Distributed Systems:
Distributed File Systems – Part 2

Dr Aidan Murphy

School of Computer Science and Informatics
University College Dublin
Ireland

Distributed Systems: Andrew File System

Introduction

- Developed at Carnegie Mellon University as part of an IBM-CMU collaboration.
- The key objective was to provide scalability and security to support file sharing between up to 5000 concurrent users.
- 2 key design features:
  - Whole-file serving: Entire files and directories are transmitted to client computers
  - Whole-file caching: Once a copy or chunk of a file has been transferred it is stored in a permanent local cache.

Typical Scenario

- When the client issues an open command for a shared file it checks the cache.
  - A copy of the file is downloaded to the cache if it is not present.

- The copy is then opened and the resulting UNIX file descriptor is returned.

- Subsequent read, write and other operations are applied to the local copy.

- While the client issues a close command, the local copy is checked.
  - Changes cause the file to be transmitted back to the server.
  - The client copy is maintained.

Why go for upload/download?

- Based on observations about typical workloads in academic and other environments:
  - Files are small (less than 10k in size)
  - Reads are about 6 times more common than writes.
  - Sequential access is common, random access is rare.
  - Most files are accessed by only one user
  - Most shared files are modified by one user
  - Recently used files will most probably be used again

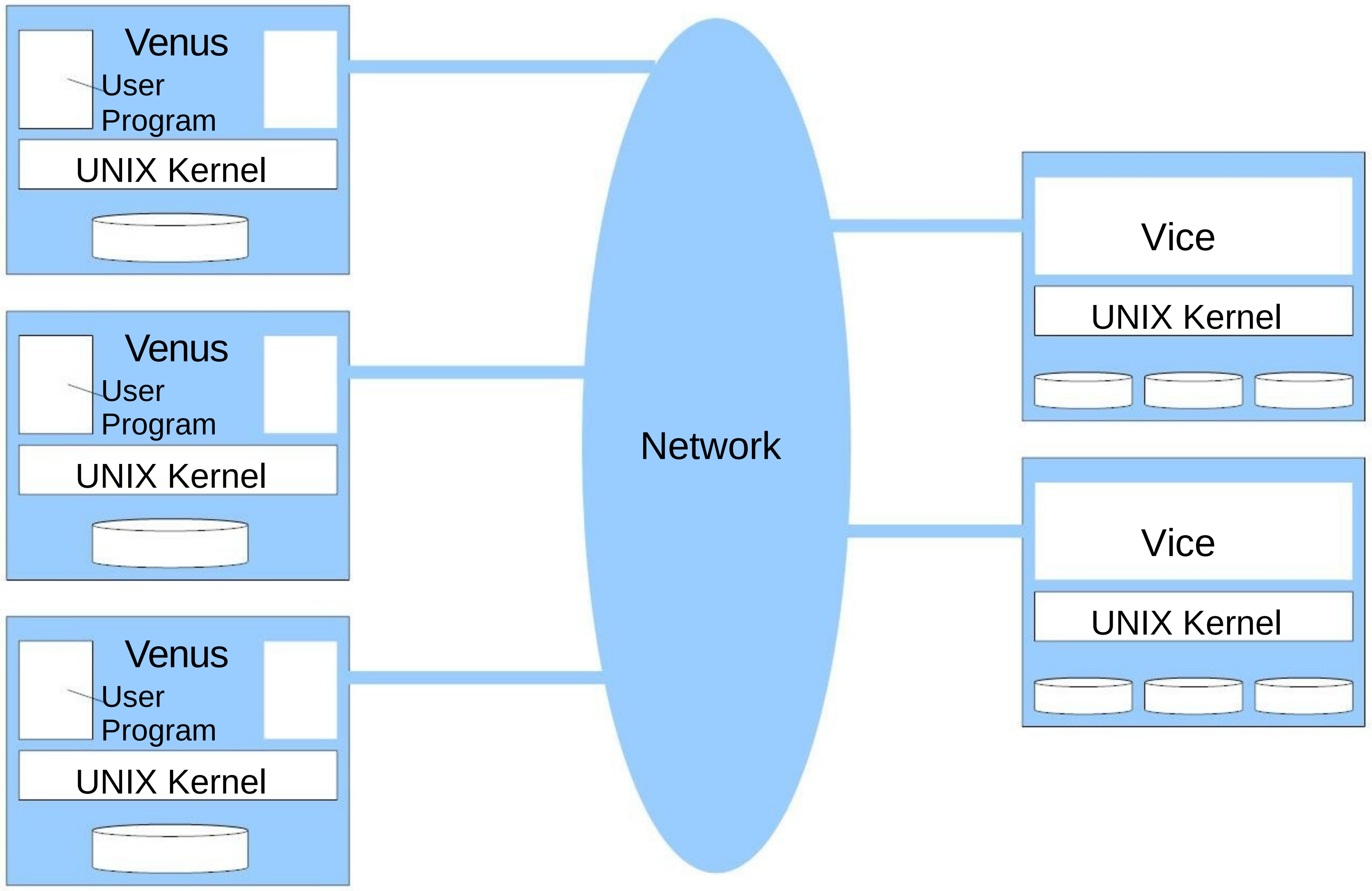
- Today, file sizes two magnitudes larger – but then so are network transmission speeds!

