communicate between hosts
- Manage connections between components
- Relocate components
- Reconfigure the application

Monitor (policy)

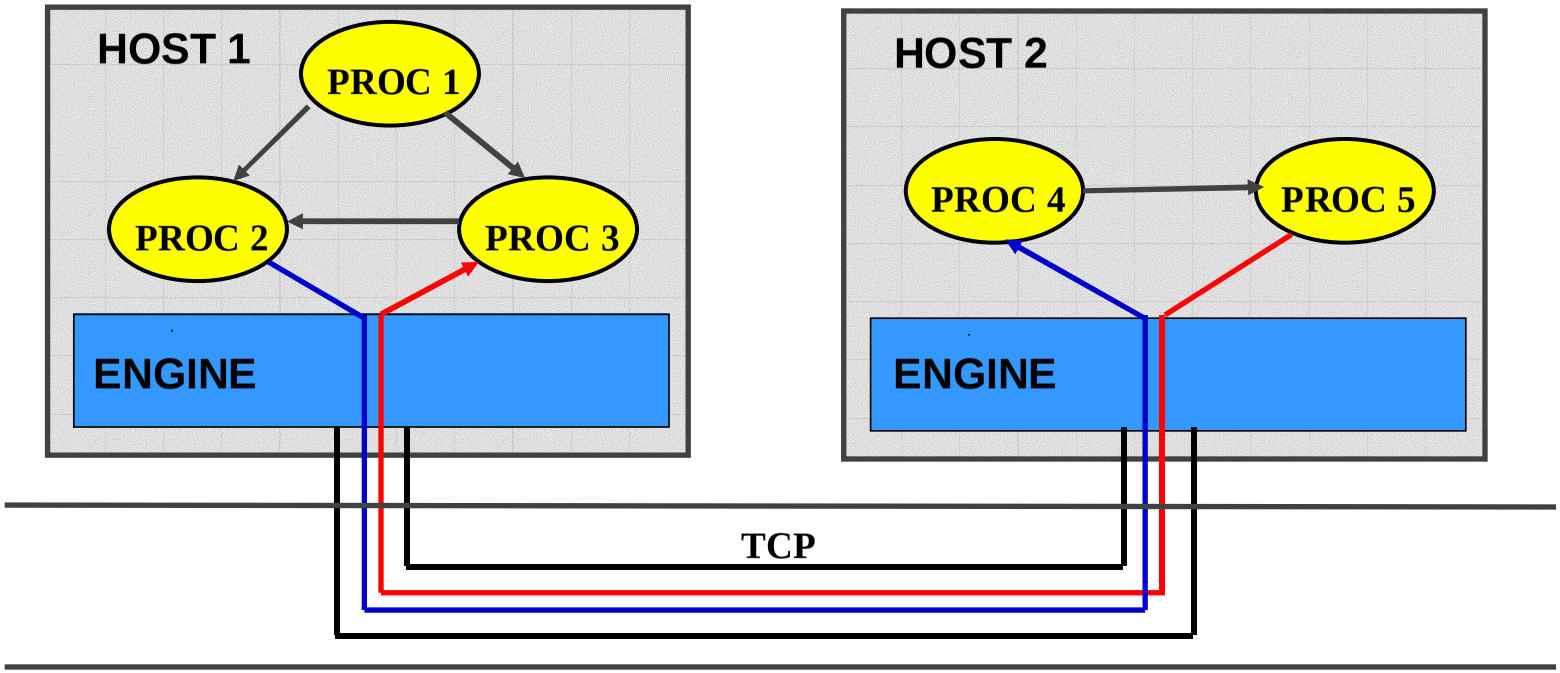
- Monitor performance
- Make reconfiguration decisions
- Implement application-specific reconfiguration policies

PROCs*

- Message-based communication through ports
- Ports are not typed
  - No syntactic checks
  - Handle wrong data types
- Mobile components
- Unique identifier
- Low communication costs

