Critical Infrastructure Security: The Emerging Smart Grid

Cpt S 580-04, Cpt S 483-02, EE 582-03, EE 483-01

Dr. Anurag K. Srivastava, Dr. Carl Hauser, Dr. Dave Bakken

Computation Lecture 5: Overview of Distributed Computing (2/2)
Today’s Content

1. Middleware (Slides & 1 page)
2. Cloud Computing (slides only)
3. Architectural Models (2.1 and 2.3), as much as time allows
Middleware Perspective

• “Middleware is like underwear: it is absolutely essential, but it should never be seen in public.” unknown witticist

• Background info (only first page required): http://www.eecs.wsu.edu/~bakken/middleware.pdf
Context: (Most) Technology Marches On

• Hardware technology’s progress phenomenal in last few decades
  ▪ Moore’s Law
  ▪ Metcalf’s Law
  ▪ Graphics processing power

• Software technology’s progress is much more spotty
  ▪ “Software crisis”
  ▪ Yet SW is a large and increasing part of complex apps/systems!

• Apps and systems are rapidly becoming (more) networked
  ▪ Oops, distributed software is much harder yet to get right…

• Middleware a promising technology for programmability of distributed systems
Why Middleware?

• Middleware == “A layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system”

• Middleware exists to help manage the complexity and heterogeneity inherent in distributed systems

• Middleware provides higher-level building blocks ("abstractions") for programmers than the OS provides
  ▪ Can make code much more portable
  ▪ Can make them much more productive
  ▪ Can make the resulting code have fewer errors
  ▪ Analogy — MW:sockets ≈ HOL:assembler

• Middleware sometimes is informally called “plumbing”
  ▪ Connects parts of a distributed application with “data pipes” and passes data between them
Middleware Benefit: Masking Heterogeneity

- Middleware’s programming building blocks mask heterogeneity
  - Makes programmer’s life much easier!!
- Kinds of heterogeneity masked by middleware (MW) frameworks
  - All MW masks heterogeneity in network technology
  - All MW masks heterogeneity in host CPU
  - Almost all MW masks heterogeneity in operating system (or family thereof)
    - Notable exception: Microsoft middleware (*de facto*; not *de jure* or *de fiat*)
  - Almost all MW masks heterogeneity in programming language
    - Noteable exception: Java RMI
  - Some MW masks heterogeneity in vendor implementations
    - Object Management Group (omg.org) best here: CORBA (object-oriented), DDS (publish-subscribe)
Middleware Benefit: Transparency

• Middleware can provide useful transparencies:
  ▪ Access Transparency
  ▪ Location transparency
  ▪ Concurrency transparency
  ▪ Replication transparency
  ▪ Failure transparency
  ▪ Mobility transparency

• Masking heterogeneity and providing transparency makes programming distributed systems much easier to do!
Middleware and Legacy Systems

• Legacy systems are a huge problem (and asset) in industry and military domains!

• Middleware often called a “glue” technology: integrated “legacy” components
  ▪ Much distributed programming involves integrating components, not building them from scratch!

• Middleware’s abstractions are general enough to allow legacy systems to be “wrapped”
  ▪ Distributed objects are best here because more general
  ▪ End result: a very high-level “lowest common denominator” of interoperability
Middleware Wrapping Legacy Servers

Client VM
- CP/M & Visual Basic
- MacOS & Java
- MVS & Jovial
- Win8 & C#
- ...

Server VM
- IBM System 360 & PL/I
- DEC Ultrix & FORTRAN
- Apollo Aegis & COBOL
- Linux & C++
- Solaris & Ada

Middleware Distributed Object Wrapping Layer

Operation(args)
return value/exception
Middleware vs. Sockets

• Middleware is much easier to program!

• Example interface from CORBA (OMG) IDL:

```plaintext
module HelloApp {
    interface Hello {
        bool MyFunction(in float a, in string b,
        in int c, in string d, in float e,
        out double ret);
    }
};
```
Middleware vs. Sockets(2)

- Calling that interface in C++ with CORBA

```
boolean success =
    helloImpl.MyFunction(3.3, "hello", 2345, "bakken!", 67.34, doubleBox);
```

float a; char b[5]; int c; char d[7]; float e;
double rval; int success

// Ignore read errors. Hardcode field size, assume
// all systems are same CPU arch. and bit size
read(socket, &a, sizeof(float));
read(socket, b, sizeof(char) * 5);
read(socket, &c, sizeof(int));
read(socket, d, sizeof(char)*7);
read(socket, &e, sizeof(float));

// … continued on next slide …
Middleware vs. Sockets (4)

// ... continued from previous slide ...