Design Issues

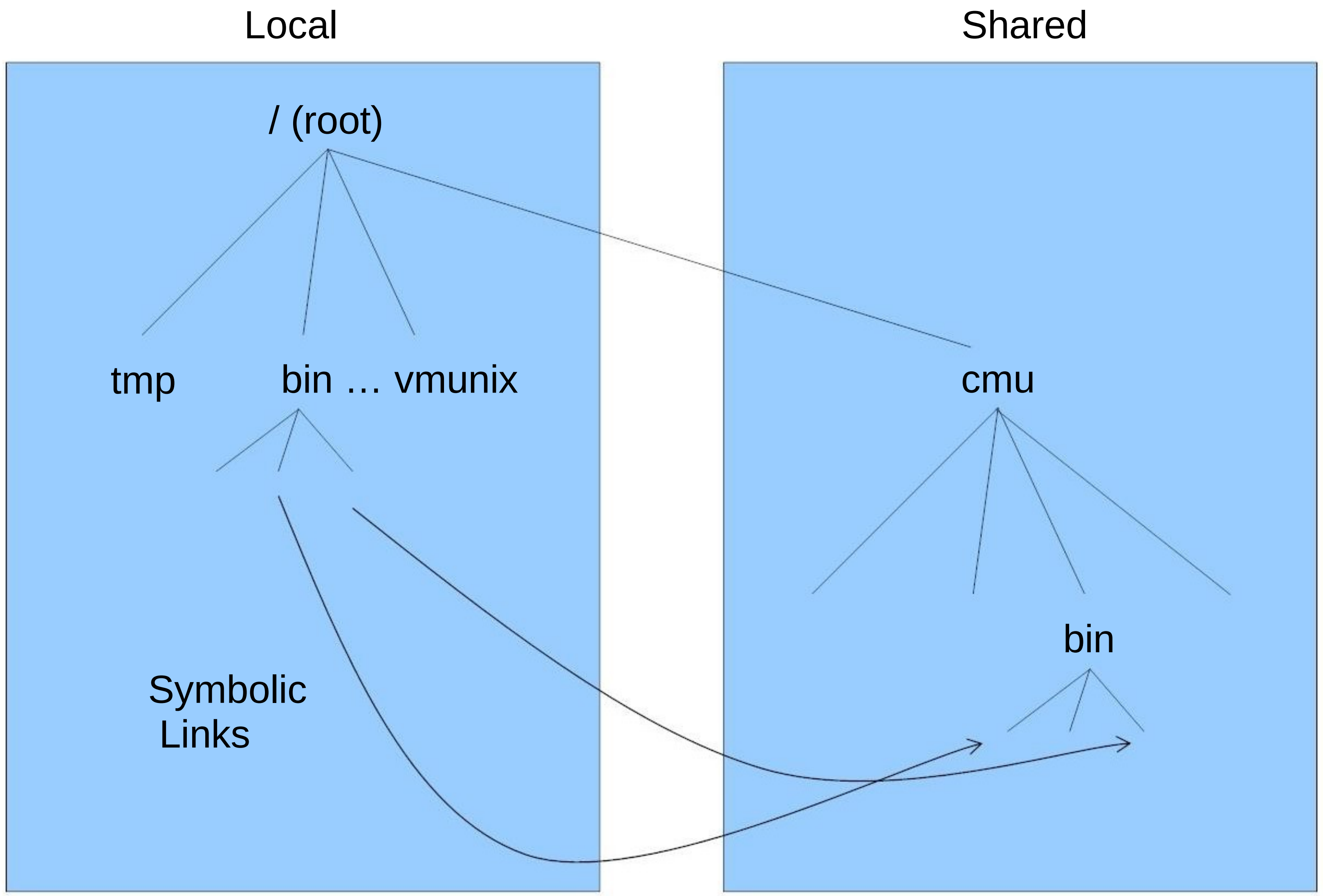
- Andrew employs a large local cache (say 100Mb):
  - It is used to store local copies of the files that the user most commonly accesses.
  - This includes both shared code libraries and that user's personal files.
  - This means that, after the initial access, most shared files can be accessed at the same speed as local files.