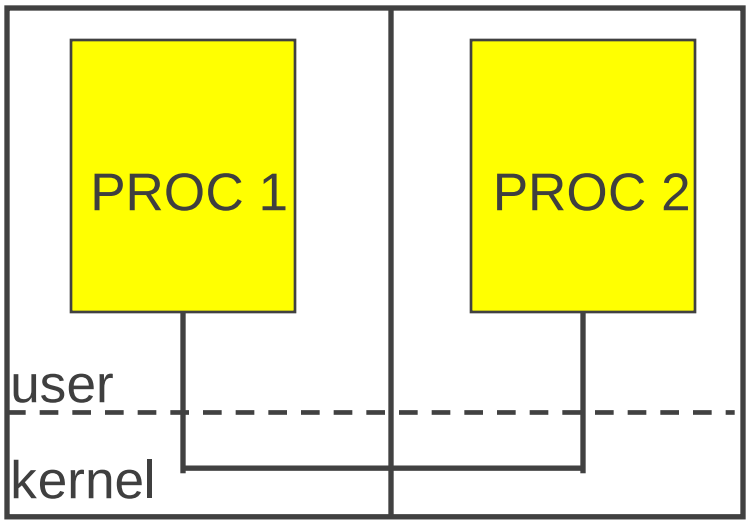
*Processing and Routing Component

Remote Communication

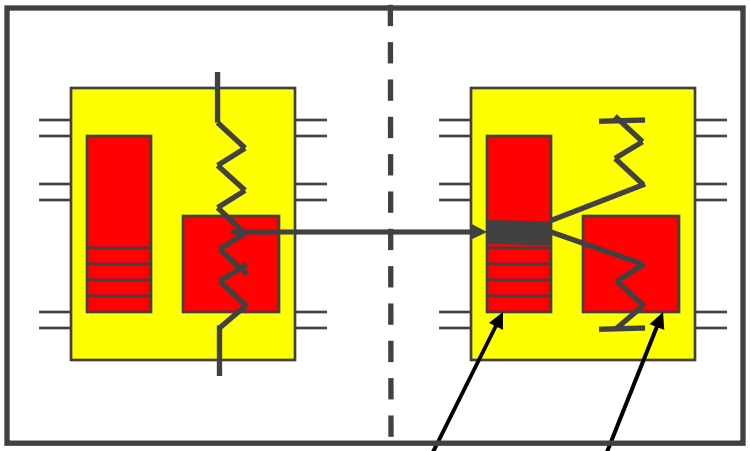


- Multiplex virtual connections between PROCs
  - Low cost to establish connections
  - Hide transient network and communication failures

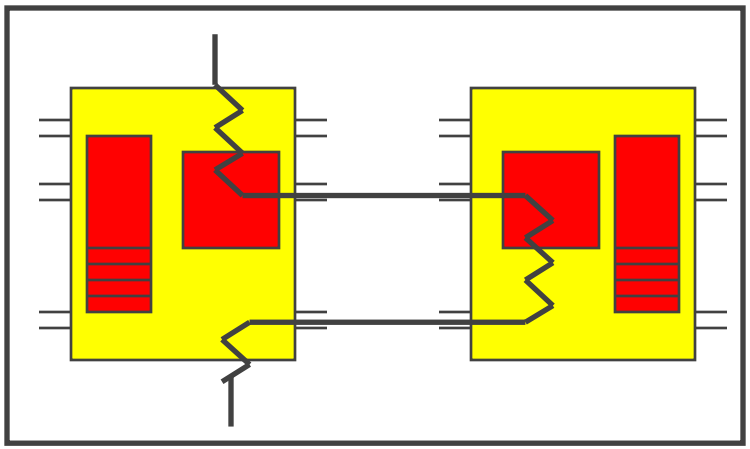
Local Communication



asynchronous communication



synchronous communication



Communication Performance

- Environment: Pentium III 733MHz, 256MB, 100 Mbps LAN
- Roundtrip latencies (µsec) for inter-PROC communication and raw TCP

message size (bytes)	local PROCs synchronous	local PROCs asynchronous	local procedure call	local TCP message	Local TCP byte[ ]
0	1.1	9.8	.18	477	332
1000	1.1	9.8	.18	6867	431

message size (bytes)	remote PROCs	remote TCP message	remote TCP byte[ ]
0	2186	522	375
1000	5226	7232	526

Communication Throughput

Remote communication (message size = 1-10 kB)

- DACIA: 6.87 Mbps
- TCP (Java): 32 Mbps

Local communication

- Synchronous: 1,810,000 messages/s
- Asynchronous: 204,000 messages/s

Component Mobility

- Transfer the PROC's state
  - Code and data
  - Message queue
  - Connections state
- Implicit/explicit state capture
- Movement at well-defined times

- Cost of PROC movement – 4.4 ms (size = 725 bytes)
  - Java serialization cost (1.4 ms)
- Component mobility more effective for long-term environment changes

Moving Algorithm

```
A1  : output(proc, msg) {
A2  :     proc.refcnt++
A3  :     if (proc.moving != HERE) {
A4  :         synchronized (proc) {
A5  :             if (proc.moving == MOVING) {
A6  :                 proc.refcnt--
A7  :                 wait(proc)
A8  :                 remote_output(new_proc,msg)
A9  :             }
A10:             else
A11:                 remote_output(new_proc,msg)
A12:         }
A13:     }
A14:     else {
A15:         proc.input(msg)
A16:         proc.refcnt --
A17:     }
A18: }
```

Initially: proc.moving = HERE
refcnt = 0

```
B1  : move(proc) {
B2  :     proc.moving = MOVING
B3  :     // notify the async thread
B4  :     while (proc.refcnt > 0)
B5  :         sleep(timeout)
B6  :     execute PROC move
B7  :     synchronized (proc) {
B8  :         proc.moving = AWAY
B9  :         notifyAll(proc)
B10:     }
B11: }
```

Persistent Connectivity

- Reliable inter-PROC communication during component relocation
- Maintain virtual connections between PROCs
- Mobility transparent to other components
- Messages buffered and/or forwarded
- Mask network failures
- Disseminate PROC location information

Is DACIA a Mobile Agent System?

