

# Chapter 12: Distributed Shared Memory

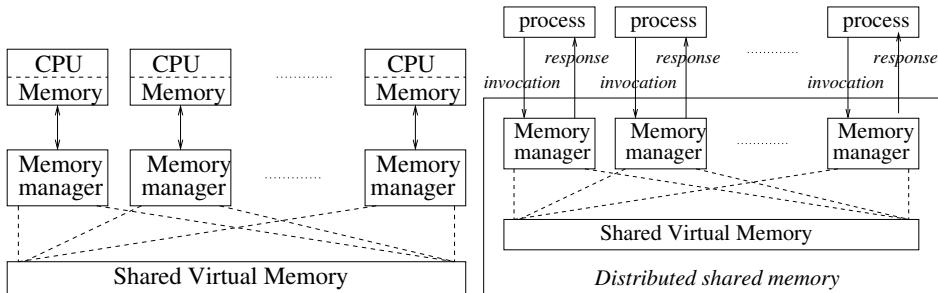
Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

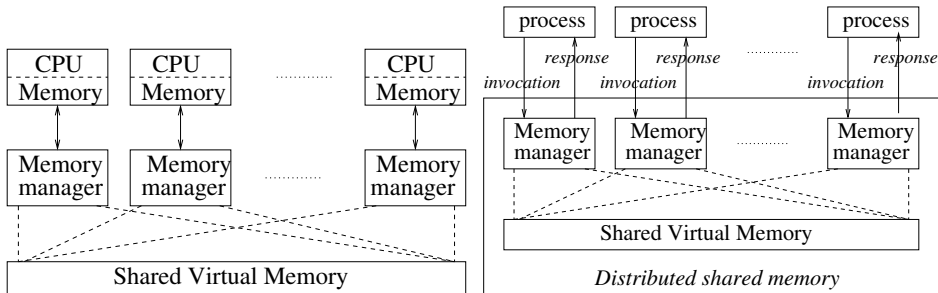
# Distributed Shared Memory Abstractions

- communicate with Read/Write ops in shared virtual space
- No Send and Receive primitives to be used by application
  - ▶ Under covers, Send and Receive used by DSM manager
- Locking is too restrictive; need concurrent access
- With replica management, problem of consistency arises!
- $\implies$  weaker consistency models (weaker than von Neumann) reqd



# Distributed Shared Memory Abstractions

- communicate with Read/Write ops in shared virtual space
- No Send and Receive primitives to be used by application
  - ▶ Under covers, Send and Receive used by DSM manager
- Locking is too restrictive; need concurrent access
- With replica management, problem of consistency arises!
- $\implies$  weaker consistency models (weaker than von Neumann) reqd



# Advantages/Disadvantages of DSM

## Advantages:

- Shields programmer from Send/Receive primitives
- Single address space; simplifies passing-by-reference and passing complex data structures
- Exploit locality-of-reference when a block is moved
- DSM uses simpler software interfaces, and cheaper off-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
- No memory access bottleneck, as no single bus
- Large virtual memory space
- DSM programs portable as they use common DSM programming interface

## Disadvantages:

- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than msg-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

# Advantages/Disadvantages of DSM

## Advantages:

- Shields programmer from Send/Receive primitives
- Single address space; simplifies passing-by-reference and passing complex data structures
- Exploit locality-of-reference when a block is moved
- DSM uses simpler software interfaces, and cheaper off-the-shelf hardware. Hence cheaper than dedicated multiprocessor systems
- No memory access bottleneck, as no single bus
- Large virtual memory space
- DSM programs portable as they use common DSM programming interface

## Disadvantages:

- Programmers need to understand consistency models, to write correct programs
- DSM implementations use async message-passing, and hence cannot be more efficient than msg-passing implementations
- By yielding control to DSM manager software, programmers cannot use their own msg-passing solutions.

# Issues in Implementing DSM Software

- Semantics for concurrent access must be clearly specified
- Semantics – replication? partial? full? read-only? write-only?
- Locations for replication (for optimization)
- If not full replication, determine location of nearest data for access
- Reduce delays, # msgs to implement the semantics of concurrent access
- Data is replicated or cached
- Remote access by HW or SW
- Caching/replication controlled by HW or SW
- DSM controlled by memory management SW, OS, language run-time system

# Issues in Implementing DSM Software

- Semantics for concurrent access must be clearly specified
- Semantics – replication? partial? full? read-only? write-only?
- Locations for replication (for optimization)
- If not full replication, determine location of nearest data for access
- Reduce delays, # msgs to implement the semantics of concurrent access
- Data is replicated or cached
- Remote access by HW or SW
- Caching/replication controlled by HW or SW
- DSM controlled by memory management SW, OS, language run-time system

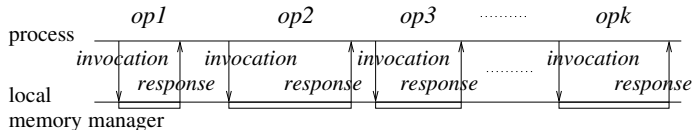
# Comparison of Early DSM Systems

Type of DSM	Examples	Management	Caching	Remote access
single-bus multiprocessor	Firefly, Sequent	by MMU	hardware control	by hardware
switched multiprocessor	Alewife, Dash	by MMU	hardware control	by hardware
NUMA system	Butterfly, CM*	by OS	software control	by hardware
Page-based DSM	Ivy, Mirage	by OS	software control	by software
Shared variable DSM	Midway, Munin	by language runtime system	software control	by software
Shared object DSM	Linda, Orca	by language runtime system	software control	by software



# Memory Coherence

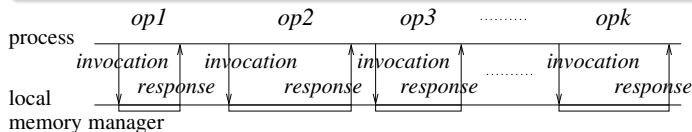
- $s_i$  memory operations by  $P_i$
- $(s_1 + s_2 + \dots s_n)! / (s_1! s_2! \dots s_n!)$  possible interleavings
- Memory coherence model defines which interleavings are permitted
- Traditionally, Read returns the value written by the most recent Write
- "Most recent" Write is ambiguous with replicas and concurrent accesses
- DSM consistency model is a *contract* between DSM system and application programmer



# Strict Consistency/Linearizability/Atomic Consistency

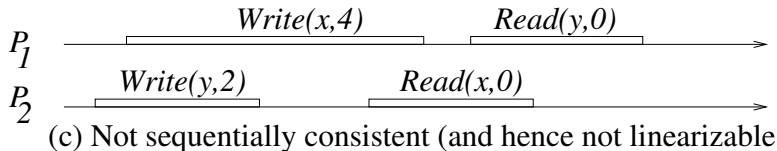
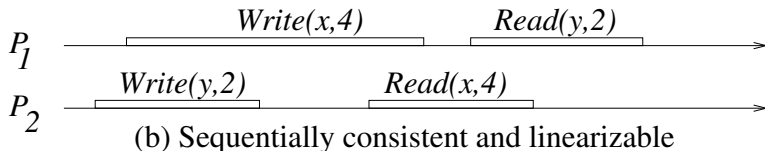
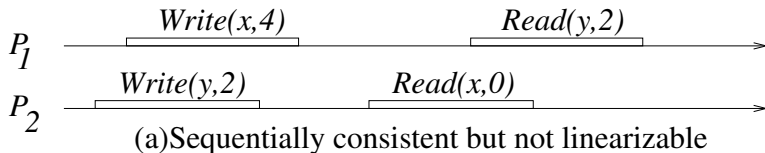
## Strict consistency

- ① A Read should return the most recent value written, per a global time axis. For operations that overlap per the global time axis, the following must hold.
- ② All operations appear to be atomic and sequentially executed.
- ③ All processors see the same order of events, equivalent to the global time ordering of non-overlapping events.



Sequential invocations and responses to each Read or Write operation.

# Strict Consistency / Linearizability: Examples



Initial values are zero. (a),(c) not linearizable. (b) is linearizable

# Linearizability: Implementation

- Simulating global time axis is expensive.
- Assume full replication, and total order broadcast support.

(shared var)

**int:** *x*;

(1) When the Memory Manager receives a *Read* or *Write* from application:

(1a) **total\_order\_broadcast** the *Read* or *Write* request to all processors;

(1b) **await** own request that was broadcast;

(1c) **perform** pending response to the application as follows

(1d)     **case** *Read*: return value from local replica;

(1e)     **case** *Write*: write to local replica and return ack to application.

(2) When the Memory Manager receives a **total\_order\_broadcast**(*Write*, *x*, *val*) from network:

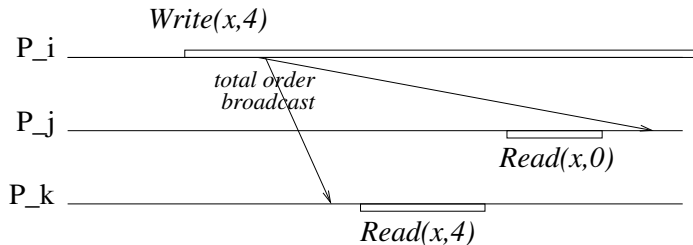
(2a) **write** *val* to local replica of *x*.

(3) When the Memory Manager receives a **total\_order\_broadcast**(*Read*, *x*) from network:

(3a) **no operation**.

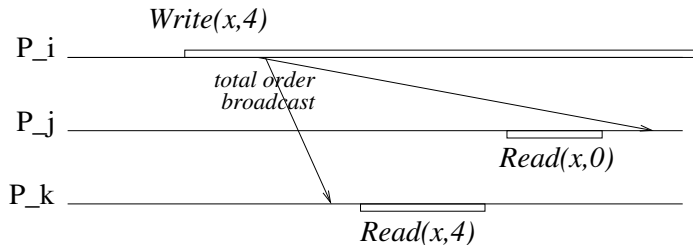
# Linearizability: Implementation (2)

- When a Read is simulated at other processes, there is a no-op.
- Why do Reads participate in total order broadcasts?
- Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



# Linearizability: Implementation (2)

- When a Read is simulated at other processes, there is a no-op.
- Why do Reads participate in total order broadcasts?
- Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



# Sequential Consistency

## Sequential Consistency.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

Any interleaving of the operations from the different processors is possible. But all processors must see *the same* interleaving. Even if two operations from different processors (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the *common* sequential order seen by all. See examples used for linearizability.

# Sequential Consistency

Only Writes participate in total order BCs. Reads do not because:

- all consecutive operations by the same processor are ordered in that same order (no pipelining), and
- *Read* operations by different processors are independent of each other; to be ordered only with respect to the *Write* operations.
- Direct simplification of the LIN algorithm.
- Reads executed atomically. Not so for Writes.
- Suitable for Read-intensive programs.



# Sequential Consistency using Local Reads

(shared var)

**int:**  $x$ ;

(1) When the Memory Manager at  $P_i$  receives a *Read* or *Write* from application:

(1a) **case** *Read*: **return** value from local replica;

(1b) **case** *Write*( $x, val$ ): **total\_order\_broadcast** $_i(Write(x, val))$  to all processors including itself.

(2) When the Memory Manager at  $P_i$  receives a **total\_order\_broadcast** $_j(Write, x, val)$  from network:

(2a) **write**  $val$  to local replica of  $x$ ;

(2b) **if**  $i = j$  **then return** ack to application.

# Sequential Consistency using Local Writes

(shared var)

**int:**  $x$ ;

(1) When the Memory Manager at  $P_i$  receives a  $Read(x)$  from application:

(1a) **if**  $counter = 0$  **then**

(1b)     **return**  $x$

(1c) **else** Keep the  $Read$  pending.

(2) When the Memory Manager at  $P_i$  receives a  $Write(x, val)$  from application:

(2a)  $counter \leftarrow counter + 1$ ;

(2b) **total\_order\_broadcast** <sub>$i$</sub>  the  $Write(x, val)$ ;

(2c) **return** ack to the application.

(3) When the Memory Manager at  $P_i$  receives a **total\_order\_broadcast** <sub>$j$</sub> ( $Write, x, val$ ) from network:

(3a) **write**  $val$  to local replica of  $x$ .

(3b) **if**  $i = j$  **then**

(3c)      $counter \leftarrow counter - 1$ ;

(3d)     **if** ( $counter = 0$  and any  $Reads$  are pending) **then**

(3e)         **perform** pending responses for the  $Reads$  to the application.