- Unfortunately, this does not work for files such as databases which are frequently accessed and updated by multiple users.

AFS Design


Venus
User Program
UNIX Kernel
Venus
User Program
UNIX Kernel
Venus
User Program
UNIX Kernel
Network
Vice
UNIX Kernel
Vice
UNIX Kernel

AFS File Space



AFS Implementation

- Vice:
  - Implements a Flat File Service.
  - Groups files into volumes (e.g. system binaries, documentation, a users home directory) for ease of location and movement.
  - Each file is identified by a unique 96-bit file identifier (fid).

Volume number	File Handle	Uniquifier

- Venus:
  - Updates the local cache, downloading files as necessary.
  - Manages shared directory structure.
  - Resolves directories issued by clients to fids using a step-by-step lookup.
  - Maintains cache coherence via callback promises

AFS Cache Coherence

- Vice supplies a token, known as a callback promise, for each copy of a file that is transmitted to a Venus process.
  - It represents a promise that Vice will contact that Venus process should any other client modify that file.
  - Callback promises are stored locally, initially with a valid state.
- When a file is updated, the Vice server performs a callback.
  - It contacts each client that was issued a token and informs it that the file was updated.
  - This causes the callback promise to be changed to cancelled.
- When Venus receives and open request, it checks the callback promise.
  - If it is set to cancelled, then a fresh copy of the file is retrieved.

AFS Cache Coherence

- When a workstation is restarted, Venus imposes an additional check on all files in the cache:

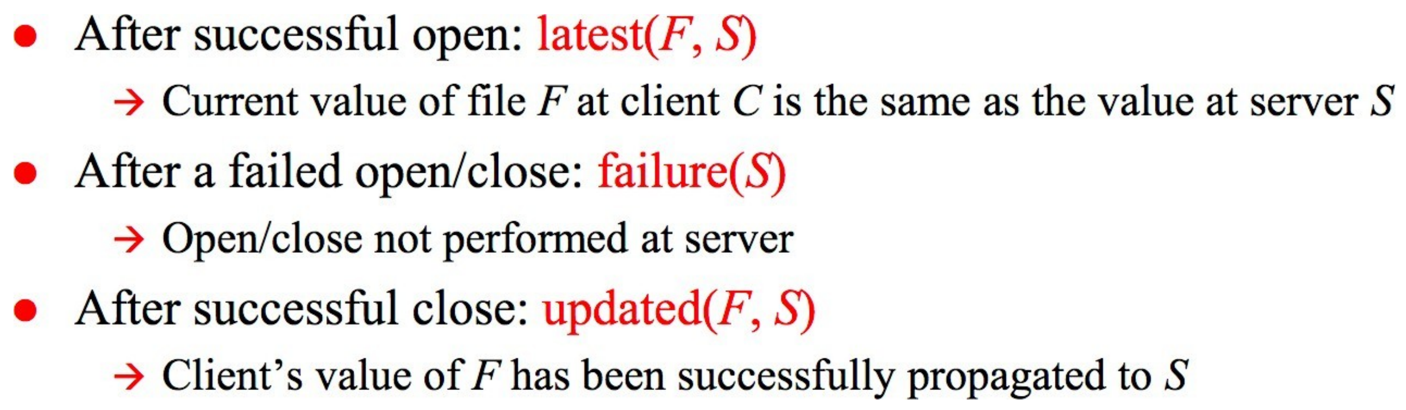
- Upon first access of a file, it submits a cache validation request to the server containing the files’ last modification timestamp.

- If the timestamp is correct, then the callback promise is validated.

- This validation also occurs if time T has elapsed since the communication with the server.