PROCs

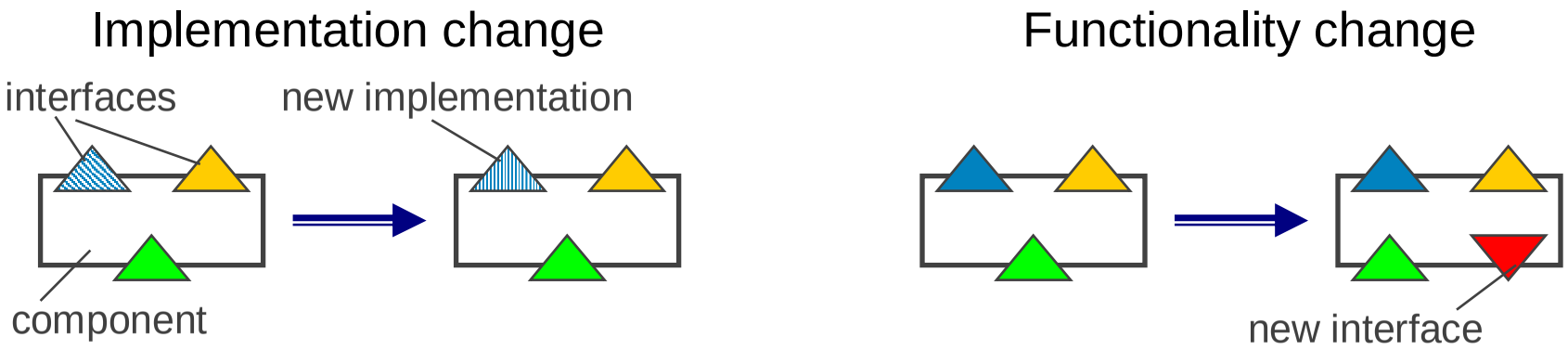
- Not autonomous
- Interconnected and part of a distributed application
- Small state size
- Frequent interactions
- Synchronous and asynchronous
- Engines and PROCs are part of the same trust domain

Mobile agents

- Autonomous, initiate moves by themselves
- Larger state size
- Infrequent, asynchronous interactions

Dynamic Code Loading

- Load PROCs
  - Extend application functionality
  - Introduce new components
  - Replace/upgrade existing components – state transfer

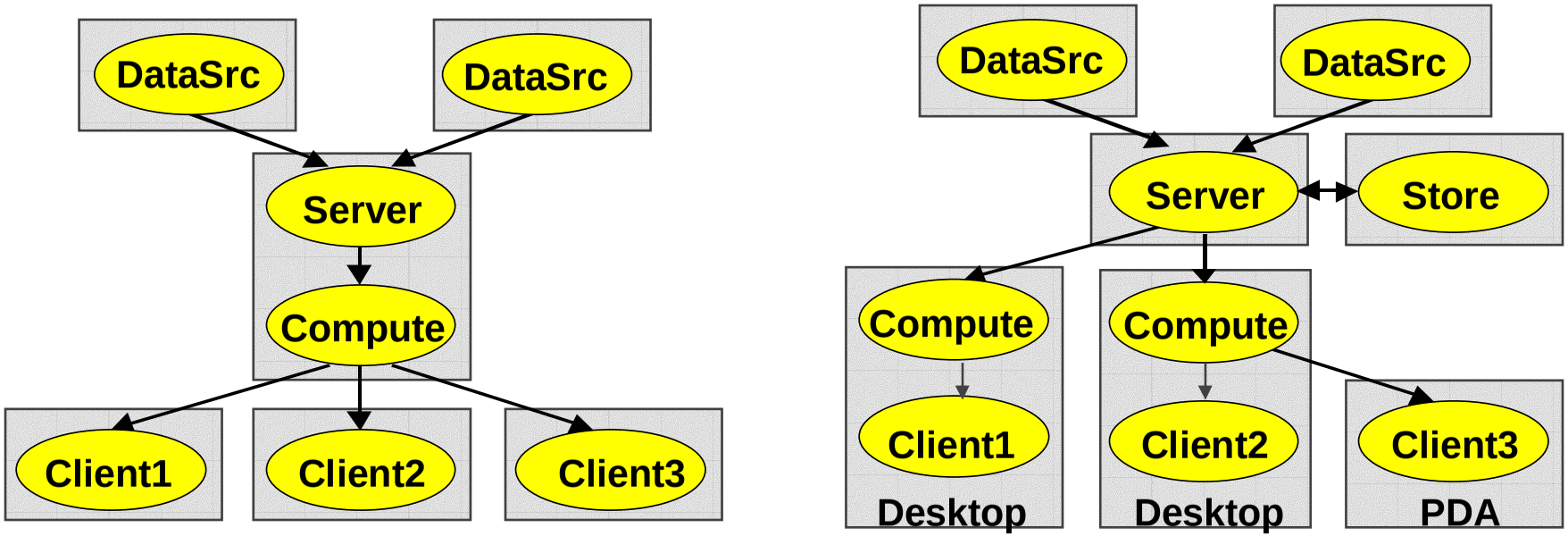


  - Move components (code and data)
- Load monitors
  - Change reconfiguration policies at runtime