Locally issued Writes get acked immediately. Local Reads are delayed until the locally preceding Writes have been acked. All locally issued Writes are pipelined.

# Causal Consistency

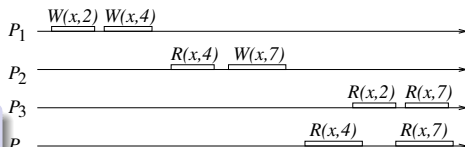
In SC, all Write ops should be seen in common order.

For *causal consistency*, only causally related Writes should be seen in common order.

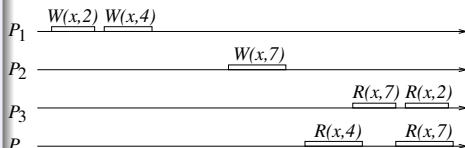
## Causal relation for shared memory systems

- At a processor, local order of events is the causal order
- A Write causally precedes Read issued by another processor if the Read returns the value written by the Write.
- The transitive closure of the above two orders is the causal order

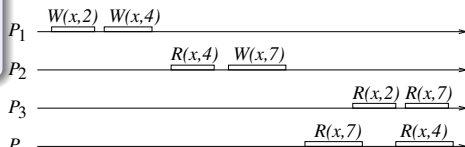
Total order broadcasts (for SC) also provide causal order in shared memory



(a) Sequentially consistent and causally consistent



(b) Causally consistent but not sequentially consistent



(c) Not causally consistent but PRAM consistent

# Pipelined RAM or Processor Consistency

## PRAM memory

Only Write ops issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

PRAM can be implemented by FIFO broadcast? PRAM memory can exhibit counter-intuitive behavior, see below.

(shared variables)

**int:**  $x, y$ ;

Process 1

...

(1a)  $x \leftarrow 4$ ;

(1b) **if**  $y = 0$  **then** **kill**( $P_2$ ).

Process 2

...

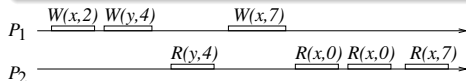
(2a)  $y \leftarrow 6$ ;

(2b) **if**  $x = 0$  **then** **kill**( $P_1$ ).

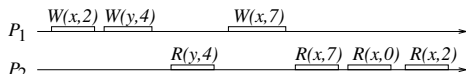
# Slow Memory

## Slow Memory

Only Write operations issued by the same processor and to the same memory location must be seen by others in that order.

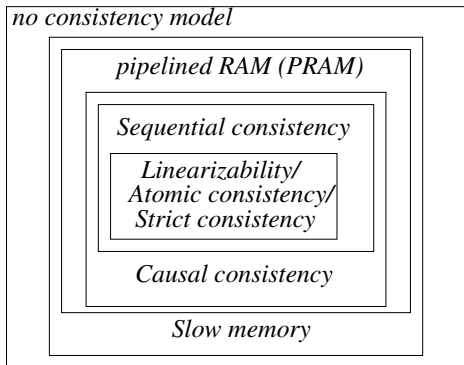


(a) Slow memory but not PRAM consistent



(b) Violation of slow memory consistency

# Hierarchy of Consistency Models



# Synchronization-based Consistency Models: Weak Consistency

- Consistency conditions apply only to special "synchronization" instructions, e.g., barrier synchronization
- Non-sync statements may be executed in any order by various processors.
- E.g., weak consistency, release consistency, entry consistency

## Weak consistency:

All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.

- Accesses to sync variables are sequentially consistent
- Access to sync variable is not permitted unless all Writes elsewhere have completed
- No data access is allowed until all previous synchronization variable accesses have been performed

Drawback: cannot tell whether beginning access to shared variables (enter CS), or finished access to shared variables (exit CS).

# Synchronization based Consistency Models: Release Consistency and Entry Consistency

Two types of synchronization Variables: *Acquire* and *Release*

## Release Consistency

- *Acquire* indicates CS is to be entered. Hence all *Writes* from other processors should be locally reflected at this instruction
- *Release* indicates access to CS is being completed. Hence, all Updates made locally should be propagated to the replicas at other processors.
- *Acquire* and *Release* can be defined on a subset of the variables.
- If no CS semantics are used, then *Acquire* and *Release* act as barrier synchronization variables.
- Lazy release consistency: propagate updates on-demand, not the PRAM way.

## Entry Consistency

- Each ordinary shared variable is associated with a synchronization variable (e.g., lock, barrier)
- For *Acquire* /*Release* on a synchronization variable, access to only those ordinary variables guarded by the synchronization variables is performed.

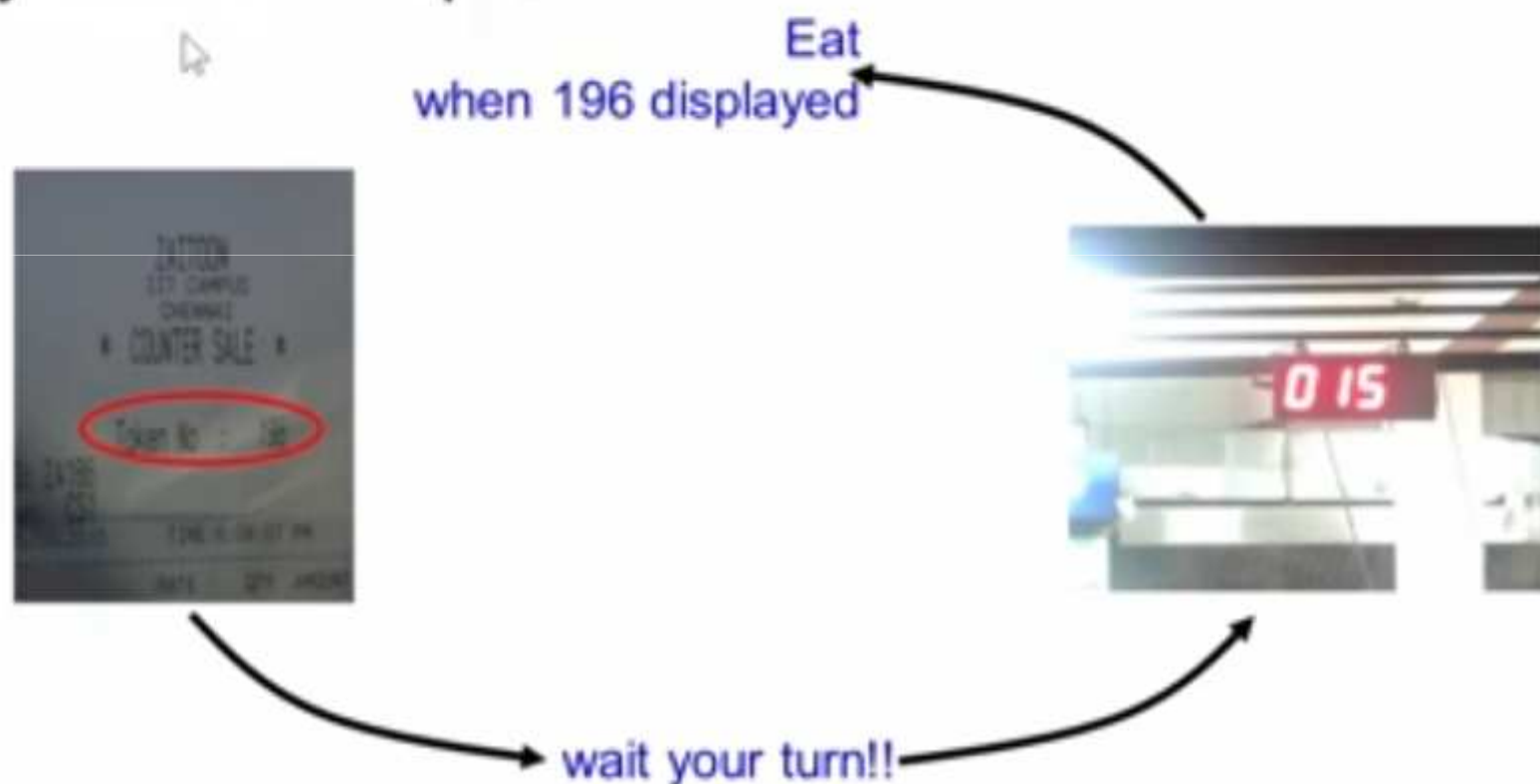


# Lamport's Bakery Algorithm for DSM

Bakery Algorithm for Mutually Exclusive access to the shared variables stored in DSM

# Bakery Algorithm

- Synchronization between  $N > 2$  processes
- By Leslie Lamport



# Simplified Bakery Algorithm

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    for(p = 0; p < N; ++p){  
        while (num[p] != 0 and num[p] < num[i]);  
    }  
}
```

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

# Simplified Bakery Algorithm (example)

Processes numbered 0 to N-1  
num is an array N integers (initially 0).  
Each entry corresponds to a process

```
lock(i){  
    num[i] = MAX(num[0], num[1], ....., num[N-1]) + 1  
    for(p = 0; p < N; ++p){  
        while (num[p] != 0 and num[p] < num[i]);  
    }  
}
```

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	4	1	2	3



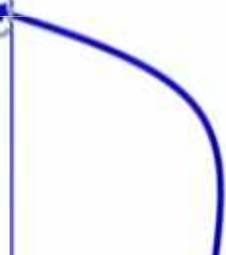
# Simplified Bakery Algorithm

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){  
    num[i] = MAX(num[0], num[1], ....., num[N-1]) + 1;  
    for(p = 0; p < N; ++p){  
        while (num[p] != 0 and num[p] < num[i]);  
    }  
}
```

critical section

```
unlock(i){  
    num[i] = 0;  
}
```




This is at the doorway!!!  
It has to be atomic  
to ensure two processes  
do not get the same token

# Simplified Bakery Algorithm (why atomic doorway?)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1;  
    for(p = 0; p < N; ++p){  
        while (num[p] != 0 and num[p] < num[i]);  
    }  
}
```

This is at the doorway!!!  
Assume it is not atomic



critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	<b>P5</b>
0	0	0	0	0




# Simplified Bakery Algorithm (why atomic doorway?)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    for(p = 0; p < N; ++p){  
        while (num[p] != 0 and num[p] < num[i]);  
    }  
}
```

This is at the doorway!!!  
Assume it is not atomic



critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	3	1	2	2

# Original Bakery Algorithm

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

Choosing ensures that a process  
is not at the doorway  
i.e., the process is not 'choosing'  
a value for num

$(a, b) < (c, d)$  which is equivalent to:  $(a < c)$  or  $((a == c) \text{ and } (b < d))$



# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	0	1	2	2

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p) < (num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	3	1	2	2

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ....., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	<b>P5</b>
0	3	0	2	2

$(a, b) < (c, d)$  which is equivalent to:  $(a < c)$  or  $((a == c) \text{ and } (b < d))$

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

check  $p < i$

P1	P2	P3	P4	P5
0	3	0	2	2

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))



# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	3	0	0	2

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	3	0	0	0

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){  
    choosing[i] = True  
    num[i] = MAX(num[0], num[1], ..., num[N-1]) + 1  
    choosing[i] = False  
    for(p = 0; p < N; ++p){  
        while (choosing[p]);  
        while (num[p] != 0 and (num[p],p)<(num[i],i));  
    }  
}
```

doorway

critical section