AFS Update Semantics

- Attempts to approximate to one-copy semantics for UNIX file access.
  - Full one-copy semantics requires that a write results in all cached copies of a file being updated before another operation is applied to that file.
- For a client C operating on a file F the following guarantees are maintained:



- After successful open: latest(*F*, *S*)
  - → Current value of file *F* at client *C* is the same as the value at server *S*
- After a failed open/close: failure(*S*)
  - → Open/close not performed at server
- After successful close: updated(*F*, *S*)
  - → Client's value of *F* has been successfully propagated to *S*

Other Aspects

- Threads
  - Venus and Vice are multi-threaded
- Read-only replicas
  - Multiple read-only copies of volumes containing frequently used (read) files (e.g. /bin) are maintained on different servers.
- Bulk transfers
  - The 64k limit on file transfer size reduces network overheads

AFS Review

	FFS	AFS
Objective	Reference Architecture	Scalable and Secure
Security	Directory Service/RPC	RPC-based
	Encryption	Private key algorithm
Naming	Location Transparency	Location Indepence
File Access	Remote Access Model + Cache	Upload / Download + Cache
Cache-Update Policy	Write-Through	Callback Promise (token)
	N/A	Write-on-close
Consistency	N/A	Timestamp-based
Service Type	Stateless	Stateful
Replication	N/A	Read-only

Peer to Peer Networks

What is P2P?

- Peer-to-Peer (P2P) Systems:
  - A set of networked devices that have **equal responsibility** and which **share resources and workload** as necessary.
  - Each device is both a **client** and a **server**.
  - There is no (or little) centralised control.

- Examples of Peer-to-Peer Systems:
  - File Sharing Applications e.g. Gnutella, BitTorrent, Kazaa, …
  - Instant Messengers and Telephony e.g. Skype, Yahoo, MSN
  - Data Processing Services e.g. Seti@Home

- Peer-to-Peer Systems **most effective** when used to store very large collections of immutable data

- Their design **diminishes their effectiveness** for applications that store and update mutable data

Characteristics of P2P Systems

- Aim of P2P Systems:
  - To deliver a service that is fully decentralised and self-organising, dynamically balancing the storage and processing loads between all the participating computers as computers join and leave the service.
- Key Characteristics:
  - Every user contributes resources to the service.
  - While resources differ, all nodes in a P2P system have the same functional capabilities and responsibilities.
  - Their correct operation does not depend on the existence of any centrally-administered systems.
  - They can be designed to offer a limited degree of anonymity to the providers and users of resources.
- Key Problem:
  - How to deal with the placement and accessing of shared resources in a manner that balances the workload and ensures availability, but does not add undue overheads?

Placement of Resources

- Deciding at which node(s) a resource should be located
  - Can be managed by the P2P application, or left to nature (the users)
- Examples:
  - Andrew File System
    - Consists of a set of file servers that decide how many copies of a file are needed to meet demand and where those copies should be located.
    - Users are not aware of the existence of multiple copies of the file (location independence).
  - Napster, Gnutella
    - Each user runs the application (peer) and decides which file(s) will be located at their node.
    - Copies are created in an ad-hoc way - each user downloads the files that they want (and then shares them).
    - Users are aware of the number of copies that they have access to.

Accessing (Locating) Resources

- In (centralised) managed environments: searching is not a problem:
  - We know where all the copies of a file are
  - We can develop algorithms that employ knowledge of the placement scheme that has been used.
- In P2P environments: resources are distributed over a diverse set of nodes.
  - How do we find the resource we are looking for?
  - We need to perform some kind of search!
- In such ad-hoc environments we have a problem:
  - We do not know where the resource is located
  - We may not even know what nodes exist!
  - While a number techniques have been developed to handle this, locating resources in a P2P system is still a hot topic of research…

Types of P2P Systems

- Three main types of P2P system:
  - Centralized systems: peer connects to server which coordinates and manages communication
    - e.g. SETI@home

centralised discovery
centralised communication

  - Decentralized systems: peers run independently with no central services.
    - Discovery is decentralized and communication takes place between the peers. e.g. Gnutella, Pastry

decentralised discovery
decentralised communication

  - Hybrid systems: peers connect to a server to discover other peers
    - peers manage the communication themselves (e.g. Napster).
    - This is also called 'Brokered P2P'.

centralised discovery
decentralised communication

Peer-to-Peer Middleware Systems

P2P Middleware Systems

- With the growing popularity of P2P systems, P2P researchers worked to develop generalized versions of the first P2P system
  - that utilise existing routing, naming, data replication and security techniques in new ways to provide a reliable resource sharing layer over an unreliable and untrusted collection of computers and networks
- Aims:
  - To be application independent
  - scalability: to share resources, storage and data present in computers at the edges of the internet on a global scale
- They provide reference architecture that allowed researchers to focus on specific P2P issues.
- These systems are commonly knows as P2P Middleware Systems.