Outline

- ✓Motivation and Design Goals
- ✓Related Work
- ✓System Architecture
- ✓Component Mobility
- Dynamic Application Reconfiguration
- Building Adaptive Applications
- Summary and Future Work

Dynamic Application Reconfiguration



- Connect / disconnect components
- Change components' location
- Load new components
- Remove components

Reconfiguration Mechanisms

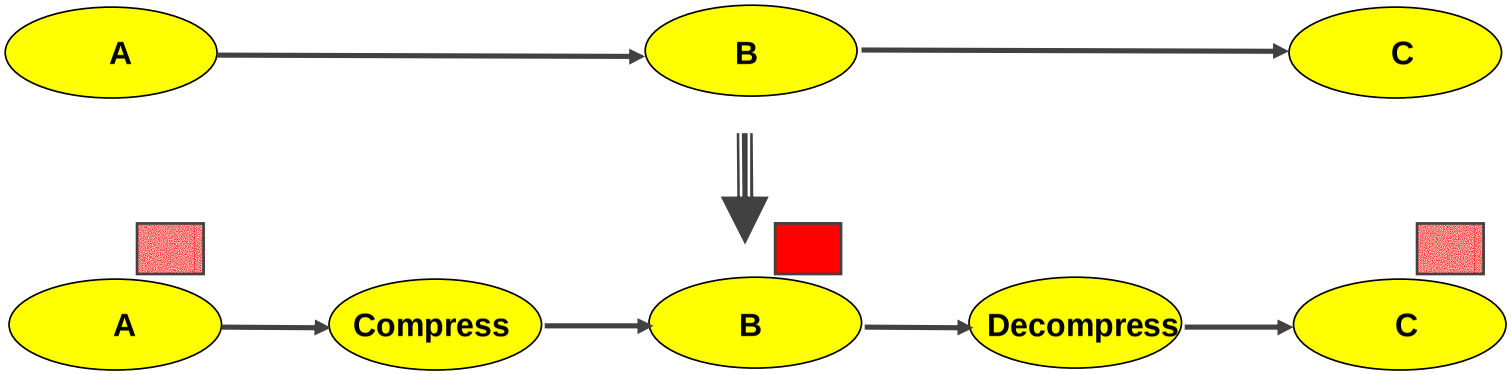
- Programming API :
  - *connect (hostName, portNumber)*
  - *connectProcs (sourceProcID, sourcePortNo, destProcID, destPortNo)*
  - *disconnectProcs (sourceProcID, sourcePortNo)*
  - *moveProc (procID, hostName)*
  - *load (className)*
  - *start (procID)*
  - *startMonitor ( )*
- Specialized monitors
- Command-line interface
- Graphical interface

Reconfiguration Requirements

- Performance
  - Time between reconfiguration request and execution
  - Time to perform reconfiguration
  - Number of control messages exchanged
- Application disturbance - number of components affected by reconfiguration
- Termination
- Automated reconfiguration
- Correctness

Application Consistency

- Component consistency
  - Component state is not altered
- End-to-end application consistency
  - Complete ongoing operations
  - Integrity of data in traffic
- Synchronize reconfiguration with application execution



Performing Reconfiguration

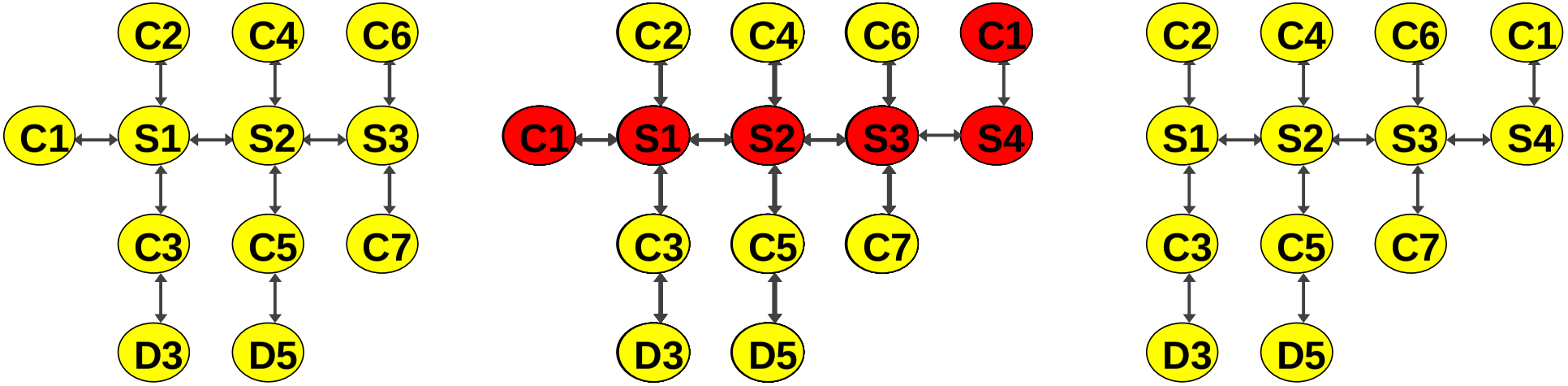
Definitions:

