LECTURE 3: CONCURRENT & DISTRIBUTED ARCHITECTURES

Contents

• Introduction
• Flynn’s Taxonomy
• MIMD: Tight & Loose Coupling
• Software Architectures for Distributed Systems:
  – Layered, Object-/Event-based, Shared Dataspace Architectures
• System Architectures
  – Centralized Architectures:
    • 2 & multi-tiered architectures
    • Fat & Thin Clients
  – Decentralized Architectures
    • Structured P2P Systems: Chord & Pastry Routing algorithms
    • Unstructured P2P Systems
    • Hybrid Systems
• Architectures V Middleware
Intro to Architectures in Concurrent & Dist’d Systems: S/w V System Architectures

- Organizing concurrent & distributed systems is mostly about the software components making up the system.
- These software architectures (aka Programming Models) dictate the organization & interaction of the various s/w components.
- The actual realization of a system requires instantiating and placing software components on real machines.
- There are many different choices that can be made in doing so.
- The final instantiation of a software architecture is referred to as a system architecture (aka Machine Model).

SECTION 3.1: CONCURRENT ARCHITECTURES & PROGRAMMING MODELS
Aside on Writing Concurrent Code

1. Identify concurrency in task
   – Do this on a piece of paper
2. Expose the concurrency when writing the task
   – Choose a programming model and language that allow you to express this concurrency
3. Exploit the concurrency
   – Carefully choose a language & hardware that facilitate advantage to be taken of the concurrency (often one ⇔ another)

• Value of a programming model is judged on
  – Generality: how well a range of different problems can be expressed for a variety of different architectures,
  – Performance: how efficiently compiled programs can execute on these architectures.

Parallel Programming Model

• Definition: Programming model comprises languages & libraries that create an abstract view of the machine.
  – Control
    • What orderings exist between operations?
    • How do different threads of control synchronize?
  – Data
    • What data is private vs. shared?
    • How is logically shared data accessed or communicated?
  – Synchronization
    • What operations can be used to coordinate parallelism?
    • What are the atomic (indivisible) operations?
Concurrent Architecture Taxonomies

- As seen above, Michael Flynn in 1966 classified machines into a taxonomy by the number of instruction and data streams.
- We examine these from the standpoint of concurrent architectures.

![Processor Organizations Diagram]

More on MIMD

- **MIMD**
  - General purpose processor
  - Each can process all instructions necessary.
  - Further classified by method of processor communication:
    - **Tight Coupling**
    - **Loose Coupling**
Concurrent Architectures

- **Machine Model #1: Shared Memory**
- Processors all connected to a large shared memory
  - Typically Symmetric Multiprocessors (SMPs e.g. IBM SMPs)
  - Multicore chips, except caches are often shared in multicores
  - But
    - Bus is a bottleneck (interconnect performance not scalable)
    - Also, shared memory can give issues with *race conditions*
    - Can be fixed by adding locks of some sort, at performance cost

---

Programming Models

- **Programming Model # 1: Shared Memory**
  Program is a collection of threads of control.
  - Each thread has set of private variables, e.g., local stack variables & set of shared variables, e.g., static variables
  - Implicit comms between threads writing/reading shared variables
  - Threads coordinate by synchronizing on shared variables
  - Here model used by threads calculating the sum \( S \) of an array
Concurrent Architectures (/2)

- **Machine Model #2: Distributed Memory**
  Processors have their own memory but typically fast interconnect
  - Each processor has its own memory and cache but cannot directly access another processor’s memory.
  - Each “node” has a Network Interface (NI) for all communication and synchronization.
  - Example: IBM SP2, Beowolf Cluster

![Diagram of distributed memory model](image)

Programming Models (/2)

- **Programming Model #2: Message Passing**
  Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space—NO shared data.
  - Logically shared data is partitioned over local processes.
  - Here, similar calculation as last time.

![Diagram of message passing model](image)
Concurrent Architectures (/3)

• **Machine Model #3: Clusters**
Used for computation-intensive purposes, (Vs for IO operations such as web service or DBs.)
• Emerged due to trends e.g. low-cost cores, high speed n/ws & s/w for HP distributed computing.
• Wide applicability from small biz clusters to fastest supercomputers
• Applications are nonetheless limited, since s/w must be purpose-built per task.