... calculating return values etc goes here ...

// send back return value
write(socket, &rval, sizeof(double));

// cant tell if it actually was received, or if socket is broken
write(socket, &success, sizeof(int));

// again, no error checking
Middleware vs. Sockets (5)

• This socket code ignored all of the following:
  • Errors with the socket
  • Differences in CPU architecture (endianness)
  • Differences in representation of data types between languages
  • I/O errors
  • Type checking of data variables

• All of the above (and much more) are handled by middleware

• Middleware’s programming building blocks (abstractions) mask heterogeneity
  • Makes programmer’s life much easier!!
Multi-Layered Middleware
One Middleware Layering Taxonomy

**Applications**
- Domain-Specific Services
  - Services and APIs tailored to (and reusable only within) certain domains (health care, telecommunications, etc)
  - Examples: CORBA Domain Interfaces, Boeing Bold Stroke architecture

**Domain-Specific Services**
- Common MW Services
  - Adds high-level, domain-independent reusable services for events, fault tolerance, security,
  - Examples: CORBA Services, Eternal, QuO

**Common MW Services**
- Distribution MW
  - Provides rich distributed object model that supports much heterogeneity and transparency
  - Examples: CORBA, .NET, Java RMI

**Distribution MW**
- Infrastructure MW
  - Encapsulates core OS Comm. and concurrency services (sometimes enhances them too)
  - Examples: JVM (and other VMs), ACE, group comm.

(Figure courtesy of D. Schmidt)
Today’s Content

1. Middleware (Slides & 1 page)

2. Cloud Computing (slides only)

3. Architectural Models (2.1 and 2.3), as much as time allows
Sources of Info

1. cloudcomputing.ieee.org. Includes some training classes.


• Wikipedia.org articles on “Cloud computing” and “MapReduce”
Sources of Info (2)


Over-Hyped, Under-Defined Terms

- Cloud
- Service-Oriented Architecture (SOA)
- .... many more on the early side of the Hype Cycle!
- A gem from the past, when object-oriented programming was starting to get popular:
  “I have a cat named Trash. In the current political climate, it would seem that if I were trying to sell him (at least to a Computer Scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented.”

Prof. Roger King, U. Colorado at Boulder, 1989
Cloud Computing

• The “next new thing”
  ▪ Big data centers (probably hosted by power industry vendors or NERC or DHS/DoE, not Amazon or Google)
  ▪ These permit “consolidation”
    – 10x or better reductions in cost of operation
    – Far better equipment utilization and management
    – New styles of elastic computing, potential to compute directly on massive data collections
    – Adds up to a new way of computing that forces us to undertake new kinds of thinking
  ▪ But deliberately designed to trade off consistency for highest possible scalability
Utility Drivers for Cloud Computing

• Understaffed on the IT front
• Opportunity for much broader sharing of operational and planning data
• Opportunity for very advanced data analytics
Clouds From a Hardware Point of View

• Illusion of infinite computing resources available on demand
  ▪ Can start small with no need to provision far ahead

• Elimination of up-front commitment by Cloud users
  ▪ Saves precious capital
  ▪ Can start small: what I call “just-in-time expansion”

• Ability to pay for use of computing resources on a short-term basis as needed
NIST’s Cloud Essential Characteristics

• On-demand self-service
• Broad network access
• Resource pooling
• Rapid elasticity
  ▪ This is key .... We’ll talk about cloud killer apps for power grids shortly
• Measured service
IBM’s and a Book’s Cloud Definition

• “A cloud is a pool of virtualized computer resources. A cloud can host a variety of different workloads, including batch-style backend jobs and interactive and user-facing applications.”