- *Changing set (CS)* : set of components involved in reconfiguration
- *Reactive chain* : sequence of components executing dependent operations
- *Component state* :
  - *Active* – can send or receive messages unrestricted
  - *Passive* - does not receive or send messages; does not have pending messages; all potential senders are passive
  - *Pseudo-passive* - can receive messages only from a specific set of components; may have pending messages

Assumptions:

- All message handling routines complete in bounded time
- Finite reactive chains

Reconfiguration Algorithm



1. Find the changing set CS
   - Components that will be connected/disconnected/removed
2. Block the execution of all components in CS
   - Traverse reactive chains
   - Suspend message handling
3. Flush all messages in traffic among components in CS
   - Block the receiving of new messages
   - Conditional activation of components
4. Execute the reconfiguration
5. Activate all components

Complexity

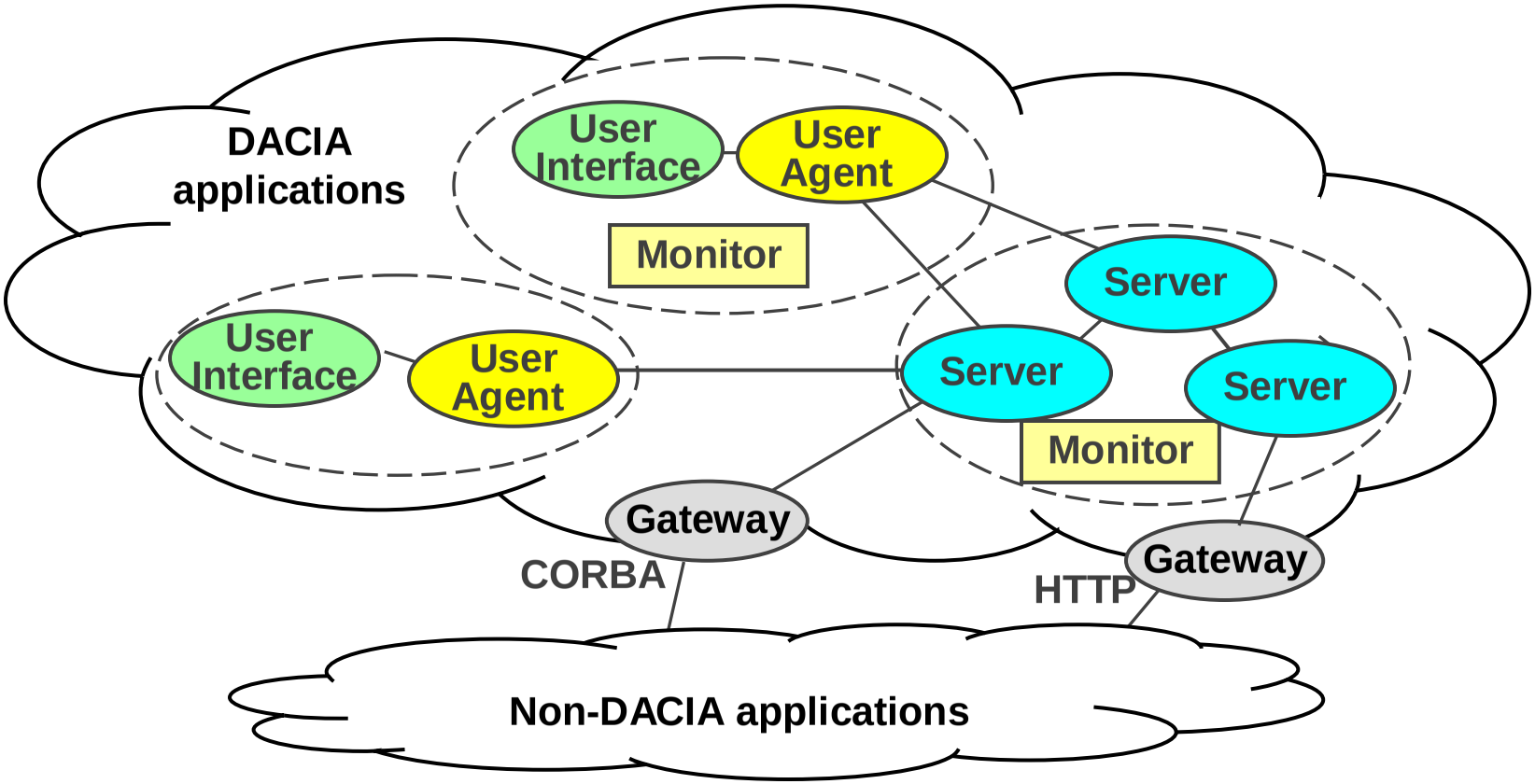
- *n* – number of components
- *m* – number of connections