```
unlock(i){  
    num[i] = 0;  
}
```

P1	P2	P3	P4	P5
0	0	0	0	0

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# References

- Leslie Lamport's Bakery Algorithm Paper.  
<http://lamport.azurewebsites.net/pubs/bakery.pdf>
- NPTEL Video Lecture:  
<https://www.youtube.com/watch?v=YHQxp-XduS0>



# Chapter 18: Peer-to-peer Computing and Overlay Graphs

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

# Characteristics

- P2P network: application-level organization of the network to flexibly share resources
- All nodes are equal; communication directly between peers (no client-server)
- Allow location of arbitrary objects; no DNS servers required
- Large combined storage, CPU power, other resources, without scalability costs
- Dynamic insertion and deletion of nodes, as well as of resources, at low cost

Features	Performance
self-organizing	large combined storage, CPU power, and resources
distributed control	fast search for machines and data objects
role symmetry for nodes	scalable
anonymity	efficient management of churn
naming mechanism	selection of geographically close servers
security, authentication, trust	redundancy in storage and paths

**Table:** Desirable characteristics and performance features of P2P systems.

# Napster

Central server maintains a table with the following information of each registered client: (i) the client's address (IP) and port, and offered bandwidth, and (ii) information about the files that the client can allow to share.

- A client connects to a meta-server that assigns a lightly-loaded server.
- The client connects to the assigned server and forwards its query and identity.
- The server responds to the client with information about the users connected to it and the files they are sharing.
- On receiving the response from the server, the client chooses one of the users from whom to download a desired file. The address to enable the P2P connection between the client and the selected user is provided by the server to the client.

Users are generally anonymous to each other. The directory serves to provide the mapping from a particular host that contains the required content, to the IP address needed to download from it.

# Structured and Unstructured Overlays

- Search for data and placement of data depends on P2P overlay (which can be thought of as being below the application level overlay)
- Search is data-centric, not host-centric
- Structured P2P overlays:
  - ▶ E.g., hypercube, mesh, de Bruijn graphs
  - ▶ rigid organizational principles for object storage and object search
- Unstructured P2P overlays:
  - ▶ Loose guidelines for object search and storage
  - ▶ Search mechanisms are ad-hoc, variants of flooding and random walk
- Object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

# Data indexing

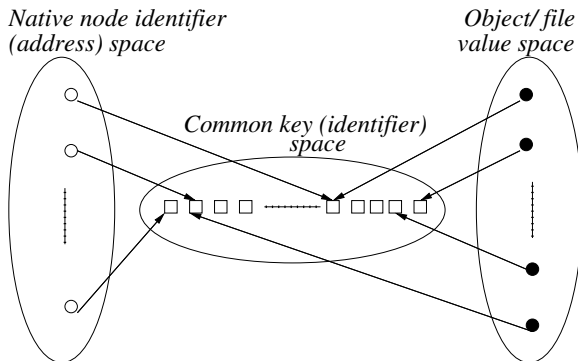
Data identified by indexing, which allows physical data independence from apps.

- Centralized indexing, e.g., versions of Napster, DNS
- Distributed indexing. Indexes to data scattered across peers. Access data through mechanisms such as Distributed Hash Tables (DHT). These differ in hash mapping, search algorithms, diameter for lookup, fault tolerance, churn resilience.
- Local indexing. Each peer indexes only the local objects. Remote objects need to be searched for. Typical DHT uses flat key space. Used commonly in unstructured overlays (E.g., Gnutella) along with flooding search or random walk search.

Another classification

- Semantic indexing - human readable, e.g., filename, keyword, database key. Supports keyword searches, range searches, approximate searches.
- Semantic-free indexing. Not human readable. Corresponds to index obtained by use of hash function.

# Simple Distributed Hash Table scheme



Mappings from node address space and object space in a simple DHT.

- Highly deterministic placement of files/data allows fast lookup.
- But file insertions/deletions under churn incurs some cost.
- Attribute search, range search, keyword search etc. not possible.

## Peer-to-Peer (P2P) Systems:

Peer-to-Peer (P2P) systems are a type of network architecture where computers, or "peers," in the network act as both clients and servers. In a P2P network, each computer can share resources (such as files or processing power) and request resources from other computers directly, without the need for a centralized server. P2P systems are often associated with file sharing and decentralized communication.

## Key Characteristics of P2P Systems:

1. **Decentralization:** In P2P systems, there is no central server that controls or manages the network. Instead, all peers have equal status, and they can communicate and share resources directly with one another.
2. **Shared Resources:** Peers in a P2P network can share files, data, or services with other peers. This makes P2P networks efficient for tasks like file sharing, where users can upload and download content from each other.
3. **Scalability:** P2P networks are often scalable because adding new peers to the network does not require significant changes to the existing infrastructure.
4. **Redundancy:** P2P networks can be more robust because multiple copies of resources can exist on different peers. If one peer goes offline, others can still provide the same resources.



Aspect	P2P Model	Client-Server Model
Resource Sharing	All peers share resources with each other.	Clients request resources from a central server.
Network Topology	Decentralized; no central server.	Centralized; a central server manages resources.
Scalability	Typically more scalable as new peers can be easily added.	May require more centralized infrastructure to scale.
Dependency on Server	No central server dependency.	Clients depend on the central server.
Resource Availability	Resource availability depends on the presence of peers with the desired resources.	Resource availability depends on the server's availability.
Network Traffic	Peers communicate directly, resulting in distributed traffic.	Traffic is routed through the central server, which can lead to bottlenecks.
Redundancy	Redundancy is achieved by the presence of multiple peers with the same resources.	Redundancy is achieved through backup servers.
Example Use Cases	File sharing (BitTorrent), distributed computing (SETI@home).	Web hosting, email services, online gaming.

# First Generation P2P – Napster

- Centralised server

# First Generation P2P – Napster

- Centralised server
- Each node registers list of files that it has to the central server

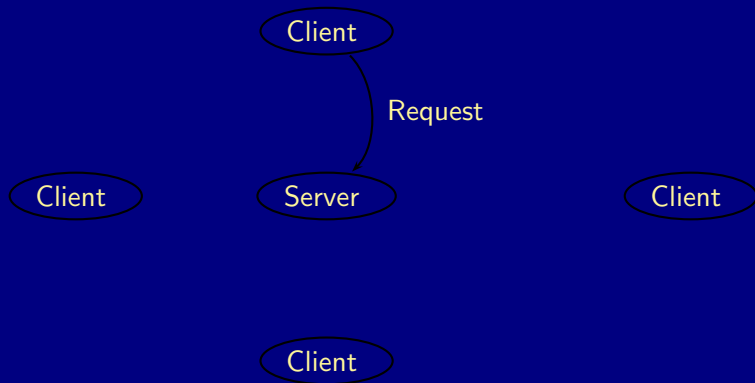
# First Generation P2P – Napster

- Centralised server
- Each node registers list of files that it has to the central server
- When a node wishes to retrieve a file it requests from the central server a list of client nodes that have that file

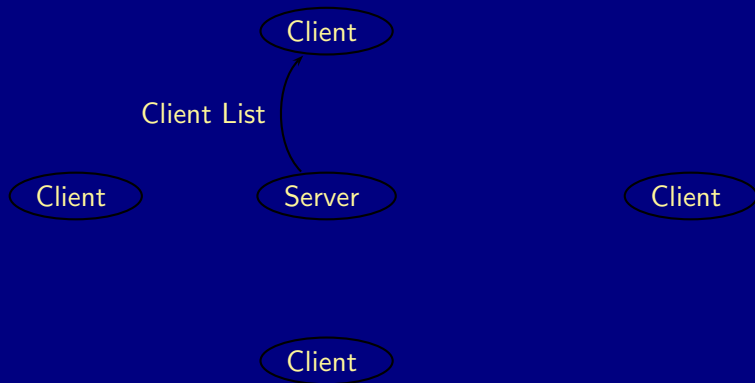
# First Generation P2P – Napster

- Centralised server
- Each node registers list of files that it has to the central server
- When a node wishes to retrieve a file it requests from the central server a list of client nodes that have that file
- Then the client picks a node from which to download the file.

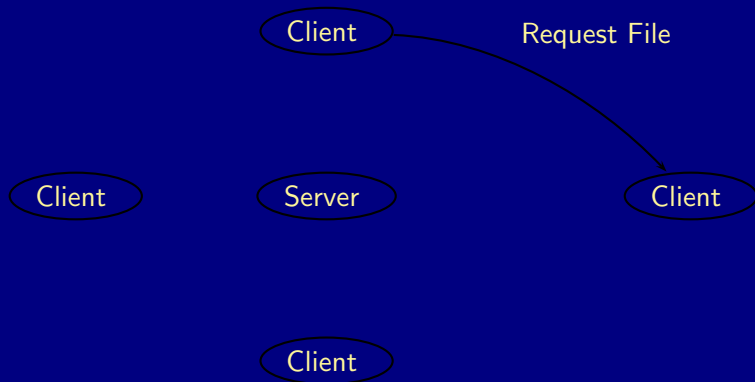
# First Generation P2P – Napster



# First Generation P2P – Napster

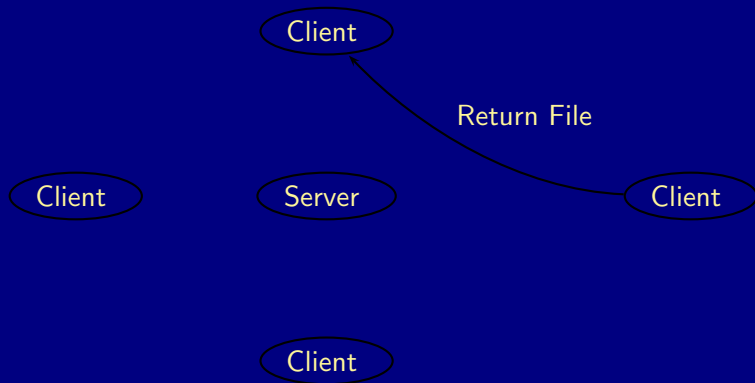


# First Generation P2P – Napster





# First Generation P2P – Napster



# First Generation P2P – Napster

- Problems with Napster like protocols

# First Generation P2P – Napster

- Problems with Napster like protocols
  - Single point of failure – The server.

# First Generation P2P – Napster

- Problems with Napster like protocols
  - Single point of failure – The server.
  - Client only downloads from one other client at a time.

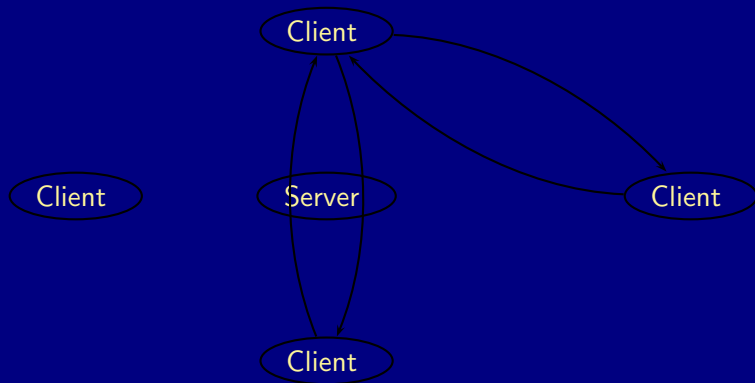
# First Generation P2P – Napster

- Problems with Napster like protocols
  - Single point of failure – The server.
  - Client only downloads from one other client at a time.