• “Cloud computing applies a virtualized platform with **elastic resources on demand** by provisioning hardware, software, and data sets dynamically.” [emphasis mine]
  ▪ **Placement problem** is huge here (DS II from [CDKB5] chap. 2)
  ▪ Computer scientists would love to pass in info to influence placement and be told what mapping decisions were made
MapReduce

- Programming model built for processing and generating large sets of data (e.g., Apache Hadoop)
- Parallel & distributed algorithm on a cluster
- Map() performs filtering and sorting
  - Divides problems into smaller sub-problems based on key & distributed them to worker nodes
- Reduce(): Collects answers to sub-problems and combines in a given way for output
- Much more scalable than sequential algorithms
Deployment Models

- **Private Cloud**: for use by a single organization
  - RTE France has its own data center

- **Public Cloud**: open for use by public, many companies and users using simultaneously

- **Community Cloud**: operated for a specific community of consumers with shared concerns
  - E.g. ISO New England is starting one, will involve others
  - This is ultimately what grids need, IMO, managed right

- **Hybrid Cloud**: mix of more than one of the above 3.
  - Often private us used, but then public for overflow demand
Service Models (Bottom Up)

• Infrastructure as a Service (IaaS)
  ▪ Lets consumers provision processing, storage, network, data, Operating System, etc. Low-level.

• Platform as a Service (Paas)
  ▪ Provides much higher-level APIs: languages, libraries, middleware, etc. (c.f. “platform” from DS II section, from [CDKB5] chap. 2).
<table>
<thead>
<tr>
<th>Service and Deployment Models</th>
<th>Saas</th>
<th>Paas</th>
<th>IaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Cloud</td>
<td></td>
<td>Apprenda, Stackato</td>
<td>VMware, Hyper-V, OpenStack, CloudStack</td>
</tr>
<tr>
<td>Public Cloud</td>
<td>Salesforce.com, QuickBooks online, Office 365</td>
<td>Google AppEngine, Microsoft Azure, Vmware, CloudFoundry.com</td>
<td>Amazon EC2, Rackspace</td>
</tr>
<tr>
<td>Hybrid Cloud</td>
<td>(courtesy of Leverhawk.com)</td>
<td>Custom CloudFoundry</td>
<td>Custom, Rackspace</td>
</tr>
</tbody>
</table>
Risks of Cloud Computing

• Security & Privacy

• Vendor Lock-In
  ▪ Platform Lock-in: APIs not portable
  ▪ Data Lock-in: your data is stuck with a vendor who may legally own it
  ▪ Tools Lock-In: Tools to manage and develop cloud apps may not work across multiple vendors

• Isolation Failure

• Management Interface Compromise

• Insecure or Incomplete Data Deletion

• …. Probably a lot of grid-specific regulatory limitations…
GridCloud

- Combining GridStat/WSU plus Cornell (Birman et al) cloud computing technology
  - See slides from NASPI meeting February 2012
- Challenging questions with *highly elastic* apps
  - Rapid elasticity at scale
  - Predictability of such elasticity
  - Data consistency with such elasticity
- Architecture on next slide; running with EC2 for 2 years now
- Then outlining 8 killer apps that GridCloud will enable
GridCloud Architecture

- **State Est.**: Anjan Bose et al. Linear Hierarchical State Estimation
- Substation SE instances a la GridSim; see Power & Energy Mag. Feb'12
#1: Mitigation Control

- Rare combination of events do happen
  - Have lead to many blackouts when not mitigated!

- E.g., *N-3 contingency* (3 failures) never planned for
  - Infrequent but hugely expensive to analyze
  - GridCloud commissions thousands of nodes analyzing candidate mitigation steps in parallel
    - Once a contingency happens, you only have to solve N-2!
  - Best approach (actionable steps) is given to operators

- Acknowledgements: Prof. Mani Venkatasubramanian (WSU)
#2: Oscillation Alarm Processing

- Grids oscillate between regions
  - Negatively damping can lead to blackout
  - E.g., Oregon/California in July 1996: 0.3 Hz (!!)

- GridCloud commissions massive parallel computations exploring huge permutation space
  - Looking for trends and correlations of alarm data
  - Also huge number of model-based simulations too
  - Finds root cause much faster than possible today in much broader set of conditions

- Acknowledgements: Prof. Mani Venkatasubramanian (WSU)
Post-Tripping Fault Diagnosis

- Protection scheme trips a relay, but why?
  - Underlying cause must be ascertained *post facto*

- GridCloud commissions massive computations to identify the fault(s) that provoked the trip(s)
  - Many different kinds of fault diagnosis algorithms, all could be run in parallel
  - Possible integration candidate: openFLE (fault location engine) from Grid Protection Alliance

- Acknowledgements: Prof. Anuraug Srivastava (WSU)
#4: Multi-Resolution Frequency Disturbance Visualization