Programming Models (/3)

• **Programming Model # 3: Hybrids**
Need to run "same/similar computation" on many nodes very fast
  — Common model: Hybrid MPI + OpenMP
    • Each SMP node = 1 MPI process, w MPI comm on node interconnect
    • OpenMP inside of each SMP node
  — Maybe gives the highest performance?
    • Advantage: Could be good for heavyweight comms between nodes & lightweight threads within a node
    • Disadvantages:
      – Very difficult to start with OpenMP and modify for MPI
      – Very difficult to program, debug, modify and maintain
      – Generally, cannot do MPI calls within OpenMP parallel regions
      – Only people experienced in both should use this mixed prog model
Architectures for Distributed Systems

• Introduction
  - Examine traditional centralized distributed systems architectures where 1 server implements most s/w components (thus functionality)
  - Remote clients access the server using simple communication means.
  - Also consider decentralized architectures in which machines more or less play equal roles, as well as hybrid organizations.
  - From Lecture 1, one aim of distributed systems is separating applications from underlying platforms using a m/w layer.
  - Adopting such a layer is an important architectural decision: main purpose is distribution transparency.
  - However, trade-offs must be made to have transparency, leading to various techniques to make middleware adaptive.
Distributed Architectural Styles

• **#1 Layered Architectures**
  - Basic idea is simple: organize components in layers
  - Component at layer \( N \) can call those at underlying layer \( N - 1 \) (but not vice versa)
  - This so-called *application layering* is shown in the diagram
  - A key observation is that control generally flows from layer to layer
  - E.g. requests go down the hierarchy whereas the results flow upward.
  - This model has been widely adopted by the networking community

Distributed Architectural Styles (/2)

• **#2 Object-Based Architectures**
  - A far looser organization is followed in object-based architectures,
  - Each object corresponds to what we have defined as a *component*,
  - These components are connected through a *(remote) procedure call* mechanism.
  - This software architecture matches the client-server system architecture (described below).
  - Layered & object-based architectures still form the most important styles for large s/w systems
Distributed Architectural Styles (/3)

• #3 Event-Based Architectures
  – Processes communicate thro event propagation, optionally with data.
  – For DS, event propagation usually associated with so-called publish/subscribe.
  – Idea: processes publish events & m/w ensures only subscribed processes receive them.
  – The main advantage of such systems is loose coupling of processes
  – Needn’t refer to each other explicitly.
  – A.k.a. decoupled in space, or referentially decoupled.

Distributed Architectural Styles (/4)

• #4 Shared Data-Space Architectures
  – Event-based architectures can be combined w data-centered architectures
  – Gives what is also known as shared data spaces.
  – Essence: processes now also decoupled in time
  – Thus need not both be active when communication takes place.
  – Also, many shared data spaces use a SQL-like interface to shared repository.
  – Means data can be accessed using a description rather than an explicit ref, as per files.
SECTION 3.3: SYSTEM ARCHITECTURES: CENTRALIZED & DECENTRALIZED ARCHITECTURES

System Architectures: Centralized Architectures

*Basic Client–Server Model Characteristics*
- There are processes offering services (*servers*)
- There are processes that use services (*clients*)
- Clients and servers can be on different machines
- Sometimes Clients can be servers & vice versa
- Clients follow request/reply model with respect to using services
- Thinking in terms of Clients requesting Services from Servers aids in the understanding of Distributed Systems

[Diagram of Plain Ol' Client-Server]
System Architectures (/2):

- **Application Layering: Traditional three-layered view**
  1. **User-interface layer** contains units for an application’s user interface
  2. **Processing layer** contains the functions of an application, i.e. no specific data
  3. **Data layer** contains data client wants to process thro application components

    - Found in many distributed info systems, using traditional DB technology and accompanying applications.