- Solutions
  - Have more than one server.

# First Generation P2P – Napster

- Problems with Napster like protocols
  - Single point of failure – The server.
  - Client only downloads from one other client at a time.
- Solutions
  - Have more than one server.
  - Make the clients more complicated and download from multiple clients (essentially what Bit-torrent does)

# First Generation P2P



# First Generation P2P – Gnutella

- Again files are distributed across the network



# First Generation P2P – Gnutella

- Again files are distributed across the network
- But no central server

# First Generation P2P – Gnutella

- Again files are distributed across the network
- But no central server
- A node must know the IP address of at least one other Gnutella node. Clients initialised with a set of working nodes

# First Generation P2P – Gnutella

- Again files are distributed across the network
- But no central server
- A node must know the IP address of at least one other Gnutella node. Clients initialised with a set of working nodes
- Each node request each node in its working set

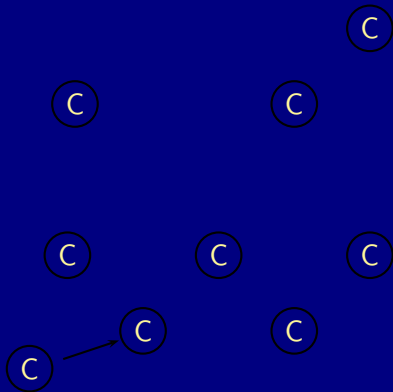
# First Generation P2P – Gnutella

- Again files are distributed across the network
- But no central server
- A node must know the IP address of at least one other Gnutella node. Clients initialised with a set of working nodes
- Each node request each node in its working set
- If a node receives a request either:
  - The file is there
  - Otherwise the request is propagated on

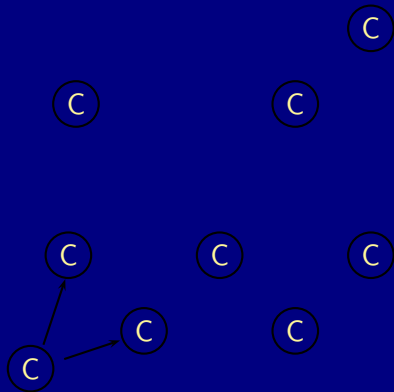
# First Generation P2P – Gnutella

- Again files are distributed across the network
- But no central server
- A node must know the IP address of at least one other Gnutella node. Clients initialised with a set of working nodes
- Each node request each node in its working set
- If a node receives a request either:
  - The file is there
  - Otherwise the request is propagated on
- Requests have a lifetime TTL (Time to live).

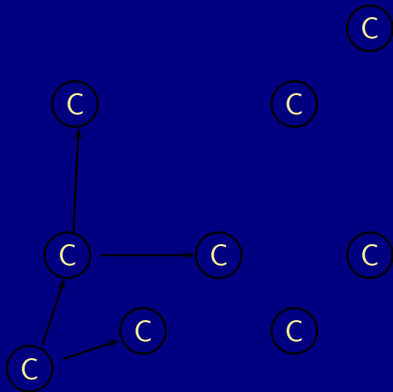
# First Generation P2P – Gnutella



# First Generation P2P – Gnutella

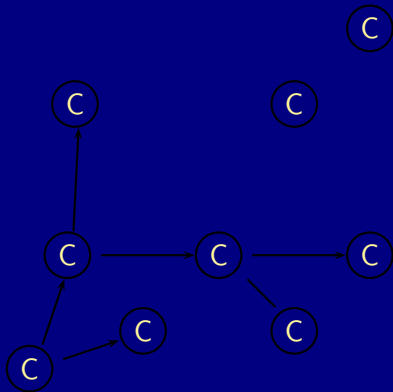


# First Generation P2P – Gnutella

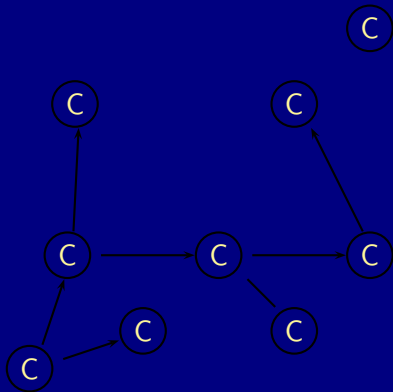




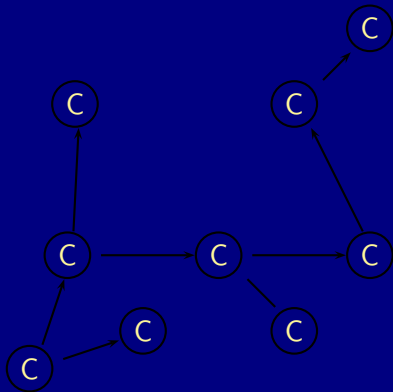
# First Generation P2P – Gnutella



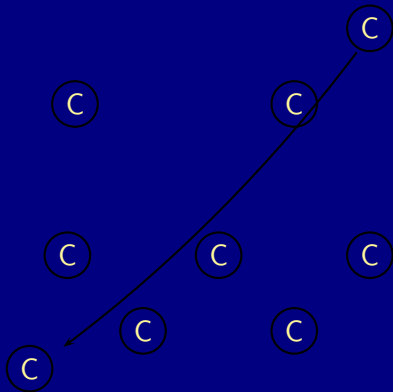
# First Generation P2P – Gnutella



# First Generation P2P – Gnutella



# First Generation P2P – Gnutella



# First Generation P2P – Scalability

- Napster did not last long enough to test scalability issues, but

# First Generation P2P – Scalability

- Napster did not last long enough to test scalability issues, but
- Think of Google with a central server, scalability is less of a problem today.

# First Generation P2P – Scalability

- Napster did not last long enough to test scalability issues, but
- Think of Google with a central server, scalability is less of a problem today.
- Gnutella essentially the protocol tries to find a node by flooding the network.

# First Generation P2P – Scalability

- Napster did not last long enough to test scalability issues, but
- Think of Google with a central server, scalability is less of a problem today.
- Gnutella essentially the protocol tries to find a node by flooding the network.
- Gnutella can have the problem that the network has more request messages floating around than anything else.



# First Generation P2P – Scalability

- Napster did not last long enough to test scalability issues, but
- Think of Google with a central server, scalability is less of a problem today.
- Gnutella essentially the protocol tries to find a node by flooding the network.
- Gnutella can have the problem that the network has more request messages floating around than anything else.
- Instead of flooding do a random walk from node to node, works but it can take a can take a long time to find the file.

# The second generation of P2P systems

Issue a P2P file sharing systems must address:

File placement Where to publish the file to be shared by others?

# The second generation of P2P systems

Issue a P2P file sharing systems must address:

File placement Where to publish the file to be shared by others?

File Look up Given a named item, how do you find it or download it?

# The second generation of P2P systems

Issue a P2P file sharing systems must address:

File placement Where to publish the file to be shared by others?

File Look up Given a named item, how do you find it or download it?

Scalability How does the performance degrade with the network size?

# The second generation of P2P systems

Issue a P2P file sharing systems must address:

File placement Where to publish the file to be shared by others?

File Look up Given a named item, how do you find it or download it?

Scalability How does the performance degrade with the network size?

Self-Organization How does the network handle nodes joining and leaving the network?

# The second generation of P2P systems

Issue a P2P file sharing systems must address:

File placement Where to publish the file to be shared by others?

File Look up Given a named item, how do you find it or download it?

Scalability How does the performance degrade with the network size?

Self-Organization How does the network handle nodes joining and leaving the network?

# The second generation of P2P systems

Additionally users often find attractive:

Censorship resistance How does the network function if nodes are shut down in an attempt to censor items?

# The second generation of P2P systems

Additionally users often find attractive:

Censorship resistance How does the network function if nodes are shut down in an attempt to censor items?

Fault-tolerance How can performance be kept in the presence of node failures.



# The second generation of P2P systems

Additionally users often find attractive:

Censorship resistance How does the network function if nodes are shut down in an attempt to censor items?

Fault-tolerance How can performance be kept in the presence of node failures.

Free-rider elimination Discourage nodes that only download and never upload.

# The second generation of P2P systems

Additionally users often find attractive:

Censorship resistance How does the network function if nodes are shut down in an attempt to censor items?

Fault-tolerance How can performance be kept in the presence of node failures.

Free-rider elimination Discourage nodes that only download and never upload.

# Overlay networks

- Gnutella type protocols flood the network with lots of request.

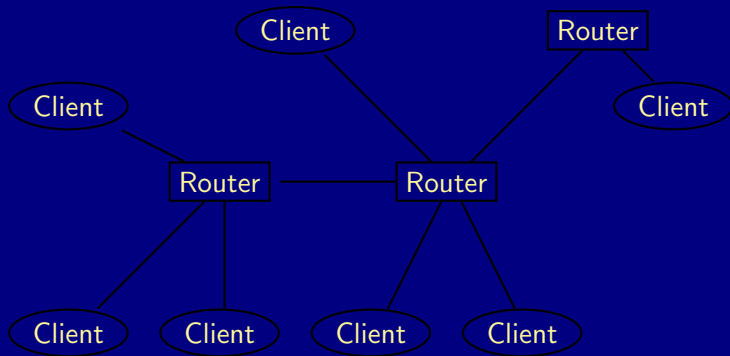
# Overlay networks

- Gnutella type protocols flood the network with lots of request.
- What is needed is some map that of nodes in the network that have files.

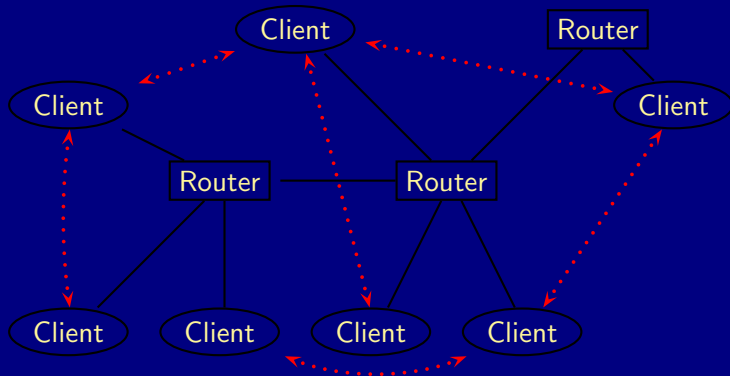
# Overlay networks

- Gnutella type protocols flood the network with lots of request.
- What is needed is some map that of nodes in the network that have files.
- The basic idea is that of an overlay network, a network over a network.

# Overlay networks



# Overlay networks



# Overlay Networks

- The overlay network has a different notion of neighbour to the underlying network.



# Overlay Networks

- The overlay network has a different notion of neighbour to the underlying network.
- In the overlay network we need some way of storing routing tables and a routing algorithm.

# Bit torrent

- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.

# Bit torrent

- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*

# Bit torrent

- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*
- Each peer can have some or all of the chunks

# Bit torrent

- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*
- Each peer can have some or all of the chunks
- A *seeding peer* has all the chunks.

# Bit torrent

- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*
- Each peer can have some or all of the chunks
