Storage services

Consider various computing environments and scenarios

Professional, academic, commercial, home – *based on traditional wired networks*

Mobile users with computing devices – may be *internet-connected and/or using wireless/ad.hoc*

Pervasive/active environments – sensor networks' logs/databases - *wired and wireless networks*

Some scenarios
(consider domain architecture, naming, location, authentication, authorisation, communication)

1. Single domain behind firewall – local files served by network-based file service
   plus accessing remote files and services.

2. Digital libraries, copyright, professional societies, publishers: scientific archive.
   issues: persistence of data through technology change; persistence of scientific archive
   who guarantees long term persistence?

3. Internet-based, cooperative P2P file-storage

4. GRID/cloud: storage for e-science applications

5. Commercial data centres

Examples of requirements for storage

Traditional environments

- program/document storage and development. Loading and running programs.
  Run-time data access and storage.

- application services (local and remote) need storage. Databases, CAD, versioning systems (SVN), email, newsgroups, naming directories, applications for download.

Mobile users

- detached operation: copy, disconnect, remote-work, reconnect, synchronise files – conflicts?
- access to files from remote locations – secure connection to home domain
- or use an internet-based service to place files close to where they will be used?

Peer-to-Peer (P2P)

- use spare capacity across the Internet for file storage, backup/archive
  e.g. Ocean Store from Berkeley, built on Planet Lab.
- Cooperative model e.g. music and film “sharing” e.g. Gnutella, Napster

Storage for e-science (grid -> cloud computing)

- e.g. petabytes of astronomic or genomic data and storage for computations on such data
- e.g. public data such as EHRs (security/trust is critical)

File structure, media types, indexing and retrieval,

Should a storage service provide support for structure representation, indexing and retrieval?

e.g. web-pages are composite documents, containing links to images.
Should general file services support structured files, as opposed to the byte-sequence abstraction?
Aim to avoid storing multiple copies of large image objects.
Issue: persistence of objects linked to – "40% of URLs fail after 2 years" (dangling references).

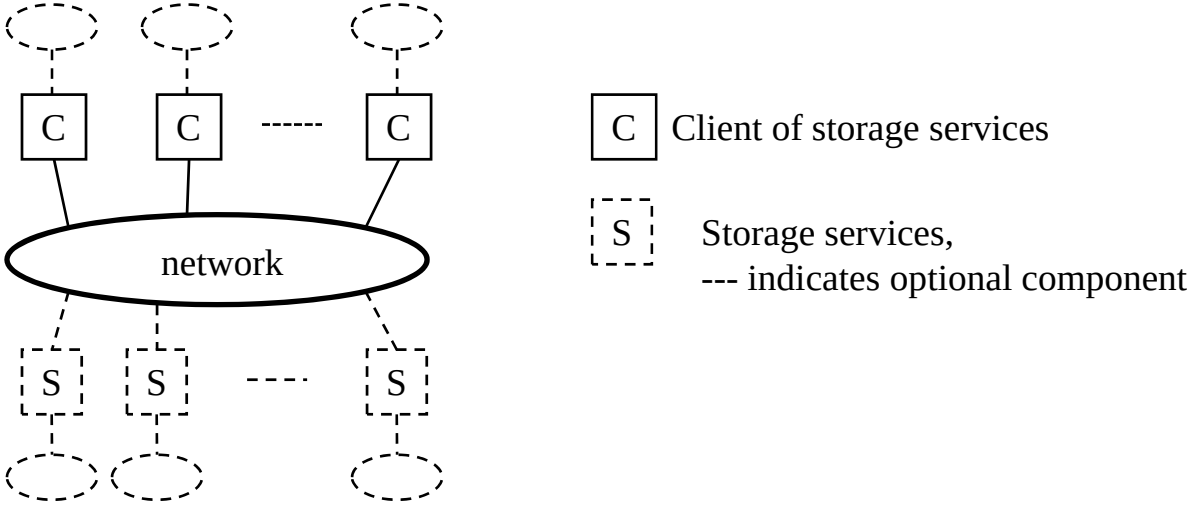
e.g. Cooperative work / versions e.g. SVN, records structure above the basic file abstraction
- aids synchronisation of updates and retrieval of any version.

e.g. Video/film stores, and content-delivery networks, deal in unstructured files but partition into blocks for transmission and reassembly.
e.g. Large collections e.g. photographs, e.g. videos of "my day" for memory-loss patients,
e.g. Audits of professional caring (NHS, SS) or business activities, process for suspicious or dangerous behaviour. Determine patterns that may imply fraud.
e.g. Logs of sensor data recording traffic, pollution, building projects (tunnels (Arup CTRL))
The data is analysed statistically to extract behaviour in context
– to determine which factors are significant