- Find whether an unmarked component belongs to a reactive chain between two marked components – O(n+m) - breadth-first traversal
- Flush all messages
  - $O(n^2)$ if no cycles
  - $O(k \cdot n^2)$ if there are cycles iterated at most *k* times
  - O(n+m) in most real cases
- Number of control messages exchanged – O(n)

Outline

- ✓Motivation and Design Goals
- ✓Related Work
- ✓System Architecture
- ✓Component Mobility
- ✓Dynamic Application Reconfiguration
- Building Adaptive Applications
- Summary and Future Work

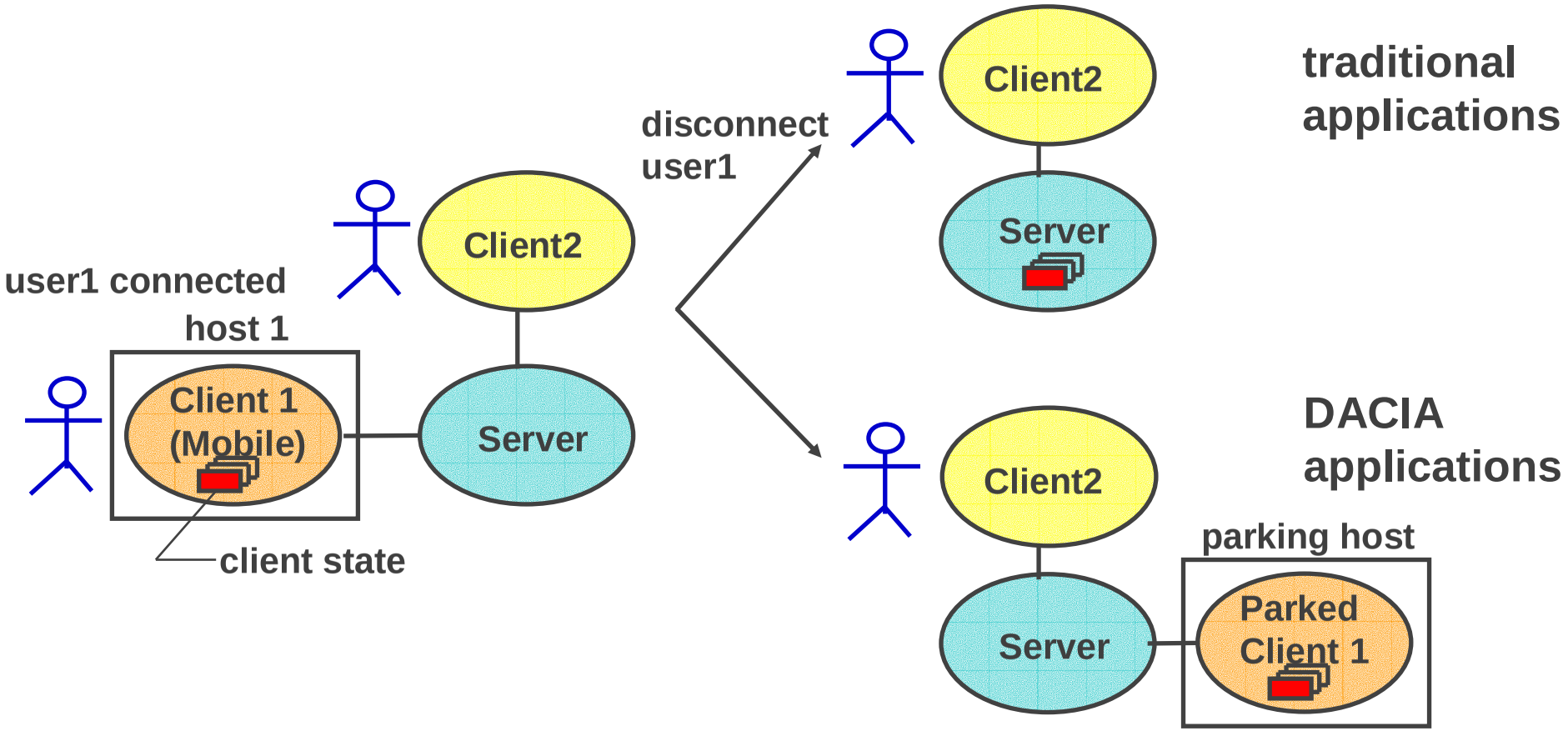
Structuring Distributed Applications



- Types of components
- Separate *User Interface* and *User Agent* code
- Horizontal/vertical decomposition