- A *seeding peer* has all the chunks.
- A *download peer* has some of the chunks.

# Bit torrent

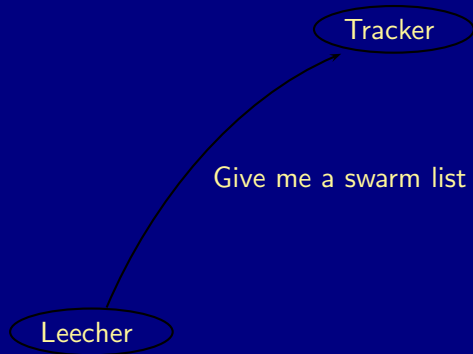
- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*
- Each peer can have some or all of the chunks
- A *seeding peer* has all the chunks.
- A *download peer* has some of the chunks.
- The idea is that even while a peer is downloading it can still be serving chunks.

# Bit torrent

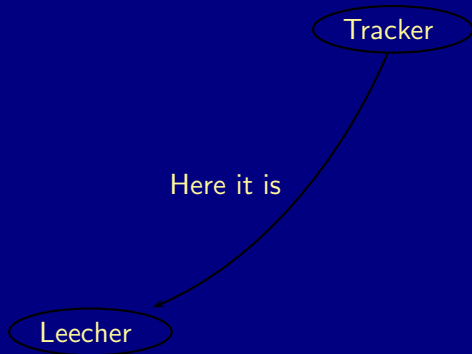
- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*
- Each peer can have some or all of the chunks
- A *seeding peer* has all the chunks.
- A *download peer* has some of the chunks.
- The idea is that even while a peer is downloading it can still be serving chunks.
- Each chunk has a hash to verify if it has been downloaded properly (stops people injecting bogus chunks).



# Bit torrent

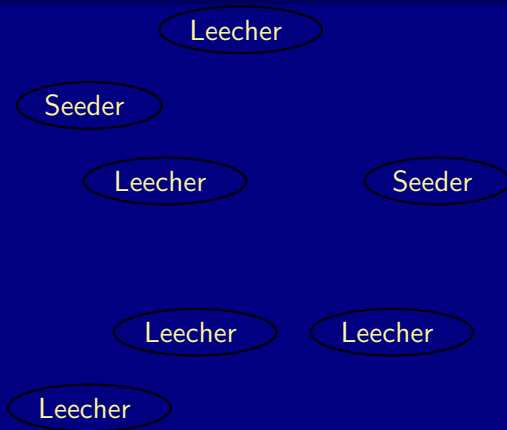


# Bit torrent



Note that the tracker need not give all the files in the swarm.

# Bit torrent



# Bit torrent

- The actual mechanism of how the client downloads from the current list of seeders and leechers (the swarm) can be quite complicated.

# Bit torrent

- The actual mechanism of how the client downloads from the current list of seeders and leechers (the swarm) can be quite complicated.
- Essentially the client asks each other client what pieces does it have?

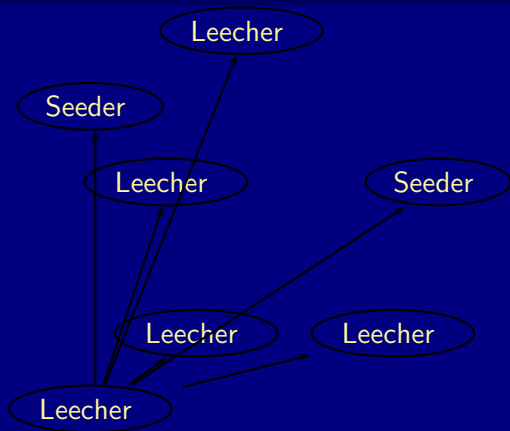
# Bit torrent

- The actual mechanism of how the client downloads from the current list of seeders and leechers (the swarm) can be quite complicated.
- Essentially the client asks each other client what pieces does it have?
- Then according to some strategy the client then asks for chunks form the other members of the swarm.

# Bit torrent

- The actual mechanism of how the client downloads from the current list of seeders and leechers (the swarm) can be quite complicated.
- Essentially the client asks each other client what pieces does it have?
- Then according to some strategy the client then asks for chunks from the other members of the swarm.
- Tit for Tat, means that you don't have to answer a request if you not getting something back from requester (bandwidth). This can make start up times a bit slow.

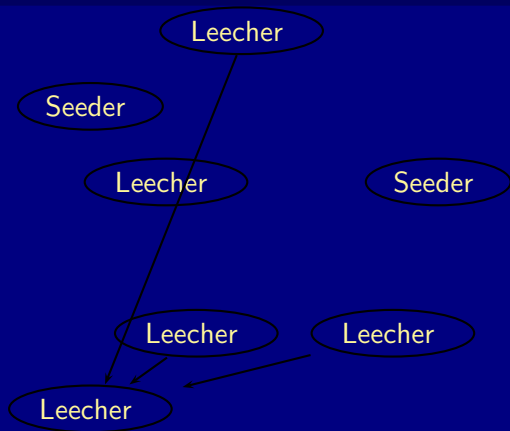
# Bit torrent



- What chunks do you have?

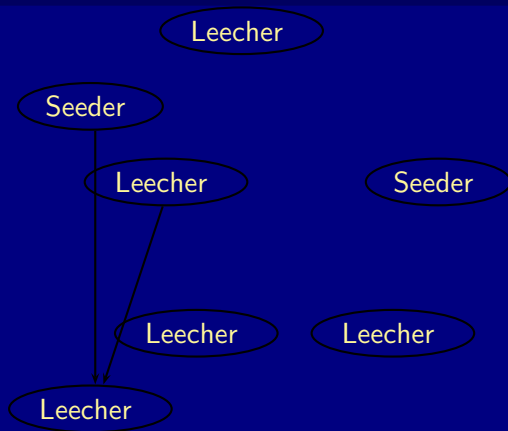


# Bit torrent



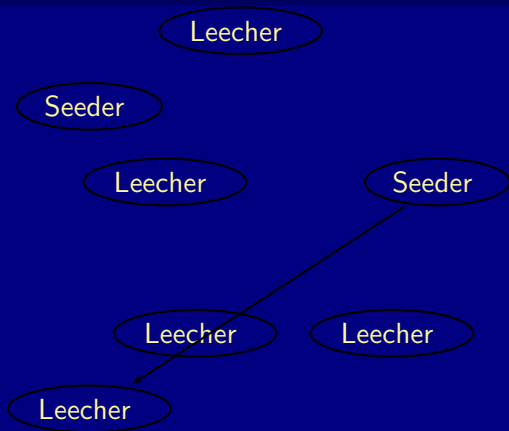
- Here is a list of chunks that I have.

# Bit torrent



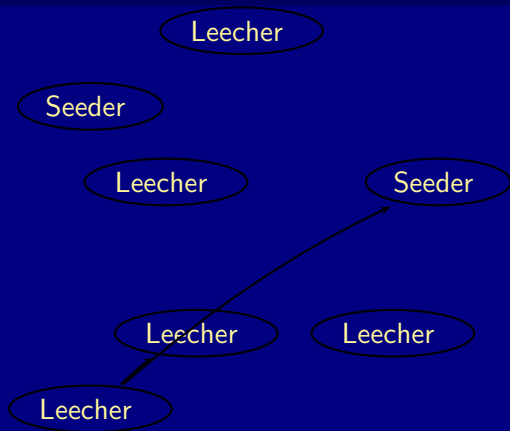
- Here is a list of chunks that I have.

# Bit torrent



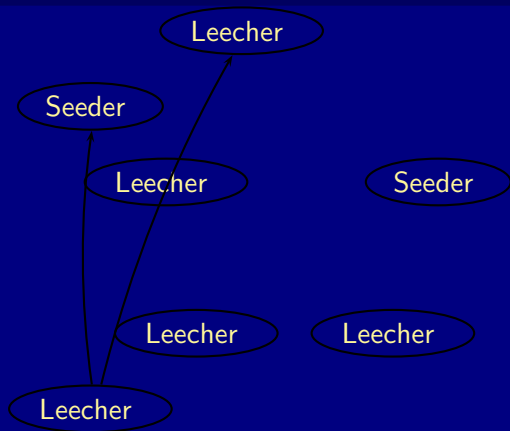
- Here is a list of chunks that I have.

# Bit torrent



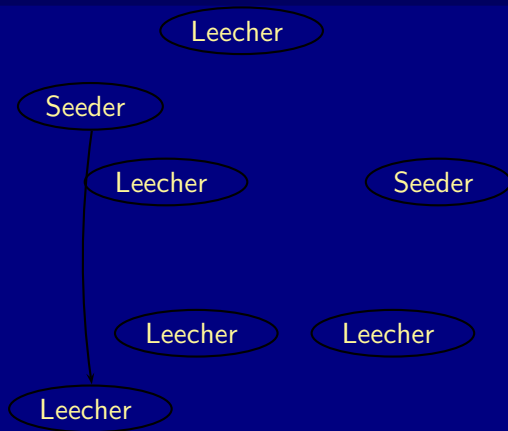
- Request some chunk

# Bit torrent



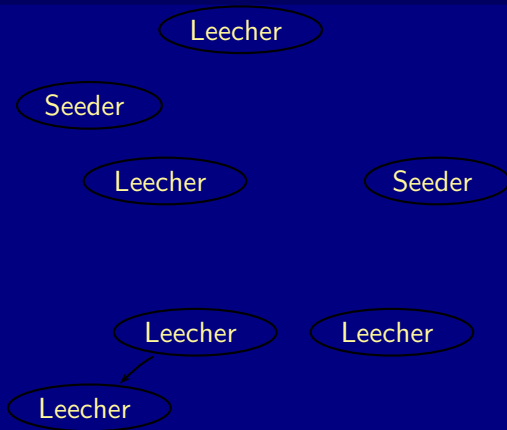
- Request some chunk

# Bit torrent



- Here are the requested chunks.

# Bit torrent



- Here are the requested chunks.

# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.



# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.
- Client might periodically ask for the chunk list from members of the swarm.

# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.
- Client might periodically ask for the chunk list from members of the swarm.
- You don't have to serve a downloaded chunk.

# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.
- Client might periodically ask for the chunk list from members of the swarm.
- You don't have to serve a downloaded chunk. There are many reasons why you might not:
  - You don't have the bandwidth.

# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.
- Client might periodically ask for the chunk list from members of the swarm.
- You don't have to serve a downloaded chunk. There are many reasons why you might not:
  - You don't have the bandwidth.
  - Tit for tat scheme says no.

# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.
- Client might periodically ask for the chunk list from members of the swarm.
- You don't have to serve a downloaded chunk. There are many reasons why you might not:
  - You don't have the bandwidth.
  - Tit for tat scheme says no.
- The idea is that the more you upload the better service you have.

# Bit torrent

Bit torrent like protocols are used in quite a few places:

- Games
  - Blizzard's World of Warcraft uses bit torrent to deliver updates
  - GnuZ The Duel (online multiplayer shot and kill game)
- Bit Torrent Inc. Legal version of Bit torrent download.
- Amazon S3 uses bit torrent in parts.
- Lots of Linux distributions offer bit-torrent downloads.

# Security

Two aspects:

- Verification of identities and verification of money (if been used as an incentive mechanism).

# Security

Two aspects:

- Verification of identities and verification of money (if been used as an incentive mechanism).
- This can be solved using standard techniques from cryptography, public/private keys.



# Security

Two aspects:

- Verification of identities and verification of money (if been used as an incentive mechanism).
- This can be solved using standard techniques from cryptography, public/private keys.
- Secure Storage, is a bit harder.

# Secure Storage

Self-Certifying Data Use cryptographic hash.

# Secure Storage

Self-Certifying Data Use cryptographic hash.

Information Dispersal Files encoded into  $m$  blocks s.t. any  $n$  is sufficient to reassemble the original data with  $m < n$ .

# Secure Storage

Self-Certifying Data Use cryptographic hash.

Information Dispersal Files encoded into  $m$  blocks s.t. any  $n$  is sufficient to reassemble the original data with  $m < n$ .

Secret Sharing Encrypt the data into  $l$  shares, so that any  $k$  nodes can decrypt but not  $k - 1$ .

# Secure Storage

Self-Certifying Data Use cryptographic hash.

Information Dispersal Files encoded into  $m$  blocks s.t. any  $n$  is sufficient to reassemble the original data with  $m < n$ .

Secret Sharing Encrypt the data into  $l$  shares, so that any  $k$  nodes can decrypt but not  $k - 1$ .

Other topics, *secure routing, distributed stenographic file systems*

# Anonymity

- With bit-torrent it is easy to find a list of people downloading a file.

# Anonymity

- With bit-torrent it is easy to find a list of people downloading a file. Just connect and look at the list of peers.
- Various types of anonymity are desirable:
  - hide the author or publisher of the content

# Anonymity

- With bit-torrent it is easy to find a list of people downloading a file. Just connect and look at the list of peers.
- Various types of anonymity are desirable:
  - hide the author or publisher of the content
  - hide the identity of a node storing the content
  - hide the identity and details of the content
  - hide details of queries for content.



# Anonymity

Freenet peer-to-peer content distribution system that makes it infeasible to discover the true origin or destination of a file passing through its network.

Onion routing provides a mechanism for anonymous connection between nodes (neither node knows the identity of each other but messages still get through).

Note that these schemes can be quite sophisticated. Via the use of techniques from cryptography it can be impossible (almost) to break the anonymity. It is more complex than just throwing away server logs.

## Other uses of P2P

Skype uses peer-to-peer protocol to forward phone calls around the net. Closed protocol, not sure how it works.

Joost Peer-to-peer internet television.

OcenStore <http://oceanstore.cs.berkeley.edu/> large scalable, fault tolerant storage system.

Distributed Databases takes files up to the next level.

Distributed Computation Seti@Home, look for messages from the little green men, or folding@home find out how proteins fold.