- Grid operates in very narrow range unless stressed
  - Frequency excursions outside this give clues to problems

- Frequency disturbance recorder (FDR): new device recording frequency disturbances at high rates
  - E.g., internal sampling of FNET device (in our lab): 1440 Hz

- GridCloud commissions thousands of parallel frequency rendering computations
  - Provide operators a rich suite of visualizations with which to better understand nature and cause of present excursion

- Acknowledgements: Prof. Yilu Liu (University of Tennessee, Knoxville)
#5: Multi-Dimensional Computations over Both Space and Time

- Two existing GridSim apps can be combined in rich ways possible only with cloud computing
- Hierarchical linear state estimation: rich coverage of (geographical) space
  - At one snapshot in time
  - Obvious extensions over more space with more PMUs
- Oscillation monitoring
  - Uses moving window of time (a few seconds typically)
  - Over streaming data
  - Produces a single number: damping factor
  - Obvious parallel computations over different sets of data with different time windows and algorithms
• Combination: provide rich set of two-dimensional (space, time) data to any desired location
  ▪ Enables extremely powerful new families of applications operating coherently over both space and time
  ▪ At each location: different time windows, different algorithms, different sets of data
  ▪ If available, people would inevitably think of many uses for this data

• Acknowledgements: Prof. Anjan Bose (WSU)
Balancing authorities (144 in North America) must have remote backup control centers
  - Hot backups with same data and apps

TVA found great value in having a tertiary control center
  - Limited to monitoring: control outputs computed but not used
  - Obvious candidates for the cloud
  - But this is barely scratching the surface here…
Major problem today: balancing authorities have almost no visibility anywhere in grid except for a few places in a few neighbors
- “Flying blind”, The Economist, 2004

Why not just share more?
- Data stored at another utility is problematic for owner

Storing in cloud could alleviate this
- Only access a subset of data and/or derived info
- Access opened up when grid sufficiently stressed
• Above is static with default steady state

• Could also drill down on demand with elastic computations
  ▪ Using higher-fidelity algorithms
  ▪ Using higher-resolution data

• FYI: ISO New England is starting a cloud pilot project (with GridCloud), and RTE France already has its own datacenter…

• Acknowledgements: Russell Robertson (Grid Protection Alliance), for the TVA example (though not the cloud possibilities)
Today’s Content

1. Middleware (Slides & 1 page)
2. Cloud Computing (slides only)
3. Architectural Models (2.1 and 2.3), as much as time allows
Introduction [2.1]

• Real-world systems should (ideally) be designed to function in widest possible range of circumstances (incl. difficulties and threats)

• Chap2: how properties and design issues of DSs can be captured and analyzed with descriptive models
  - **Physical models**: HW composition of computers (and devices) and networks that interconnect them
  - **Architectural models**: describe w.r.t. computational tasks done by computational elements (single or aggregate) connected by networks
  - **Fundamental models**: abstract perspective examining an individual aspect of a distributed system
    - **Interaction models** (struct+seq of elements’ comms), **failure models**, **security models**
Difficulties and Threats for DSs

• Many problems face designers of DSs!
• Widely varying modes of use
  ▪ Workload
  ▪ Some parts disconnected or with flaky connectivity
  ▪ Some need high bandwidth and/or low latency
• Wide range of system environments
  ▪ Heterogenieties discussed earlier
  ▪ Networks vary widely in performance (statically and dynamically)
  ▪ Scale from tens to millions of computers
  ▪ Geographic scale for DSs range from
    – tightly controllable LAN
    – somewhat controllable data center
    – normally uncontrollable (and unknowable) WAN.
Difficulties and Threats for DSs (cont.)