Should we use a database if we need to capture structure? e.g. for data mining?
We may only need to know external links

Storage services from first principles

First consider a professional, network-based environment in a single domain



1. (thin) clients have no local storage. System provides shared storage servers.
   e.g. early V system at Stanford, e.g. network computers

2. Clients have local storage. There are no dedicated storage services.
   would need e.g. Unix ***mount*** to achieve a shared filing system

3. Clients have local storage and there are also shared storage servers.
   Client discs used for?
   - Private desktop e.g. Xerox, Apple Mac, Windows
   - system files for bootstrapping
   - cached files – first-class copy is in shared service
   - temporary files not backed up by sys-admin

Storage service functionality

- open or closed? Is it bound into a single OS file system model e.g. pathname format

- how is functionality distributed?

  where is directory service for pathname resolution and access control?
  existence control / garbage collection? (reachability is via the directory graph)
  concurrency control? (based on knowledge (state) of what is open and mode)

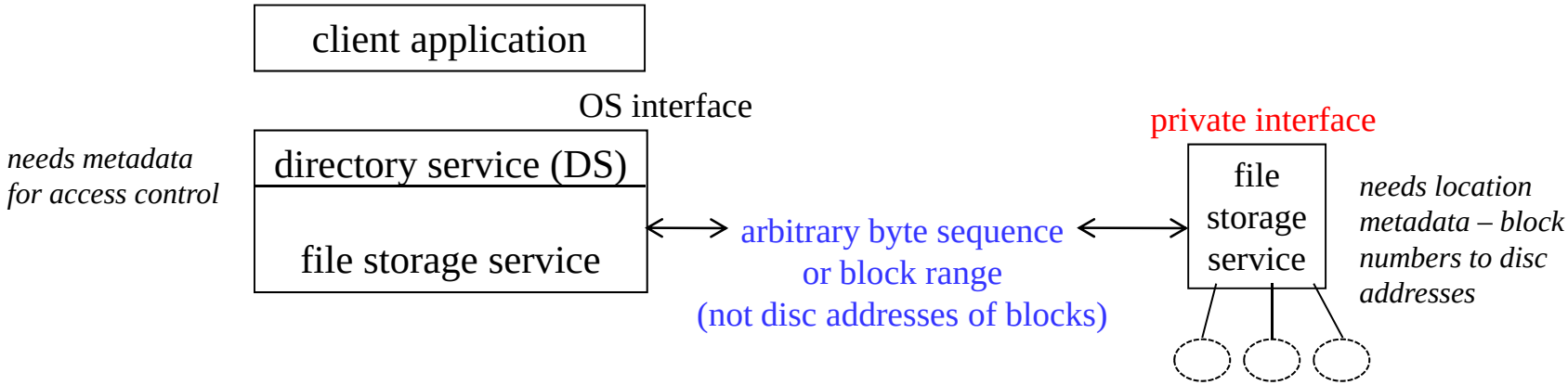
- level of interface?
  - remote block server
    e.g. early RVD (remote virtual disc)
    client system may do block allocation within an allocated partition (for minimal overhead at server) or server does allocation
    e.g. current video servers distribute blocks to achieve low latency
  - remote, UID-named files. Interactions may involve whole files or parts of files.
    server does block allocation – server overhead.
  - remote path-named files bound into a single OS naming scheme