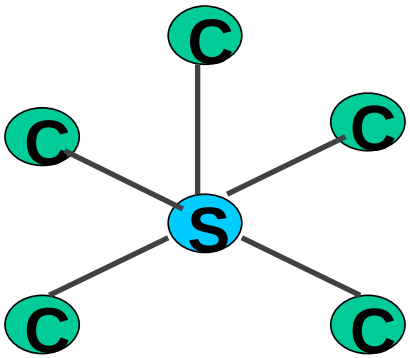
Application Parking

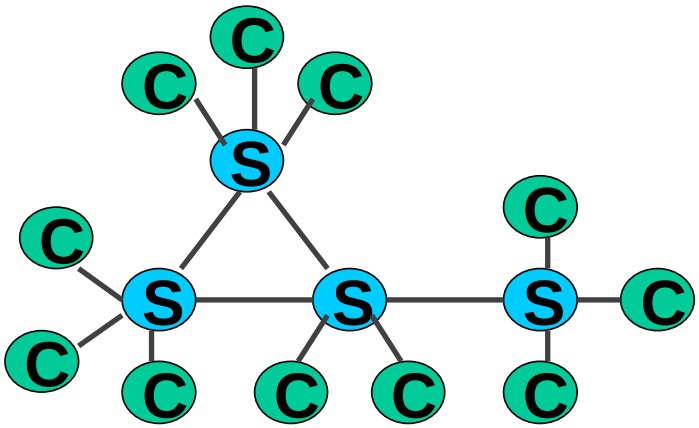
- Intermittent connectivity
- Applications participate to collaboration on the user's behalf while the user is disconnected