• Internal problems
  ▪ Non-synchronized clocks
  ▪ Conflicting data updates
  ▪ Many modes of HW+SW failure for individual components

• External threats: attacks on
  ▪ Confidentiality
  ▪ Integrity
  ▪ Availability (incl. DDoS attacks)
Architectural Models [2.3]

• Structure a system in terms of separately specified components and their relationships
• Goal: ensure structure meets present & (likely) future req.
• Concerns: reliability, manageability, adaptability, cost-effectiveness
• Three-phase buildup of concepts (*long* sub-chapter!)
  ▪ Core underlying architectural elements [2.3.1]
  ▪ Composite arch. patterns usable in isolation or combination [2.3.2]
    – Builds with elements in [2.3.1]
  ▪ Middleware platforms supporting programming styles emerging from [2.3.1] and [2.3.3]
Architectural Elements [2.3.1]

Need to consider 4 key questions:

1. What **entities** are communicating in the DS?

2. What **communication paradigm**/pattern do entities use?

3. What **roles and responsibilities** do entities have
   - May change!

4. How are entities mapped onto physical infrastructure (**placement**)
Communicating Entities

• System perspective: processes are communicating
  ▪ Simple environments (sensors): no processes, so entities ≡ nodes
  ▪ Most environments: threads, so technically the endpoints

• Programming perspective: more problem-oriented abstractions
  ▪ Objects: coherent packaging of code+data, multiple instances
    – Problem-oriented abstractions, units of decomposition
    – Access via interfaces (spec. in IDL)
    – Distributed objects more in Chap 5, 8
  ▪ Components
    – Similar to objects: code+data, interfaces
    – Also specify assumptions made (needed external components/interfaces) ... i.e., dependencies made explicit ... better “contract” for constructing systems
  ▪ Web services (access objects/components via WWW)
    – Rather ugly underlying technologies at times
Communication Paradigms

• 3 kinds:
  1. interprocess communication
  2. remote invocation
  3. indirect communication

• **Interprocess communication** (IPC)
  - Low-level support for communication
  - Usually socket API
Remote Invocation

• Most common (arguably), two-way exchange; buildup…

• Request-reply protocols (RRP) (application level 😞)
  ▪ Pattern imposed on underlying message passing to support client-server
  ▪ Client app code sends message with operation, parameters, bookkeeping in request message
  ▪ Server sends message with bookkeeping, parameters in reply message
  ▪ Low-level, typically simple embedded systems w/strong RT needs

• Remote procedure call (RPC)
  ▪ Make a remote call look (almost) like a local call
  ▪ Supports many transparencies and heterogeneities
  ▪ Directly supports client-server computing at higher level than RRP
Remote Invocation

• **Remote method invocation (RMI)**
  - Extends procedural RPC to object-oriented programming
  - Multiple object instances: can pass object refs/IDs as params
  - Tighter integration than RPC into the language
    - Benefits and drawbacks
    - E.g., Java language and Java RMI
Decoupled communication

• IPC, RRP, RPC, RMI all have explicit receivers/endpoints for each direction of comm
  ▪ Senders must know receivers IDs; receivers often know senders
  ▪ Sender and receiver must both exist at same time
  ▪ Can be less flexible than desirable for some apps

• **Space uncoupling**: senders do not need to know who sending to

• **Time uncoupling**: senders and receivers don’t have to have overlapping lifetimes (exist at same time)

• These uncouplings support **indirect communication** (Chap 6)
Overview of Indirect Communication Techniques

• **Group communication**
  - 1→many communications with group ID
  - Recipients join group, senders send to group
  - Groups often maintain membership, handle member failures
  - IP multicast trivial example, but many more fancier ones

• **Publish-subscribe**
  - Producers (publishers) send out info, publishers get it
  - Intermediate service is in between
  - Can subscribe based on data: topics
Overview of Indirect Communication Techniques (cont.)

• **Message queues**
  - Senders send to a specific queue, point-to-point
  - Consumers can get from queue (or be notified if new items)

• **Tuple spaces**
  - Structured data: (int, float, string, ...) with a given signature
  - Processes can read or remove tuples, can match values of some/all fields in tuple

• **Distributed shared memory (DSM)**
  - Abstraction of a shared address space or data structures therein
  - Lots of research in the late 80s and 90s, died out mostly
### Figure 2.2 Communicating entities and communication paradigms

<table>
<thead>
<tr>
<th>Communication entities (what is communicating)</th>
<th>Communication paradigms (how they communicate)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System-oriented entities</strong></td>
<td><strong>Interprocess communication</strong></td>
</tr>
<tr>
<td>Nodes</td>
<td>Message passing</td>
</tr>
<tr>
<td>Processes</td>
<td>Request-replica</td>
</tr>
<tr>
<td></td>
<td><strong>Remote invocation</strong></td>
</tr>
<tr>
<td></td>
<td>RPC</td>
</tr>
<tr>
<td></td>
<td>RMI</td>
</tr>
<tr>
<td><strong>Problem-oriented entities</strong></td>
<td><strong>Indirect communication</strong></td>
</tr>
<tr>
<td>Nodes</td>
<td>Group communication</td>
</tr>
<tr>
<td>Processes</td>
<td>Publish-subscribe</td>
</tr>
<tr>
<td></td>
<td>Message queues</td>
</tr>
<tr>
<td></td>
<td>Tuple spaces</td>
</tr>
<tr>
<td></td>
<td>DSM</td>
</tr>
</tbody>
</table>
Roles and Responsibilities

• Issue: what role does a given entity/component take?