## Peer-to-Peer (P2P) Systems:

Peer-to-Peer (P2P) systems are a type of network architecture where computers, or "peers," in the network act as both clients and servers. In a P2P network, each computer can share resources (such as files or processing power) and request resources from other computers directly, without the need for a centralized server. P2P systems are often associated with file sharing and decentralized communication.

## Key Characteristics of P2P Systems:

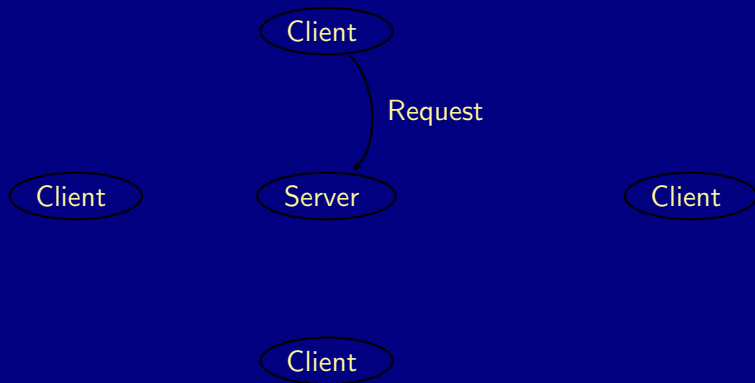
1. **Decentralization:** In P2P systems, there is no central server that controls or manages the network. Instead, all peers have equal status, and they can communicate and share resources directly with one another.
2. **Shared Resources:** Peers in a P2P network can share files, data, or services with other peers. This makes P2P networks efficient for tasks like file sharing, where users can upload and download content from each other.
3. **Scalability:** P2P networks are often scalable because adding new peers to the network does not require significant changes to the existing infrastructure.
4. **Redundancy:** P2P networks can be more robust because multiple copies of resources can exist on different peers. If one peer goes offline, others can still provide the same resources.

Aspect	P2P Model	Client-Server Model
Resource Sharing	All peers share resources with each other.	Clients request resources from a central server.
Network Topology	Decentralized; no central server.	Centralized; a central server manages resources.
Scalability	Typically more scalable as new peers can be easily added.	May require more centralized infrastructure to scale.
Dependency on Server	No central server dependency.	Clients depend on the central server.
Resource Availability	Resource availability depends on the presence of peers with the desired resources.	Resource availability depends on the server's availability.
Network Traffic	Peers communicate directly, resulting in distributed traffic.	Traffic is routed through the central server, which can lead to bottlenecks.
Redundancy	Redundancy is achieved by the presence of multiple peers with the same resources.	Redundancy is achieved through backup servers.
Example Use Cases	File sharing (BitTorrent), distributed computing (SETI@home).	Web hosting, email services, online gaming.

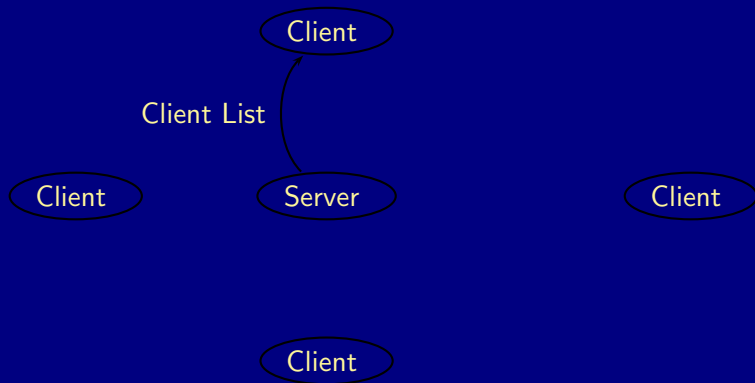
# First Generation P2P – Napster

- Centralised server
- Each node registers list of files that it has to the central server
- When a node wishes to retrieve a file it requests from the central server a list of client nodes that have that file

# First Generation P2P – Napster

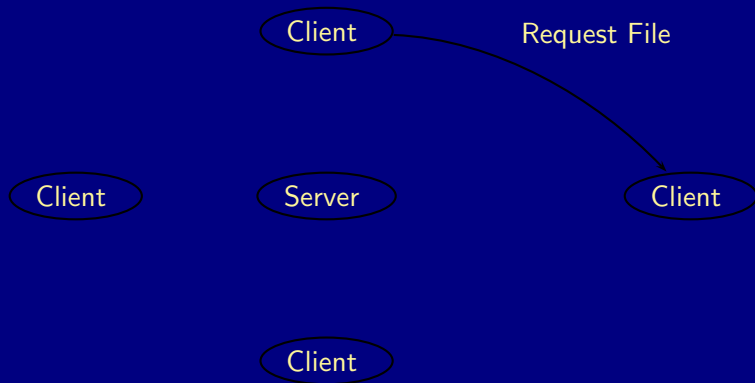


# First Generation P2P – Napster

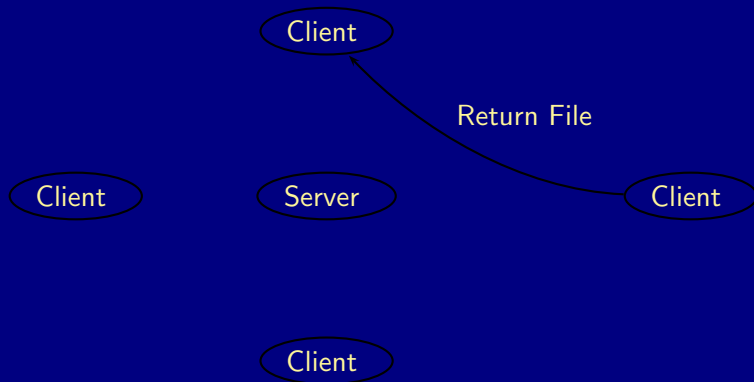




# First Generation P2P – Napster



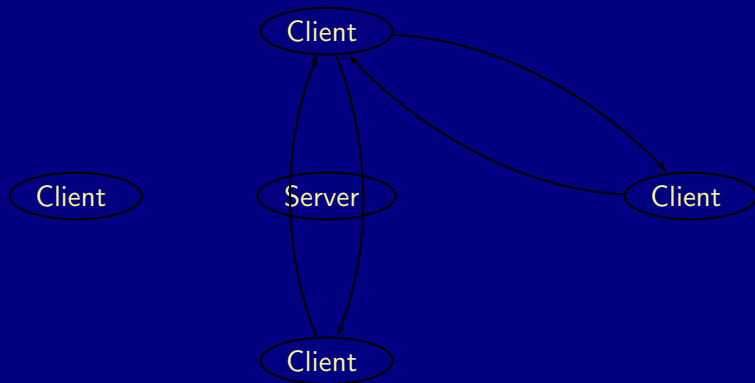
# First Generation P2P – Napster



# First Generation P2P – Napster

- Problems with Napster like protocols
  - Single point of failure – The server.
  - Client only downloads from one other client at a time.
- Solutions
  - Have more than one server.
  - Make the clients more complicated and download from multiple clients (essentially what Bit-torrent does)

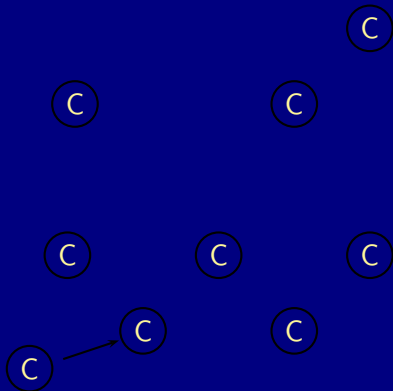
# First Generation P2P



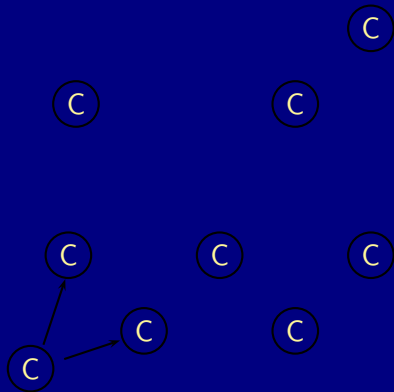
# First Generation P2P – Gnutella

- Again files are distributed across the network
- But no central server
- A node must know the IP address of at least one other Gnutella node. Clients initialised with a set of working nodes
- Each node request each node in its working set
- If a node receives a request either:
  - The file is there
  - Otherwise the request is propagated on
- Requests have a lifetime TTL (Time to live).

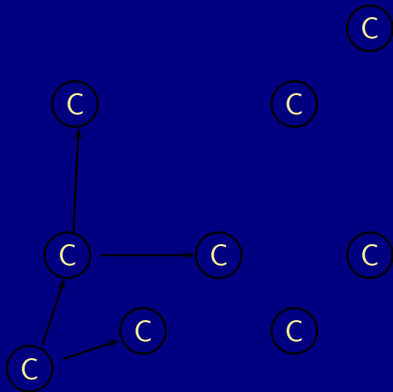
# First Generation P2P – Gnutella



# First Generation P2P – Gnutella

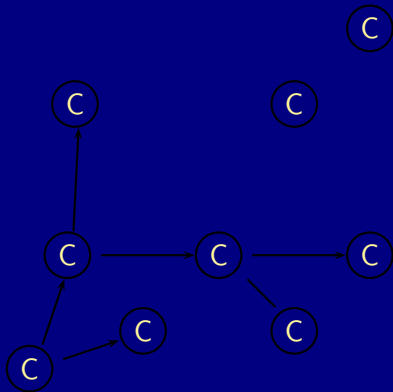


# First Generation P2P – Gnutella

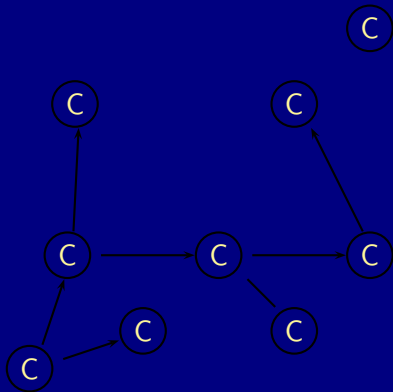




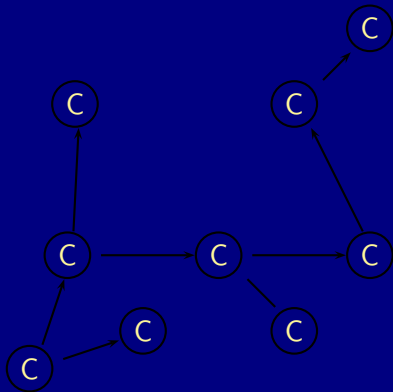
# First Generation P2P – Gnutella



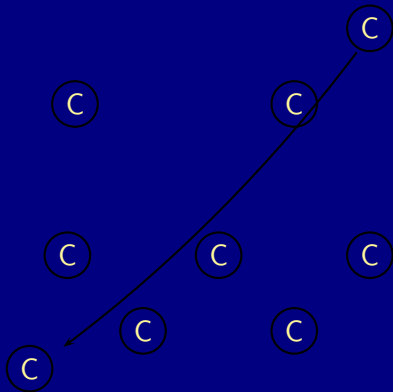
# First Generation P2P – Gnutella



# First Generation P2P – Gnutella



# First Generation P2P – Gnutella



# First Generation P2P – Scalability

- Napster did not last long enough to test scalability issues, but
- Think of Google with a central server, scalability is less of a problem today.
- Gnutella essentially the protocol tries to find a node by flooding the network.
- Gnutella can have the problem that the network has more request messages floating around than anything else.
- Instead of flooding do a random walk from node to node, works but it can take a can take a long time to find the file.

# The second generation of P2P systems

Issue a P2P file sharing systems must address:

File placement Where to publish the file to be shared by others?

File Look up Given a named item, how do you find it or download it?

Scalability How does the performance degrade with the network size?

Self-Organization How does the network handle nodes joining and leaving the network?

# The second generation of P2P systems

Additionally users often find attractive:

Censorship resistance How does the network function if nodes are shut down in an attempt to censor items?

Fault-tolerance How can performance be kept in the presence of node failures.

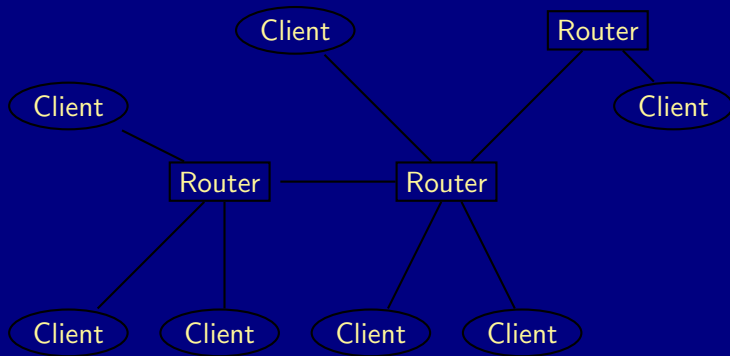
Free-rider elimination Discourage nodes that only download and never upload.

# Overlay networks

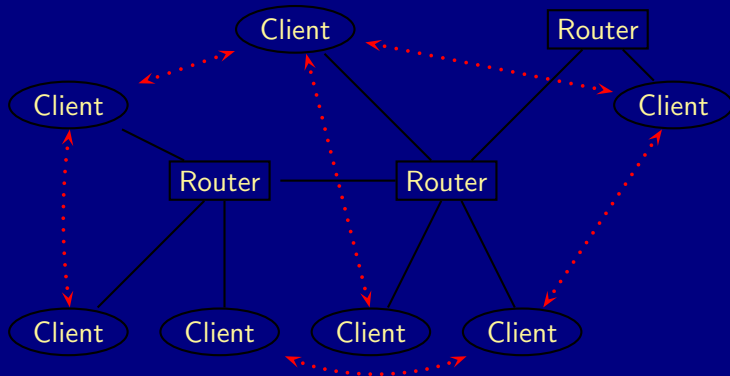
- Gnutella type protocols flood the network with lots of request.
- What is needed is some map that of nodes in the network that have files.
- The basic idea is that of an overlay network, a network over a network.



# Overlay networks



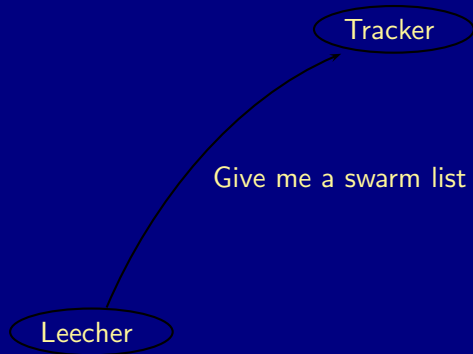
# Overlay networks



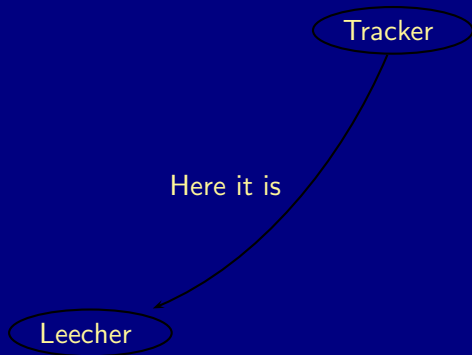
# Bit torrent

- Bit torrent uses a central server (for each file) called a *tracker* which keeps track of all peers that have the file. Note that generally the tracker does not actually have the file to be downloaded.
- A file is divided up into a number of *chunks*
- Each peer can have some or all of the chunks
- A *seeding peer* has all the chunks.
- A *download peer* has some of the chunks.
- The idea is that even while a peer is downloading it can still be serving chunks.
- Each chunk has a hash to verify if it has been downloaded properly (stops people injecting bogus chunks).

# Bit torrent

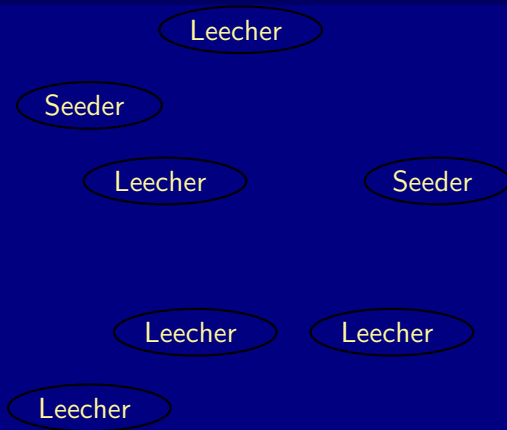


# Bit torrent



Note that the tracker need not give all the files in the swarm.

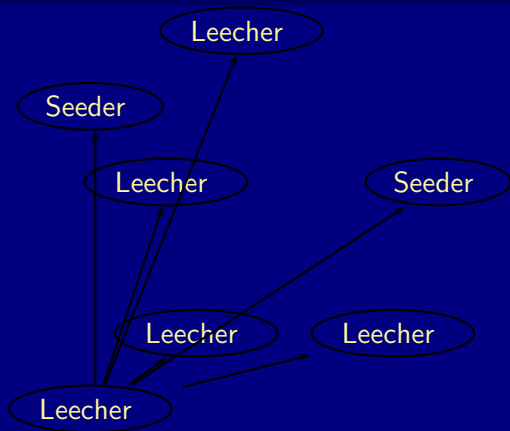
# Bit torrent



# Bit torrent

- The actual mechanism of how the client downloads from the current list of seeders and leechers (the swarm) can be quite complicated.
- Essentially the client asks each other client what pieces does it have?
- Then according to some strategy the client then asks for chunks from the other members of the swarm.
- Tit for Tat, means that you don't have to answer a request if you not getting something back from requester (bandwidth). This can make start up times a bit slow.

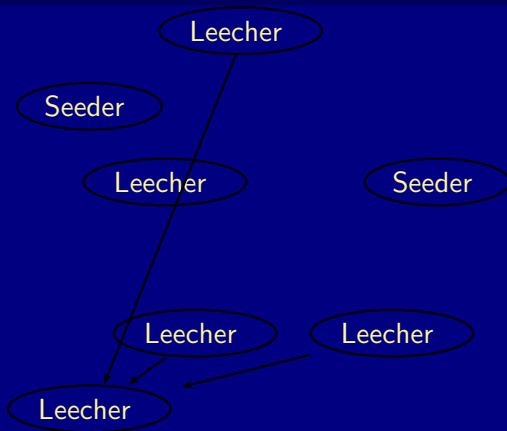
# Bit torrent



- What chunks do you have?

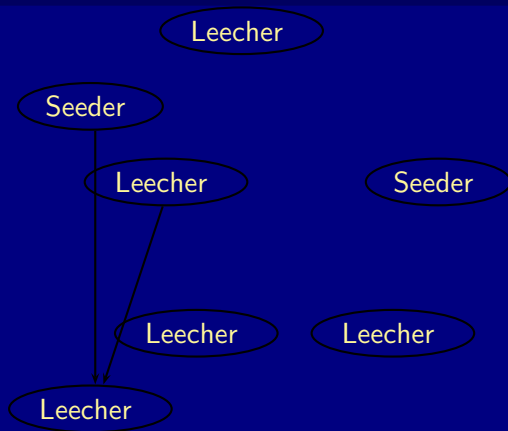


# Bit torrent



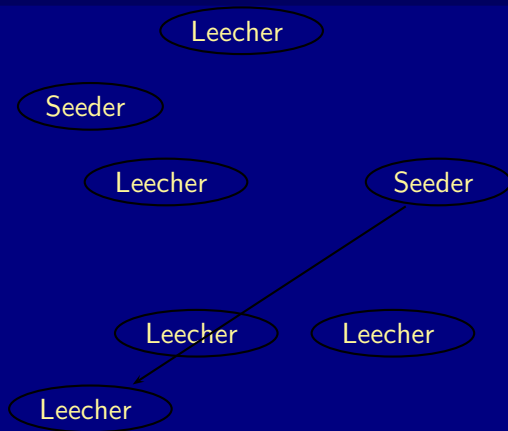
- Here is a list of chunks that I have.

# Bit torrent



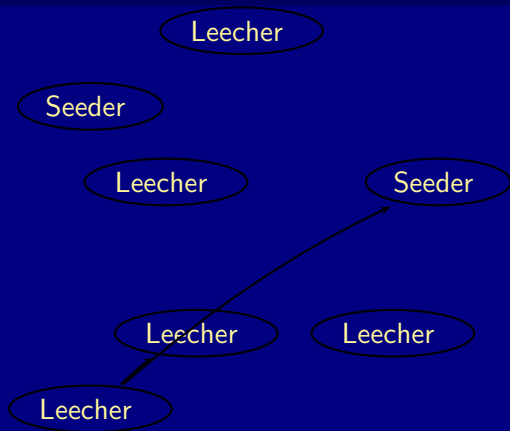
- Here is a list of chunks that I have.

# Bit torrent



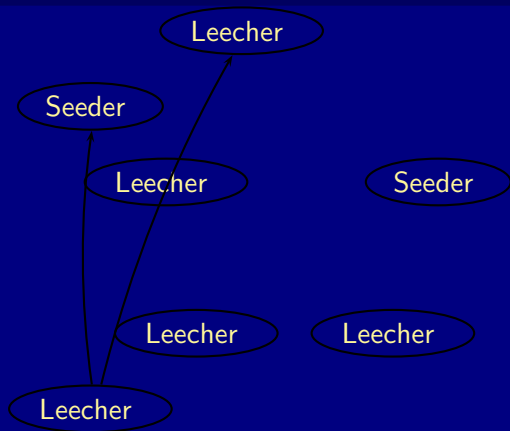
- Here is a list of chunks that I have.

# Bit torrent



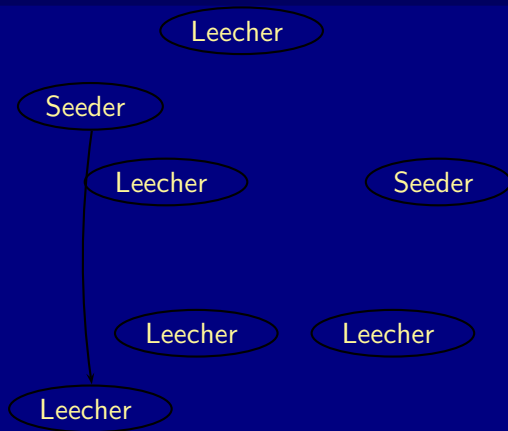
- Request some chunk

# Bit torrent



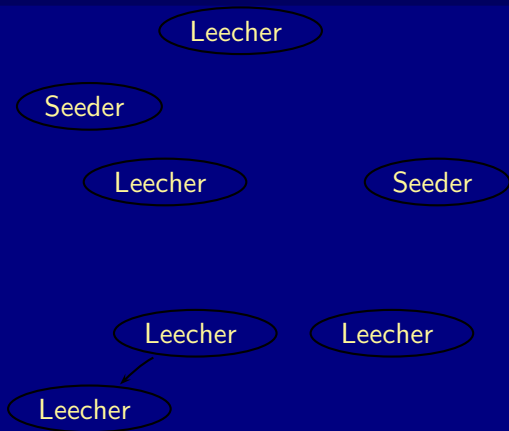
- Request some chunk

# Bit torrent



- Here are the requested chunks.

# Bit torrent



- Here are the requested chunks.

# Bit torrent

- Note, at all times the downloading client is serving requests for chunks. Helps with Tit for Tat.
- Client might periodically ask for the chunk list from members of the swarm.
- You don't have to serve a downloaded chunk. There are many reasons why you might not:
  - You don't have the bandwidth.