Typical Web Browser Architecture

![Typical Web Browser Architecture](image)

Core Functionality

Transforming user
keywords into DB
queries & ranking
results on return

System Architectures:
Centralized Architectures (/3)

- **Multi-Tiered Architectures: Variations on traditional 3-layered view**
  1. **Single-tiered**: dumb terminal/mainframe configuration
  2. **Two-tiered**: client/single server configuration
  3. **Three-tiered**: each layer on separate machine (server may act as client)

![Alternative Client-Server Organizations](image)
**System Architectures: Decentralized Architectures**

- In multi-tiered architectures, the different tiers correspond directly to logical organization of applications – called **Vertical distribution**
- In **horizontal distribution** Client or Server may be split into logically equivalent parts each with own part of data set
- In the last couple of years there has been a tremendous growth in such **peer-to-peer (P2P)** systems:
  - **Structured P2P**: nodes are organized following a specific distributed data structure (usually a Distributed Hash Table)
  - **Unstructured P2P**: nodes have randomly selected neighbours. Each node has a list of neighbours which is constructed in a random way.
  - **Hybrid P2P**: some nodes are appointed special functions in a well-organized fashion

**Decentralized Architectures (/2): Structured P2P Systems**

- In virtually all cases, have **overlay networks**
  - This is n/w where nodes are processes & links are communication channels
  - Data is routed over connections setup between nodes.
- As processes can’t communicate directly with others, available communication channel must be used (a.k.a. **Application-level Multicasting**)
  - ALM is offered by middleware (in contrast to low-level TCP/IP Multicasting)
  - Basic idea is to organize nodes in a structured overlay n/w such as a logical ring.
  - Specific nodes are made responsible for services based only on their ID.
  - Random key is assigned to a data item from a large (eg 128 bit) identifier space
  - The system provides an operation **LOOKUP(key)** that will efficiently route the lookup request to the associated node.
  - When the key is returned, the network address of node responsible for the data (known as the **successor**) item stored is returned.
**Decentralized Architectures (/3): Structured P2P Systems: Chord Case Study**

- **Details of Chord Algorithm**
  1. Assign random key \((m\text{-bit identifier})\) to data item & random number \((m\text{-bit identifier})\) to node in system,
  2. Implement an efficient & deterministic system to map a data item to a node based on some distance metric,
  3. This means that data item should physically be as close to node as possible
  4. \(\text{LOOKUP(key)} \equiv \text{returning network address of node responsible for that data item,}\)
  5. Do this by routing a request for the data item to responsible node (successor).
  6. Node with key \(k\) falls under the jurisdiction of node with smallest \(\text{id} \geq k\)
  7. This process of looking up node’s name (& any info stored there) called \text{name resolution}\n
**Decentralized Architectures (/3): Structured P2P Systems: Chord Case Study**

- **Principle of Operation of Chord**
  - Membership management in Chord doesn’t follow a logical organization of nodes in a ring as shown in diagram (previous).
  - Lookups on keys can be done in \(O(\log_2 N)\) steps.
  - Each node \(p\) maintains a finger table \(FT_p[i]\) with at most \(m\) entries:
    \[ FT_p[i] = \text{succ}(p + 2^{i-1}) \]
  - Note: \(FT_p[i]\) points to the first node succeeding \(p\) by at least \(2^{i-1}\)
  - This is because Chord is an algorithm based on binary (will look at higher order algorithms later)
  - To look up a key \(k\), node \(p\) forwards the request to node with index \(j\) satisfying
    \[ q = FT_p[j] \leq k < FT_p[j + 1] \]
  - If \(p < k < FT_p[1]\) the request is also forwarded to \(FT_p[1]\)
Decentralized Architectures (/4): Structured P2P Systems: Chord Case Study

• Building Finger Tables in Chord

Some calculations for Finger tables in the diagram:

\[ FT_1[1] = \text{succ}(1 + 2^0) = \text{succ}(2) = 4 \]
\[ FT_2[2] = \text{succ}(1 + 2^1) = \text{succ}(3) = 4 \]
\[ FT_3[3] = \text{succ}(1 + 2^2) = \text{succ}(5) = 7 \]
\[ FT_4[4] = \text{succ}(1 + 2^3) = \text{succ}(9) = 12 \]
\[ FT_{15}[15] = \text{succ}(15 + 2^0) = \text{succ}(16) = \text{succ}(0) = 1 \]
\[ FT_{16}[16] = \text{succ}(15 + 2^1) = \text{succ}(17) = \text{succ}(1) = 1 \]
\[ FT_{17}[17] = \text{succ}(15 + 2^2) = \text{succ}(19) = \text{succ}(3) = 4 \]
\[ FT_{18}[18] = \text{succ}(15 + 2^3) = \text{succ}(23) = \text{succ}(7) = 7 \]