• **Client-server**
  - Most widely studied and deployed
  - Client sends request to server, which replies
  - Can be either RPC or RMI
  - C/S w.r.t a given interaction: A→B→C means B server & client

• **Peer-to-peer (P2P):** can scale better, no centralized service
  - Observation: use not (just) centralized servers from a service, but end user can support that service (plenty of resources at edges!)
  - All entities are equals (and none/few “more equal than others”)
  - Entities run same program with same interfaces
  - Examples: BitTorrent, Skype, ..
Figure 2.3
Clients invoke individual servers

Key:
- Process: 
- Computer: 
Placement

• Issue: how to map entities (objects, services, ...) onto physical infrastructure?

• Must take into account many things:
  ▪ Patterns of communication
  ▪ Reliability and current load of given machines
  ▪ (Often) strong knowledge of application/service

• No optimal solutions, only strategies that help
  ▪ Mapping services onto multiple servers
  ▪ Caching
  ▪ Mobile code
  ▪ Mobile agents
• Mapping services to multiple servers (Fig 2.4)

• Caching
  - Cache: a store of recently used data objects closer or at a client
  - Examples?
  - Lotsa bookkeeping passed around to track updates/staleness/etc
  - If client requests stale object, it is fetched

• Mobile code (Fig 2.6)
  - Applets .... And client-side (edge) resources usually plentiful

• Mobile agents
  - Agent: a running program (code+data) that travels to carry out a task for some entity, and returns results
  - Difference from mobile code?
Figure 2.4 A service provided by multiple servers (servers are P2P)
Figure 2.6
Web applets

a) client request results in the downloading of applet code

b) client interacts with the applet
Architectural Patterns [2.3.2]

• Build on more primitive architectural elements in [2.3.1] and before

• “not themselves necessarily complete solutions but rather offer partial insights that, when combined with other patterns, lead the designer to a solution for a given problem domain”.
  - Extremely nice definition, lots of issues behind it!

• Patterns we cover
  - Layering
  - Tiered architectures
  - Thin clients
  - Other misc: proxy, brokerages, reflection
Layering

• Familiar from networking design
• In a DS, means a *vertical* organization of services into service layers

**Platform**: lowest-level HW and SW layers

**Middleware**: layer(s) of software above platform
  - masking heterogeneities
  - Providing higher-level programming abstraction
    - much closer to application’s items of domains than the platform
  - Supports different kinds of interactions: RCP, RMI, pub-sub, ...
Figure 2.7 Software and hardware service layers in distributed systems

- Applications, services
- Middleware
- Operating system
- Computer and network hardware

Platform
Tiered Architectures

- **Horizontal** organization of application/service functionality across different servers

- Typical **three-tiered architecture**:
  - **Presentation logic**: user interactions and visualization
  - **Application logic**: app-specific processing (AKA **business logic**)
  - **Data logic**: persistent storage of data (e.g., database)
  - Above on separate processes

- Two-tiered can split above functionality across client-server in different ways
Q: tiered architectures contradictory or complimentary to layering?
Thin Clients & Other Patterns

• General-purpose desktop computer can be a pain to manage

• **Thin client**: SW layer supporting a window-based UI accessing remote programs and servers (Fig 2.10)

• X-Windows early example

• Other architectural patterns
  - **Proxy**: intermediate in local address space (MW, web proxies)
  - **Brokerage**: service broker helps service requester find the right service provider (Fig 2.11)
  - **Reflection**: application/service utilizes knowledge of its internal structure; very very useful (Blair research)
    - **Introspection**: dynamic discovery of properties (read-only)
    - **Intercession**: dynamically modifying structure or behavior
Figure 2.10
Thin clients and compute servers

Network computer or PC

Thin Client

network

Application Process
Figure 2.11
The web service architectural pattern