- caching and replication

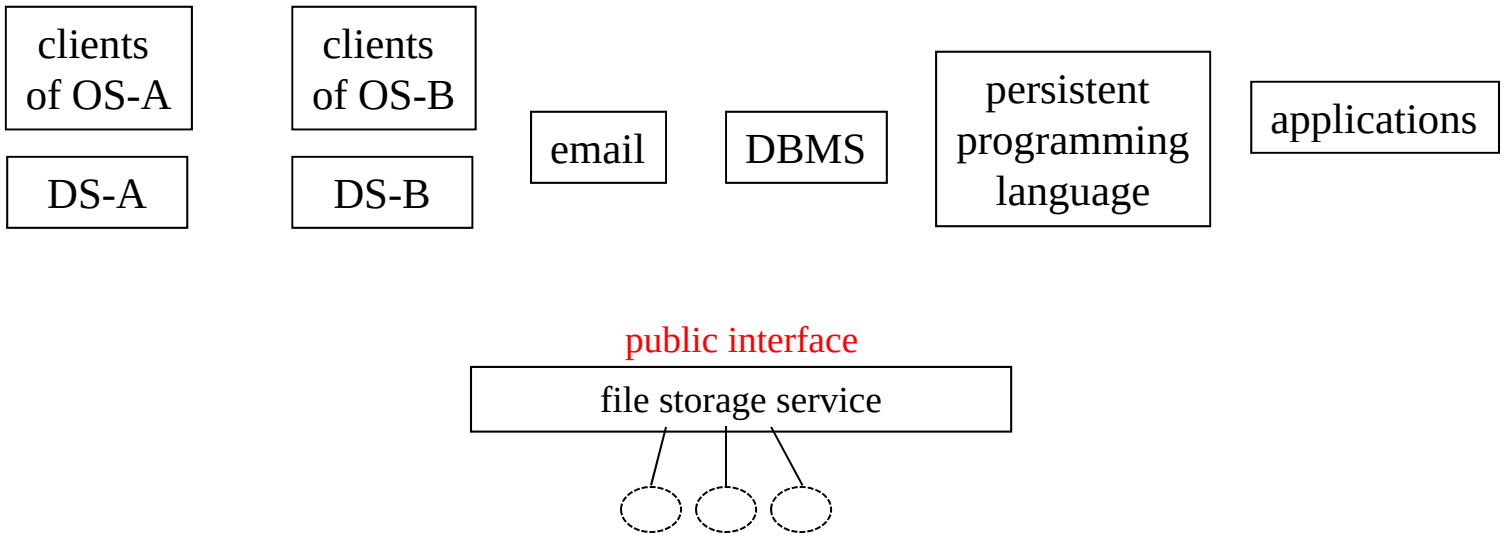
  is the service responsible for managing, or assisting with
  - multiple cached copies of a file at different clients?
  - replicas of a file (replicated by servers for reliability)?

Storage service architectures

a) Closed storage architecture (single OS accesses storage service (SS))

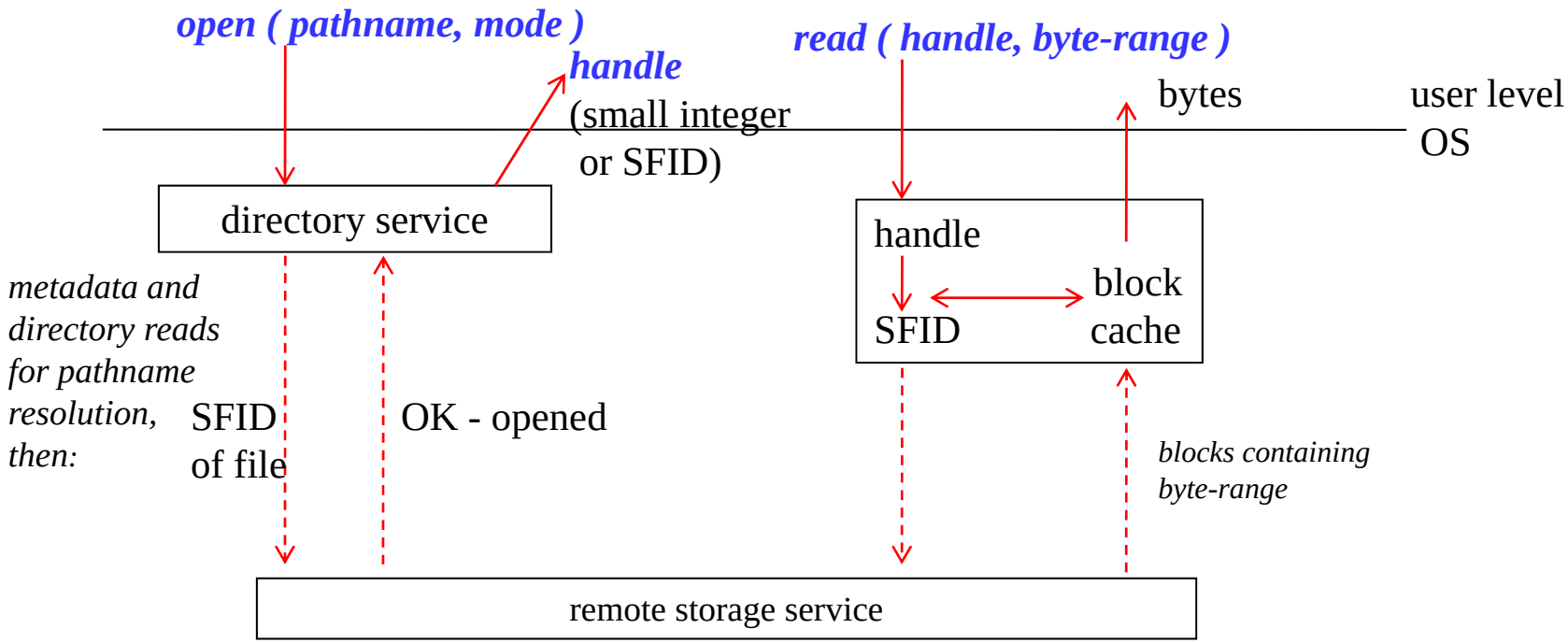


b) Open storage architecture



Remote SS interface at file storage (byte-sequence) level

SFID = system file identifier



*If the SS is stateless it does not support **open** at its interface*
and holds no info' on files in use.
Pathname resolution is still required, to obtain the SSID (hard link) of the file.