Decentralized Architectures (/5): Structured P2P Systems: Chord Case Study

• Principle of Joining a System in Chord

- Node wanting to join system starts by generating random identifier \( id = 26 \).
  - Then node simply contacts an arbitrary node & does a lookup on \( id \),
  - Returns address of \( \text{succ}(id) = 32 \), node responsible for looking after \( id \)
  - Next, node simply contacts \( \text{succ}(id) \) & it's predecessor & inserts self in ring
  - This consists of updating the finger tables.
  - Insertion also yields that each data item whose key is now associated with node \( id \), is transferred from \( \text{succ}(id) \).
- Chord scheme requires that each node also stores info on its predecessor.
Decentralized Architectures (/6):
Structured P2P Systems: Chord Case Study

• Problems in Chord

• Logical organization of overlay nodes may lead to erratic msg transfers in underlying Internet: node $k$, node $\text{succ}(k)$ may be far apart.

  — Topology-aware node assignment:
    • When assigning an ID to a node, make sure that nodes close in the ID space are also close in the network.
    • Can be very difficult.

  — Proximity routing:
    • Maintain more than one possible successor, and forward to the closest.
    • Example: in Chord $FT_2[1]$ points to first node in $[p + 2^{i-1}, p + 2^i - 1]$.
    • Node $p$ can also store pointers to other nodes in the interval.

  — Proximity neighbour selection:
    • When there is a choice of selecting who your neighbour will be (not in Chord), pick the closest one.

Decentralized Architectures (/7):
Structured P2P Systems: Pastry Case Study

• Properties of Pastry:

  • $\text{PASTRY}$ is an implementation of a Distributed Hash Table (DHT) algorithm for P2P routing overlay

  • Salient features:
    • Fully decentralized
    • Scalable
    • High fault tolerance

  • Each node is identified by a unique 128 bit node id ($\text{NodeId}$) generated randomly so each has same probability of being chosen

  • Node with similar $\text{NodeId}$ may be geographically far apart

  • Given a key, $\text{PASTRY}$ can deliver a message to node with closest $\text{NodeId}$ to key within $\log_2 N$ steps,
    where $b$ is a configuration parameter (usually $b = 4$) and $N$ is the number of nodes
Decentralized Architectures (/8):
Structured P2P Systems: Pastry Case Study (/2)

- **Pastry Routing Algorithm:**
  - Given want to find PASTRY n/w node with *NodeId* closest to given *key*
    - Note that *NodeId* & *key* are both 128 bit sequences
    - Both *NodeId* & *key* can be thought as sequence of digits with base $2^b$
  
- **Routing idea:**
  1. Each routing step, node normally forwards message to a node whose *NodeId*
     shares with *key* a prefix min. 1 digit longer than *key* shares with present node.
  2. If such a node unknown, message is forwarded to a node that shares same
     prefix of actual node but its *NodeId* is numerically closer to *key*

Decentralized Architectures (/9):
Structured P2P Systems: Pastry Case Study (/3)

- **State of a Node in Pastry:**
  - Each PASTRY node has a *state* consisting of:
    - A *routing table* $R$
      - used in the first phase of the routing (*long distances*)
    - A *neighbourhood set* $M$
      - contains *NodeId* & IP addresses of $|M|$ nodes which are *closest*
        (according to a *metric*, e.g. geog. or ping distance) to considered node
    - A *leaf set* $L$
      - contains *NodeId* & IP addresses of $|L|/2$ nodes with *NodeId* numerically *
        closest on smaller* side of present *NodeId*,
      - and $|L|/2$ nodes with *NodeId* numerically *closest on the larger* side of
        present *NodeId*.
      - $L$ usually taken to be 16
Decentralized Architectures (/10):
Structured P2P Systems: Pastry Case Study (/4)

- **Routing table in Pastry:**
  - This is a \( \lceil \log_2 N \rceil \times (2^b - 1) \) columns table
  
  \[ \text{where } \lceil \log_2 N \rceil \text{ is the max number of hops between any pair of nodes} \]
  
  \[ b \text{ is the configuration parameter (usually 4) and} \]
  
  \[ N \text{ is the number of PASTRY nodes in the network} \]
  
  - The \( 2^b - 1 \) entries at row \( n \) each refer to a node whose NodeId shares the present node NodeId in the first \( n \) digits
  
  - However the \( (n + 1) \)th digit has one of the \( 2^b - 1 \) possible values other than \( (n + 1) \)th digit digit in the present node id.
  