  - Tit for tat scheme says no.
- The idea is that the more you upload the better service you have.



# Bit torrent

Bit torrent like protocols are used in quite a few places:

- Games
  - Blizzard's World of Warcraft uses bit torrent to deliver updates
  - GnuZ The Duel (online multiplayer shot and kill game)
- Bit Torrent Inc. Legal version of Bit torrent download.
- Amazon S3 uses bit torrent in parts.
- Lots of Linux distributions offer bit-torrent downloads.

# Security

Two aspects:

- Verification of identities and verification of money (if been used as an incentive mechanism).
- This can be solved using standard techniques from cryptography, public/private keys.
- Secure Storage, is a bit harder.

# Secure Storage

Self-Certifying Data Use cryptographic hash.

Information Dispersal Files encoded into  $m$  blocks s.t. any  $n$  is sufficient to reassemble the original data with  $m < n$ .

Secret Sharing Encrypt the data into  $l$  shares, so that any  $k$  nodes can decrypt but not  $k - 1$ .

Other topics, *secure routing, distributed stenographic file systems*

# Anonymity

- With bit-torrent it is easy to find a list of people downloading a file. Just connect and look at the list of peers.
- Various types of anonymity are desirable:
  - hide the author or publisher of the content
  - hide the identity of a node storing the content
  - hide the identity and details of the content
  - hide details of queries for content.

# Anonymity

Freenet peer-to-peer content distribution system that makes it infeasible to discover the true origin or destination of a file passing through its network.

Onion routing provides a mechanism for anonymous connection between nodes (neither node knows the identity of each other but messages still get through).

Note that these schemes can be quite sophisticated. Via the use of techniques from cryptography it can be impossible (almost) to break the anonymity. It is more complex than just throwing away server logs.

## Other uses of P2P

Skype uses peer-to-peer protocol to forward phone calls around the net. Closed protocol, not sure how it works.

Joost Peer-to-peer internet television.

OcenStore <http://oceanstore.cs.berkeley.edu/> large scalable, fault tolerant storage system.

Distributed Databases takes files up to the next level.

Distributed Computation Seti@Home, look for messages from the little green men, or folding@home find out how proteins fold.

# Structured vs. unstructured overlays



## Structured Overlays:

- structure  $\implies$  placement of files is highly deterministic, file insertions and deletions have some overhead
- Fast lookup
- Hash mapping based on a single characteristic (e.g., file name)
- Range queries, keyword queries, attribute queries difficult to support

## Unstructured Overlays:

- No structure for overlay  $\implies$  no structure for data/file placement
- Node join/departures are easy; local overlay simply adjusted
- Only local indexing used
- File search entails high message overhead and high delays
- Complex, keyword, range, attribute queries supported
- Some overlay topologies naturally emerge:
  - ▶ Power Law Random Graph (PLRG) where node degrees follow a power law. Here, if the nodes are ranked in terms of degree, then the  $i^{th}$  node has  $c/i^\alpha$  neighbors, where  $c$  is a constant.
  - ▶ simple random graph: nodes typically have a uniform degree

# Unstructured Overlays: Properties

- Semantic indexing possible  $\implies$  keyword, range, attribute-based queries
- Easily accommodate high churn
- Efficient when data is replicated in network
- Good if user satisfied with "best-effort" search
- Network is not so large as to cause high delays in search

## Gnutella features

- A joiner connects to some standard nodes from Gnutella directory
- *Ping* used to discover other hosts; allows new host to announce itself
- *Pong* in response to *Ping*; *Pong* contains IP, port #, max data size for download
- *Query* msgs used for flooding search; contains required parameters
- *QueryHit* are responses. If data is found, this message contains the IP, port #, file size, download rate, etc. Path used is reverse path of *Query*.



# Search in Unstructured Overlays

Consider a system with  $n$  nodes and  $m$  objects. Let  $q_i$  be the popularity of object  $i$ , as measured by the fraction of all queries that are queries for object  $i$ . All objects may be equally popular, or more realistically, a Zipf-like power law distribution of popularity exists. Thus,

$$\sum_{i=1}^m q_i = 1 \quad (1)$$

$$\text{Uniform: } q_i = 1/m; \quad \text{Zipf-like: } q_i \propto i^{-\alpha} \quad (2)$$

Let  $r_i$  be the number of replicas of object  $i$ , and let  $p_i$  be the fraction of all objects that are replicas of  $i$ . Three static replication strategies are: uniform, proportional, and square root. Thus,

$$\sum_{i=1}^m r_i = R; \quad p_i = r_i/R \quad (3)$$

$$\text{Uniform: } r_i = R/m; \quad \text{Proportional: } r_i \propto q_i; \quad \text{Square-root: } r_i \propto \sqrt{q_i} \quad (4)$$

Under uniform replication, all objects have an equal number of replicas; hence the performance for all query rates is the same. With a uniform query rate, proportional and square-root replication schemes reduce to the uniform replication scheme.

# Search in Unstructured Overlays

For an object search, some of the metrics of efficiency:

- probability of success of finding the queried object.
- delay or the number of hops in finding an object.
- the number of messages processed by each node in a search.
- node coverage, the fraction of (distinct) nodes visited
- *message duplication*, which is  $(\# \text{messages} - \# \text{nodes visited}) / \# \text{messages}$
- maximum number of messages at a node
- *recall*, the number of objects found satisfying the desired search criteria. This metric is useful for keyword, inexact, and range queries.
- *message efficiency*, which is the recall per message used

**Guided versus Unguided Search.** In unguided or blind search, there is no history of earlier searches. In guided search, nodes store some history of past searches to aid future searches. Various mechanisms for caching hints are used. We focus on unguided searches in the context of unstructured overlays.

# Search in Unstructured Overlays: Flooding and Random Walk

- Flooding: with checking, with TTL or hop count, expanding ring strategy
- Random Walk:  $k$  random walkers, with checking
- Relationships of interest
  - ▶ The success rate as a function of the number of message hops, or TTL.
  - ▶ The number of messages as a function of the number of message hops, or TTL.
  - ▶ The above metrics as the replication ratio and the replication strategy changes.
  - ▶ The node coverage, recall, and message efficiency, as a function of the number of hops, or TTL; and of various replication ratios and replication strategies.
- Overall,  $k$ -random walk outperforms flooding

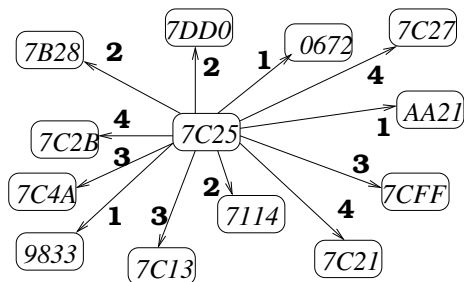
# Tapestry

- Nodes and objects are assigned IDs from common space via a distributed hashing.
- Hashed node ids are termed VIDs or  $v_{id}$ . Hashed object identifiers are termed GUIDs or  $O_G$ .
- ID space typically has  $m = 160$  bits, and is expressed in hexadecimal.
- If a node  $v$  exists such that  $v_{id} = O_G$  exists, then that  $v$  become the root. If such a  $v$  does not exist, then another unique node sharing the largest common prefix with  $O_G$  is chosen to be the *surrogate root*.
- The object  $O_G$  is stored at the root, or the root has a direct pointer to the object.
- To access object  $O$ , reach the root (real or surrogate) using prefix routing
- Prefix routing to select the next hop is done by increasing the prefix match of the next hop's VID with the destination  $O_{G_R}$ . Thus, a message destined for  $O_{G_R} = 62C35$  could be routed along nodes with VIDs  $6****$ , then  $62***$ , then  $62C**$ , then  $62C3*$ , and then to  $62C35$ .