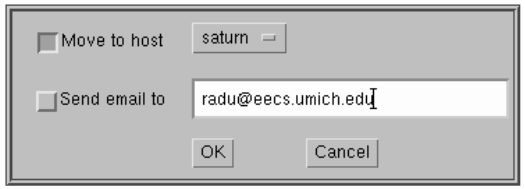
Applications

Multi-party communication

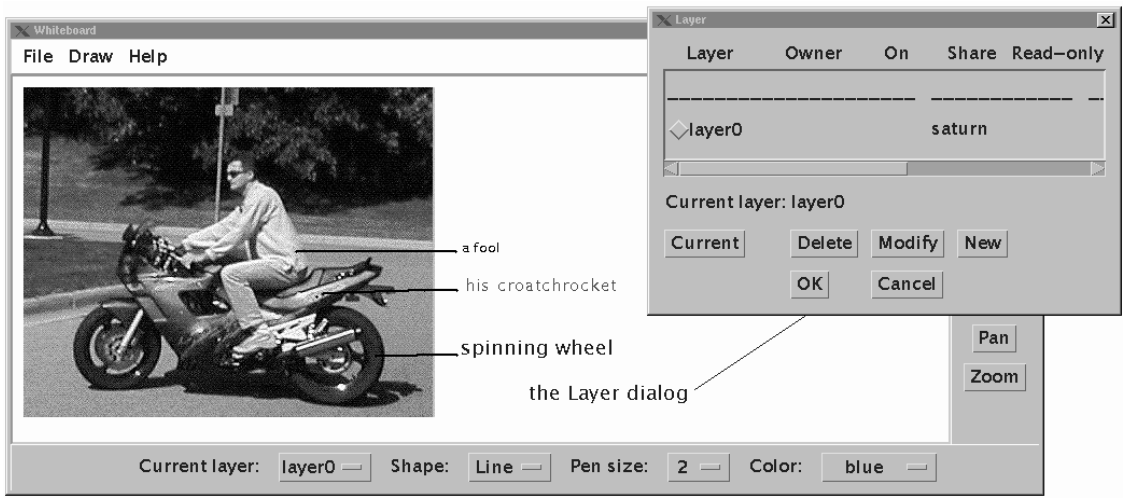




Chat-box

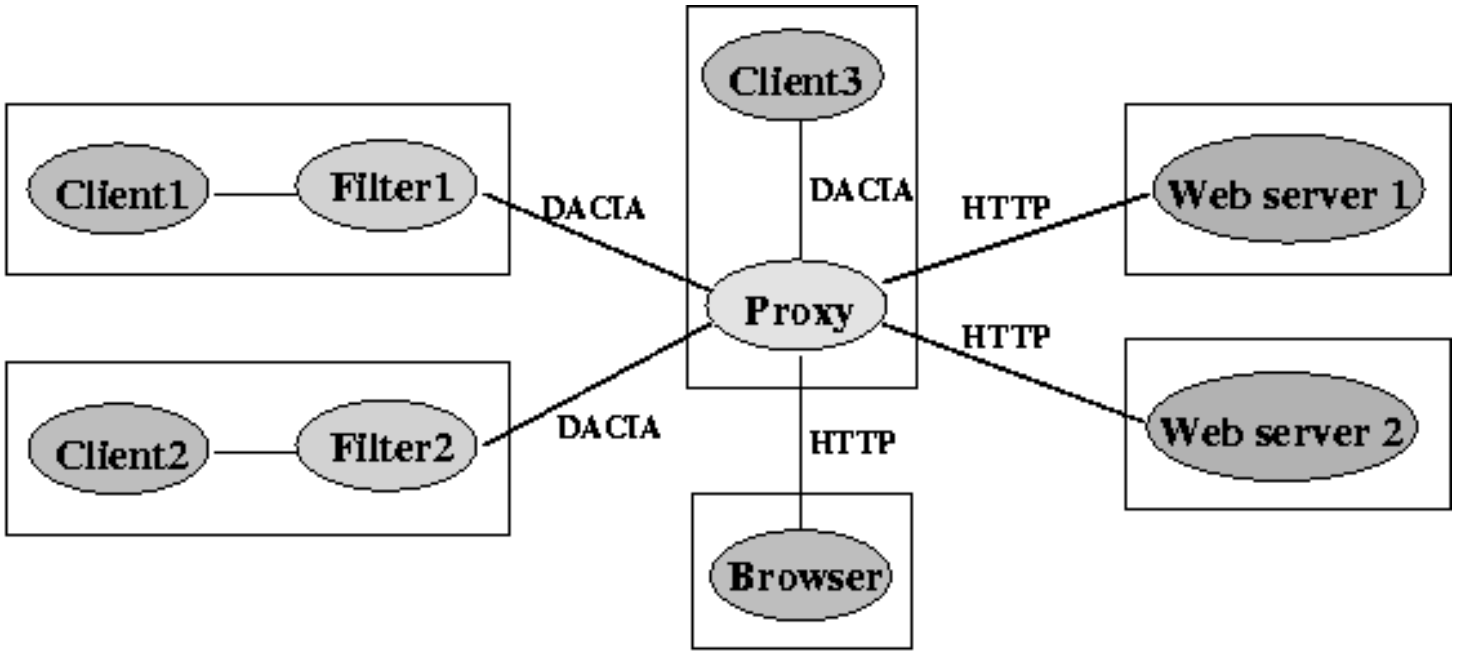


Whiteboard



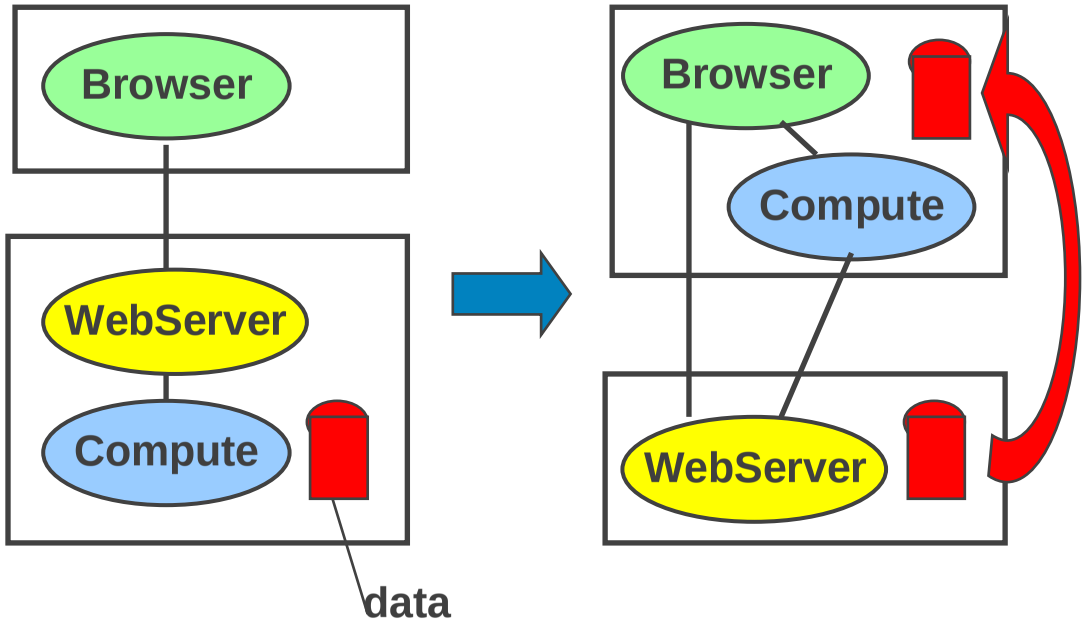
Web Proxy

- HTTP gateway
- Compose services: data filtering, client agents
- Separate user interface from active agent code
- Mobility – disconnected users



Adaptive Data Processing

- Move functionality between web servers and web clients (browsers)
- Reduce communication costs
- Local caching of data



Outline

- ✓Motivation and Design Goals
- ✓Related Work
- ✓System Architecture
- ✓Component Mobility
- ✓Dynamic Application Reconfiguration
- ✓Building Adaptive Applications
- Summary and Future Work

Summary

- DACIA - a component-based framework for building adaptive distributed applications
- Dynamic reconfiguration
  - Improve performance
  - Customized and extensible configurations
  - Cost of application maintenance and upgrade
- Application and user mobility
- Persistent connectivity
- Low communication overhead

Current and Future Work

- Location service
- Policies and algorithms for dynamic reconfiguration
- Formal specification of components
- Multiple platform implementation
- Deployment and experimentation
- Security infrastructure

Security

- Host security
  - Malicious components
  - Programming errors
- Component security
  - Execution integrity
  - Data integrity
  - Secrecy

http://www.eecs.umich.edu/~radu/dacia

radu@eecs.umich.edu