  - The choice of \( b \) is a choice between the size of the populated part of the Routing table (\( \lceil \log_2 N \rceil \times (2^b - 1) \) entries) & max number of hops.
    
    - e.g. a value of \( b = 4 \) and \( N = 10^6 \) nodes gives \( \sim 75 \) entries and \( \sim 5 \) hops
    
    - while \( b = 4 \) and \( N = 10^9 \) Nodes gives \( \sim 105 \) entries and \( \sim 7 \) hops

---

Decentralized Architectures (/11):
Structured P2P Systems: Pastry Case Study (/5)

- **Example Routing Table \( R \) in Pastry:**
  
  \[ N = 1024 \] Nodes, \( b = 2 \) so \( \lceil \log_2 N \rceil = 5 \) rows, \( 2^b - 1 = 3 \) columns
  
  - **Row \( i \):** Holds ids of Nodes whose IDs share an \( i \) digit prefix with Node
  
  - **Column \( j \):** digit \( (i + 1) = j \)
  
  - Contains topologically closest node that meets these criteria
Decentralized Architectures (/12):
Structured P2P Systems: Pastry Case Study (/6)

• **Example Routing in Pastry:** $N = 1024$ Nodes, $b = 2, L = 8$

• **Leaf Table $L$ for NodeID 32101**
  - $L/2$ smaller, $L/2$ larger
  - Fixed maximum size
  - Similar to Chord’s finger table
  - Used for routing and recovery from departures of nodes

<table>
<thead>
<tr>
<th>Smaller NodeID's</th>
<th>Larger NodeID's</th>
</tr>
</thead>
<tbody>
<tr>
<td>32100 32023</td>
<td>32110 32121</td>
</tr>
<tr>
<td>32012 32022 32123</td>
<td>32120</td>
</tr>
</tbody>
</table>

• **Neighbour Set $M$**
  - Contains nearby nodes (based on some scalar proximity metric e.g. geography, latency, IP hops etc)
  - Fixed maximum size
  - Irrelevant for routing

---

Decentralized Architectures (/13):
Structured P2P Systems: Pastry Case Study (/7)

• **Routing Algorithm of Packet with NodeID $A$, key $D$ (both 128 bit):**

  1. if $(L_{-1} \leq D \leq L_{1})$ then
  2. // $D$ is in the Leaf Node Set
  3. forward to $L_i$, such that $|D - L_i|$ is minimal, i.e. closest NodeID in $L$
  4. else
  5. // search for a node with longer shared prefix in the routing table
  6. Let $l = shl(D, A)$
  7. if ($R^D_l \neq null$) then
  8. forward to $R^D_l$ // entry in routing table row $l$, column $D_i$
  9. $D_i$ is the value of the $l$’s digit in the key $D$
  10. else
  11. // rarely
  12. forward to $T \in L \cup R \cup M$ such that
  13. $shl(T, D) \geq l, |T - D| < |A - D|$  
  14. search for node $T$ with longest prefix out of merged set

Lecture 3: Concurrent & Distributed Architectures  CA4006 Lecture Notes (Martin Crane 2019) 37

Lecture 3: Concurrent & Distributed Architectures  CA4006 Lecture Notes (Martin Crane 2019) 38
Decentralized Architectures (/14): Unstructured P2P Systems

- Many unstructured P2P systems try to maintain a random graph.
- Basic principle is for each node is required to contact a randomly selected other node:
  - Let each peer maintain a partial view of the network, consisting of $c$ other nodes.
  - Each node $P$ periodically selects a node $Q$ from its partial view.
  - $P$ and $Q$ exchange information and exchange members from their respective partial views.
- It turns out that, depending on the exchange, randomness, but also robustness of the network can be maintained.

Decentralized Architectures (/15): Unstructured P2P Systems (/2)

- Topology Management of Overlay Networks
- Basic idea is to distinguish two layers:
  1. maintain random partial views in lowest layer;
  2. be selective on who you keep in higher-layer partial view.
- Lower layer feeds upper layer with random nodes; upper layer is selective when keeping references (e.g. based on distance).
Decentralized Architectures (/16):
Unstructured P2P Systems (/3)

• **Topology Management of Overlay Networks (cont’d)**

• To construct a torus, Consider a $N \times N$ grid.