Operations in remote storage service interface

SFID ⟵ ***create***

read	***(SFID, byte-range)***	*assumes interaction at byte-sequence level*
write	***(SFID, byte-range)***	*as above, rather than whole file*
delete	***(SFID)*** ?	
lock	***(SFID)*** ?	
unlock	***(SFID)*** ?	? *are design decisions*
open	***(SFID)*** ?	
close	***(SFID)*** ?	

Does the service hold state?

- NO – specified as stateless
  - simple crash recovery
  - can't help with concurrency control
  - can't help with cache management

- YES – interface supports ***open/close***, records who has files open and access mode
  - server crash recovery – needs to interact with clients to rebuild its state
  - support for concurrency control by client OSs
    exclusive/shared locks better than single-writer/multiple-reader
  - support for cache management if clients store whole-file copies locally
    can *notify* holders of copies when a new version is written
    otherwise high traffic from clients requesting status of cached items

Existence control and garbage collection

A file should stay in existence for as long as it is reachable from the root of the directory naming graph

- Lost object problem ***SFID*** ⟵ ***create (...)***
  server allocates metadata in persistent store
  either: **server crash**
  or: reply (***SFID***) reaches client's main memory only
  client crash
  on server or client restart, client repeats ***create (...)*** and gets a new ***SFID***

- Storage service at file level can't help – doesn't see naming graphs

- A directory service can do existence control for its own objects.
  Multiple instances of the DS would have to cooperate to traverse the graph.
  This would work for a closed architecture and for a single OS's files within an open architecture.

- What about - objects shared across systems e.g. video clip in document?
  - objects not stored in directories?

- Consider a ***touch*** operation provided by the storage service. All its clients i.e. services, not users, must traverse their naming graphs and ***touch*** all their files periodically.
  Untouched files are deleted (archived).

An early case study: The Cambridge File Server (CFS)

Developed as part of the Cambridge Distributed Computing System (CDCS) in the late 1970s
CDCS was used as the Lab's research environment throughout the 1980s.

http://www.research.microsoft.com/NeedhamBook/cmds.pdf

Andrew Birrell and Roger Needham "A Universal File Server"
IEEE Trans SE 6 (5), pp 450-453, May 1980

CFS design features:

- open architecture, many OS clients e.g. Tripos, CAP, research file systems
- minimal support for structure without enforcing path-naming
- some operations with transactional semantics to avoid lost objects
- no delete operation
- garbage collection run from any processor bank machine

CFS basic concepts

CFS provides

two primitive types: ***byte*** and ***UID***

(***PUID*** = persistent ***UID, TUID*** = transient ***UID*** for open objects)

two abstractions: ***file*** – an uninterpreted sequence of bytes

named persistently by a ***PUID*** with a random component

index – a sequence of ***PUID***s, itself named by a ***PUID***

Index – used by CFS's clients to mirror their directory structures.

all index operations are transactional (failure-atomic – all-or-nothing)

- Existence control

  indexes form a general naming graph starting from a specific root index

  objects are preserved while they are reachable from the root

  - reference counts are used, recording the number of times a ***PUID*** is preserved in an index
  - an asynchronous garbage collector is used to detect cyclic structures

- Concurrency control – just MRSW

some CFS operations

file operations:

PUID ← ***create-file (index-PUID, entry, ...)*** % must store new ***PUID*** in existing index
% transaction avoids lost object problem
TUID ← ***open-file (PUID, read/write)*** % ***TUID*** = temporary ***UID*** for open file
data ← ***read (TUID, offset, amount)***
done ← ***write (TUID, offset, amount, data)***
done ← ***close-file (TUID)***
NOTE – no ***delete-file*** operation, garbage collection instead

index operations:

The index operations are used by OS clients to mirror the directory operations they offer their users.

PUID ← ***create-index (index-PUID, entry)*** % must store new ***PUID*** in existing index
% transaction avoids lost object problem
TUID ← ***open-index (PUID, read/write)*** % ***TUID*** = temporary ***UID*** for open index
done ← ***close-index (TUID)***
done ← ***preserve (index-PUID, entry, object-PUID)*** % put a link to an object in an index
PUID ← ***retrieve (index-PUID, entry)*** % extract a link from an index
done ← ***delete-entry (index-PUID, entry)*** % remove a link from an index
NOTE – no ***delete-index*** operation, garbage collection instead