P2P Middleware Systems

- P2P Middleware systems are based on 2nd generation architectures:
  - Every node in the system is a peer.
  - Some special infrastructure nodes included to improve network performance.
  - Peer nodes connect to infrastructure nodes (which may also be peers), and route searches via these infrastructure nodes.
- Perhaps the most central issue in P2P research is the issue of resource discovery (searching).
  - algorithms used to carry out the search within P2P middleware systems are known as routing overlay algorithms

Routing Overlays - basic idea

- P2P routing overlays are used to locate objects.
  - The middleware is a layer that is responsible for routing requests from any node to the node that holds the target resource.
  - However, P2P systems allow for the migration of resources between nodes.
  - As a result, the term overlay is used to clarify that it is an application layer routing mechanism (it is not a network layer mechanism like IP routing).
  - The first system to employ routing overlays was the Plaxton Mesh. Others to follow include Pastry and Tapestry.

- Routing overlays ensure that any node can access any resource by routing each request through a sequence of nodes.
  - It exploits knowledge at each of them in order to locate the resource.
  - Because P2P systems can store replicas, the routing algorithm must maintain the location of all copies.
  - It then delivers requests to the nearest “live” node

Routing Overlay - basic idea

- The main task of a routing overlay is:
  - Clients wishing to invoke an operation on a resource submit a request, which includes a GUID (unique identifier) for the resource, to the routing overlay algorithm.
    - Globally Unique IDentifiers (GUID)
  - It directs the request to a node on which a replica (or the original) of the resource resides.
- In addition the routing overlay is responsible for:
  - Creating GUID's for all new resources
  - Announcing the existence of the new resources to ensure that the resource is reachable by all clients.
  - Handling the removal of references to shared resources that are no longer in the system
  - managing the addition/removal of nodes to/from the system.

Routing Overlays - Resource GUIDS

- In a P2P System, resources are identified by a globally unique identifier (GUID) that is location independent.
  - For object-based resources the GUID is calculated using a hash function, such as SHA-1.
  - This hash is derived from some/all of the resources state. This means that clients receiving the resource can check the validity of the hash (self certifying aspect)
  - Uniqueness is verified by searching for another object with the same GUID.
- GUID's are used to determine the placement of objects, and to retrieve them
- Because of this, routing overlay routing systems are sometimes constructed using Distributed Hash Tables (DHT -this is reflected in the simple API or operations that is used to access and add resources)
- The DHT layer takes responsibility for:
  - choosing a location for each resource
  - storing it (with replicas to ensure availability)
  - providing access to it via a get() operation.

Routing Overlays - Distributed Hash Tables (DHT)

- DHTs form an infrastructure that can be used to build peer-to-peer networks.

- Question: GUID’s are used to determine the placement of objects, and to retrieve them - Within the DHT model, how does
  this happen.

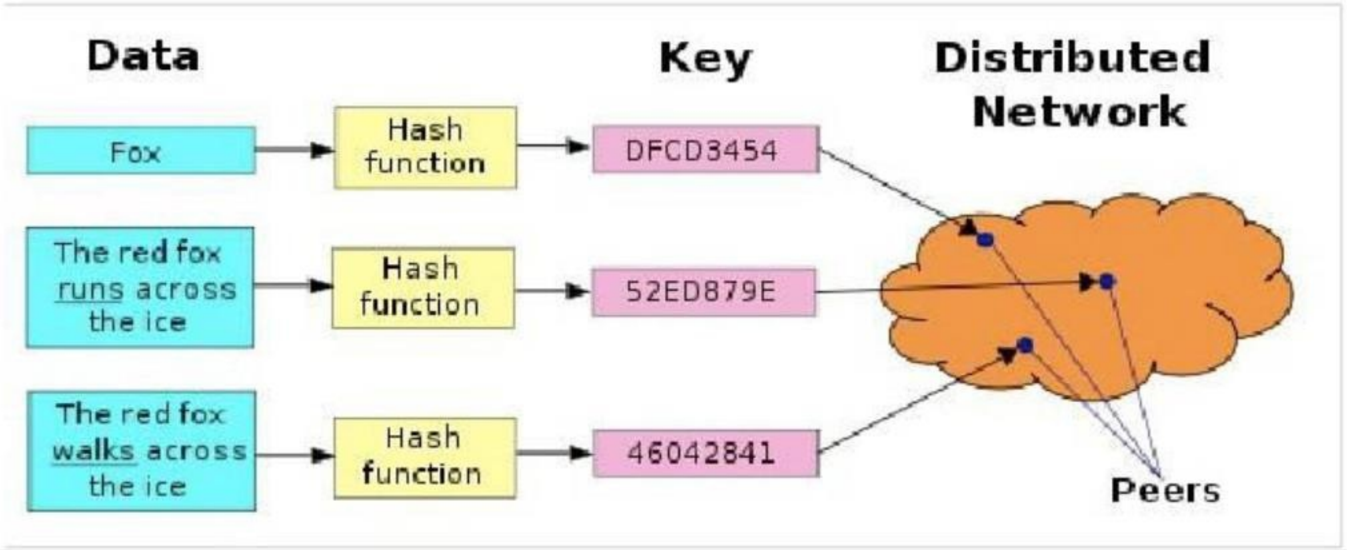
- Answer: A data item with GUID X is stored at the node whose GUID is numerically closest to X, and at the r hosts with GUIDs numerically closest to X.
    - where r is a replication factor chosen to ensure a very high probability of availability