  Keep only refs to nearest neighbours:

  $$
  \| (a_1, a_2) - (b_1, b_2) \| = d_1 + d_2
  $$

  $$
  d_i = \min\{N - |a_i - b_i|, |a_i - b_i|\}
  $$

Here, there are two points: $(a_1, a_2) = (1,3)$ and $(b_1, b_2) = (3,0)$

hence

$$
  d_1 = \min(4 - 2, 2) = 2 \text{ (both paths same length)}
$$

and

$$
  d_2 = \min(4 - 3, 3) = 1 \text{ (green path is shorter)}
$$
Decentralized Architectures (/18):
Unstructured P2P Systems (/5)

- Topology Management of Overlay Networks (cont’d)
- Explanation
  - With minimum distance condition, a toroidal shape emerges.

Decentralized Architectures (/19):
Hybrid Architectures: C-S combined with P2P

- Example: Edge-server architectures, which are often used for Content Delivery Networks

Viewing the Internet as consisting of a collection of edge servers.
Decentralized Architectures (/20):
Hybrid Architectures: C-S with P2P (/2)

- **Internet as consisting of a collection of edge servers**
- An important class of distributed systems that is organized according to a hybrid architecture is formed by *edge-server systems*.
- Such systems are deployed on the Internet where servers are placed "at the edge" of the network.
  - Edge is formed by boundary between enterprise n/w and actual Internet, (for example, as provided by an ISP).
  - Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at edge of Internet.
  - Edge-Server thus serves content and optimises delivery
- **Content Delivery Networks** offers storage of copies of webpages for rapid reaccessing.

Decentralized Architectures (/21):
Hybrid Architectures: C-S combined with P2P

- **Example: Hybrid Architectures: C/S with P2P – BitTorrent**
- **Basic Idea:** Tracker (server with list of active nodes to download chunks of file) gives single copy (*seed*) of file (F), *swarm* is all nodes with some/all of F
- **Steps:**
  1. Client Node does a Lookup on F,
  2. BT webpage gives ref to file server with *.torrent* file for F (with Tracker).
  3. BT Client s/w talks to tracker to find other BT Nodes with whole/part of F.
  4. Tracker identifies swarm (i.e. connected peers sending/receiving) F.
  5. Tracker helps client trade pieces of F needed with others in swarm.

![Diagram of BitTorrent process](image)
Architecture V Middleware

- **Architecture and Middleware**
  - Considering the architectural issues above, a question that comes to mind is where middleware fits in.
  - Important aim is to give a degree of distribution transparency, i.e. try to hide data distribution, processing, and control from applications.
  - What is commonly seen in practice is that middleware systems actually follow a specific architectural style.
  - The chosen style may not be optimal in all cases.
  - So may need to (dynamically) adapt behaviour of the middleware.

**Interceptors**

- These intercept usual flow of control when invoking a remote object.
- Thus they allow other (application specific) code to be executed.
- This is demonstrated in the diagram (over)

---

Architecture V Middleware : Interceptors (2)

- **Remote Object Invocation**
  - Basic idea:
    Object A can call a method belonging to object B, living on a different machine to A.
  - Steps:
    1. A offered a local interface (same as B’s).
    2. A calls method available in that interface.
    3. A’s call transformed into a generic object invocation, enabled thro a general object-invocation interface offered by m/w at A’s machine.
    4. Finally, GOI is transformed into a message sent thro the transport-level network interface offered by A’s local operating system.
Summary

• Flynn’s Taxonomy is a classic but still useful way to classify architectures:
  – SISD, SIMD, MIMD can still be identified in supercomputers today
  – MIMD can be split into Tight & Loose Coupling

• Software Architectures for Distributed Systems divide into:
  – Layered, Object-/Event-based, Shared Dataspace Architectures

• System Architectures
  – Centralized Architectures:
    • 2 & multi-tiered architectures
    • Fat & Thin Clients
  – Decentralized Architectures can be divided into
    • Structured P2P Systems (e.g. Chord & Pastry Routing algorithms)
    • Unstructured P2P Systems
    • Hybrid Systems (e.g. BitTorrent)

• Middleware can sometimes be used to fill in for architecture