# Tapestry - Routing Table



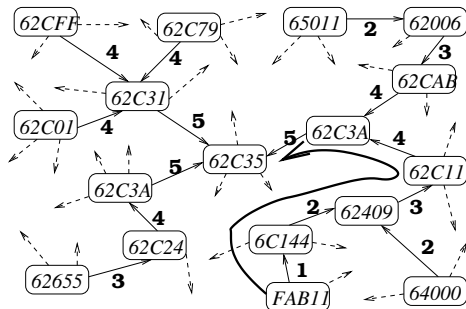
- Let  $M = 2^m$ . The routing table at node  $v_{id}$  contains  $b \cdot \log_b M$  entries, organized in  $\log_b M$  levels  $i = 1 \dots \log_b M$ . Each entry is of the form  $\langle w_{id}, IP\ address \rangle$ .
- Each entry denotes some "neighbour" node VIDs with a  $(i - 1)$ -digit prefix match with  $v_{id}$  – thus, the entry's  $w_{id}$  matches  $v_{id}$  in the  $(i - 1)$ -digit prefix. Further, in level  $i$ , for each digit  $j$  in the chosen base (e.g.,  $0, 1, \dots, E, F$  when  $b = 16$ ), there is an entry for which the  $i^{th}$  digit position is  $j$ .
- For each forward pointer, there is a backward pointer.



Some example links at node with identifier "7C25". Three links each of levels 1 through 4 are labeled.

# Tapestry: Routing

- The  $j^{\text{th}}$  entry in level  $i$  may not exist because no node meets the criterion. This is a *hole* in the routing table.
- Surrogate routing* can be used to route around holes. If the  $j^{\text{th}}$  entry in level  $i$  should be chosen but is missing, route to the next non-empty entry in level  $i$ , using wraparound if needed. All the levels from 1 to  $\log_b 2^m$  need to be considered in routing, thus requiring  $\log_b 2^m$  hops.



An example of routing from FAB11 to 62C35. The numbers on the arrows show the level of the routing table used. The dashed arrows show some unused links.

# Tapestry: Routing Algorithm

- Surrogate routing leads to a unique root.
- For each  $v_{id}$ , the routing algorithm identifies a unique spanning tree rooted at  $v_{id}$ .

(variables)

**array of array of integer**  $Table[1 \dots \log_b 2^m, 1 \dots b]$ ; // routing table

(1)  $NEXT\_HOP(i, O_G = d_1 \circ d_2 \dots \circ d_{\log_b M})$  executed at node  $v_{id}$  to route to  $O_G$ :

//  $i$  is  $(1 + \text{length of longest common prefix})$ , also level of the table

(1a) **while**  $Table[i, d_i] = \perp$  **do** //  $d_i$  is  $i$ th digit of destination

(1b)  $d_i \leftarrow (d_i + 1) \bmod b$ ;

(1c) **if**  $Table[i, d_i] = v$  **then** // node  $v$  also acts as next hop (special case)

(1d) **return**  $NEXT\_HOP(i + 1, O_G)$  // locally examine next digit of destination

(1e) **else return**  $(Table[i, d_i])$ . // node  $Table[i, d_i]$  is next hop

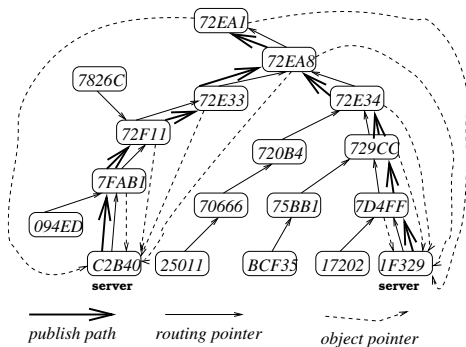
The logic for determining the next hop at a node with node identifier  $v$ ,  $1 \leq v \leq n$ , based on the  $i^{th}$  digit of  $O_G$ .

# Tapestry: Object Publication and Object Search

- The unique spanning tree used to route to  $v_{id}$  is used to publish and locate an object whose unique root identifier  $O_{GR}$  is  $v_{id}$ .
- A server  $S$  that stores object  $O$  having GUID  $O_G$  and root  $O_{GR}$  periodically publishes the object by routing a *publish* message from  $S$  towards  $O_{GR}$ .
- At each hop and including the root node  $O_{GR}$ , the *publish* message creates a pointer to the object
- This is the directory info and is maintained in *soft-state*.
- To search for an object  $O$  with GUID  $O_G$ , a client sends a query destined for the root  $O_{GR}$ .
  - ▶ Along the  $\log_b 2^m$  hops, if a node finds a pointer to the object residing on server  $S$ , the node redirects the query directly to  $S$ .
  - ▶ Otherwise, it forwards the query towards the root  $O_{GR}$  which is guaranteed to have the pointer for the location mapping.
- A query gets redirected directly to the object as soon as the query path overlaps the publish path towards the same root.



# Tapestry: Object Publication and Search



An example showing publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40. A query for the object from 094ED will find the object pointer at 7FAB1. A query from 7826C will find the object pointer at 72F11. A query from BCF35 will find the object pointer at 729CC.

# Tapestry: Node Insertions

- For any node  $Y$  on the path between a publisher of object  $O$  and the root  $G_{O_R}$ , node  $Y$  should have a pointer to  $O$ .
- Nodes which have a hole in their routing table should be notified if the insertion of node  $X$  can fill that hole.
- If  $X$  becomes the new root of existing objects, references to those objects should now lead to  $X$ .
- The routing table for node  $X$  must be constructed.
- The nodes near  $X$  should include  $X$  in their routing tables to perform more efficient routing.

Refer to book for details of the insertion algorithm that maintains the above properties.

# Tapestry: Node Deletions and Failures

## Node deletion

- Node  $A$  informs the nodes to which it has (routing) backpointers. It also provides them with replacement entries for each level from its routing table. This is to prevent holes in their routing tables. (The notified neighbours can periodically run the nearest neighbour algorithm to fine-tune their tables.)
- The servers to which  $A$  has object pointers are also notified. The notified servers send object republish messages.
- During the above steps, node  $A$  routes messages to objects rooted at itself to their new roots. On completion of the above steps, node  $A$  informs the nodes reachable via its backpointers and forward pointers that it is leaving, and then leaves.

Node failures: Repair the object location pointers, routing tables and mesh, using the redundancy in the Tapestry routing network. Refer to the book for the algorithms

# Tapestry: Complexity

- A search for an object expected to take  $(\log_b 2^m)$  hops. However, the routing tables are optimized to identify nearest neighbour hops (as per the space metric). Thus, the latency for each hop is expected to be small, compared to that for CAN and Chord protocols.
- The size of the routing table at each node is  $c \cdot b \cdot \log_b 2^m$ , where  $c$  is the constant that limits the size of the neighbour set that is maintained for fault-tolerance.

The larger the Tapestry network, the more efficient is the performance. Hence, better if different applications share the same overlay.

# Fairness in P2P systems

Selfish behavior, free-riding, leaching degrades P2P performance. Need incentives and punishments to control selfish behavior.

## Prisoners' Dilemma

Two suspects, A and B, are arrested by the police. There is not enough evidence for a conviction. The police separate the two prisoners, and separately, offer each the same deal: if the prisoner testifies against (betrays) the other prisoner and the other prisoner remains silent, the betrayer gets freed and the silent accomplice gets a 10 year sentence. If both testify against the other (betray), they each receive a 2 year sentence. If both remain silent, the police can only sentence both to a small 6-month term on a minor offense.

Rational selfish behavior: both betray the other - is not Pareto optimal (does not ensure max good for all). Both staying silent is Pareto-optimal but that is not rational. In the iterative version, memory of past moves can be used, in this case, Pareto-optimal solution is reachable.

# Tit-for-tat in BitTorrent

Tit-for-tat strategy: first step, you cooperate; in subsequent steps, reciprocate the action done by the other in the previous step.

- The BitTorrent P2P system has adopted the tit-for-tat strategy in deciding whether to allow a download of a file in solving the leaching problem.
- cooperation is analogous to allowing others to upload local files,
- betrayal is analogous to not allowing others to upload.
- *chocking* refers to the refusal to allow uploads.

As the interactions in a P2P system are long-lived, as opposed to a one-time decision to cooperate or not, *optimistic unchocking* is periodically done to unchoke peers that have been chocked. This optimistic action roughly corresponds to the re-initiation of the game with the previously chocked peer after some time epoch has elapsed.