Distributed Hash Table Operations

- put(GUID, data)
  - The data is stored in replicas at all nodes responsible for the object identified by GUID
- remove(GUID)
  - Deletes all references to GUID and the associated data
- data = get(GUID)
  - The data associated with GUID is retrieved from one of the nodes responsible for it.

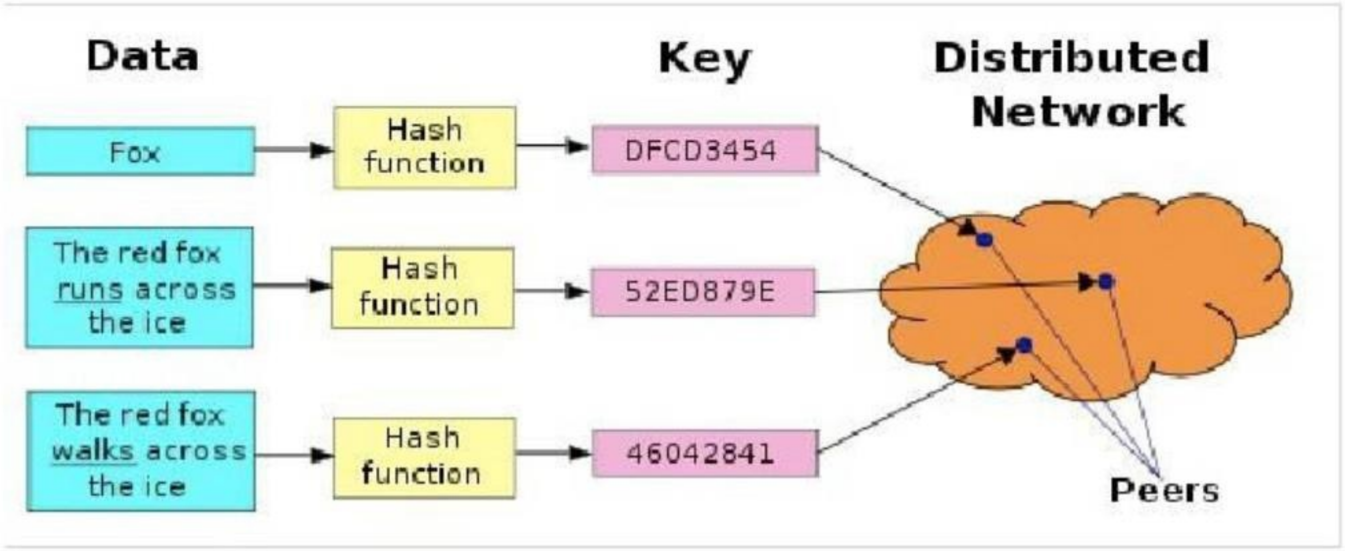
Distributed Hash Tables (DHT's)

- Networks that use DHTs include:
  - BitTorrent
- provide a lookup service similar to a hash table
  - (key, value) pairs are stored in the DHT
  - any participating node can efficiently retrieve the value (data) associated with a given key



Distributed Hash Tables (DHT's)

- Responsibility for maintaining the mapping from keys to values (data) is distributed among the nodes
  - can scale to extremely large numbers of nodes
  - handle continual node arrivals, departures, and failures.



DHT’s - Properties

- Decentralization: the nodes collectively form the system without any central coordination.
- Scalability: the system should function efficiently even with thousands or millions of nodes.
- Fault tolerance: the system should remain reliable even with nodes continuously joining, leaving, and failing
  - mainly through the replication of objects

Thank you