Availability
Availability

- High availability: time for which system is not fully usable should be extremely low (e.g. 99.99% availability)

- Robustness: ability of system to function in spite of failures of components

- Failures are more likely in large distributed systems

- To be robust, a distributed system must
  - Detect failures
  - Reconfigure the system so computation may continue
  - Recovery/reintegration when a site or link is repaired

- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure
Reconfiguration

- **Reconfiguration:**
  - Abort all transactions that were active at a failed site
    - Making them wait could interfere with other transactions since they may hold locks on other sites
    - However, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data at a failed site (more on this later)
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new server
    - E.g. name server, concurrency coordinator, global deadlock detector
Since network partition may not be distinguishable from site failure, the following situations must be avoided:

- Two or more central servers elected in distinct partitions
- More than one partition updates a replicated data item

Updates must be able to continue even if some sites are down.

Solution: majority based approach

Alternative of “read one, write all available” is tantalizing but causes problems.
Majority-Based Approach

- The majority protocol for distributed concurrency control can be modified to work even if some sites are unavailable
  - Each replica of each item has a version number which is updated when the replica is updated, as outlined below
  - A lock request is sent to at least ½ the sites at which item replicas are stored and operation continues only when a lock is obtained on a majority of the sites
  - **Read** operations look at all replicas locked, and read the value from the replica with largest version number
    - May write this value and version number back to replicas with lower version numbers (no need to obtain locks on all replicas for this task)
Majority-Based Approach

- **Majority protocol (Cont.)**
  - **Write** operations
    - find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
  - **Failures** (network and site) cause no problems as long as
    - Sites at commit contain a majority of replicas of any updated data items
    - During reads a majority of replicas are available to find version numbers
    - Subject to above, 2 phase commit can be used to update replicas
  - Note: reads are guaranteed to see latest version of data item
  - Reintegration is trivial: nothing needs to be done
- Quorum consensus algorithm can be similarly extended
Read One, Write All (Available)

- Biased protocol is a special case of quorum consensus
  - Allows reads to read any one replica but updates require all replicas to be available at commit time (called **read one, write all**)
- Read one, write all available (ignoring failed sites) is attractive, but incorrect
  - If failed link may come back up, without a disconnected site ever being aware that it was disconnected
    - The site then has old values, and a read from that site would return an incorrect value
    - If site was aware of failure, reintegration could have been performed, but no way to guarantee this
- With network partitioning, sites in each partition may update same item concurrently
  - believing sites in other partitions have all failed
Site Reintegration

- When failed site recovers, it must catch up with all updates that it missed while it was down
  - **Problem**: updates may be happening to items whose replica is stored at the site while the site is recovering
  - **Solution 1**: halt all updates on system while reintegrating a site
    - **Unacceptable** disruption
  - **Solution 2**: lock all replicas of all data items at the site, update to latest version, then release locks
    - Other solutions with better concurrency also available
Comparison with Remote Backup

- Remote backup (hot spare) systems (Section 16.9) are also designed to provide high availability
- Remote backup systems are simpler and have lower overhead
  - All actions performed at a single site, and only log records shipped
  - No need for distributed concurrency control, or 2 phase commit
- Using distributed databases with replicas of data items can provide higher availability by having multiple (> 2) replicas and using the majority protocol
  - Also avoids failure detection and switchover time associated with remote backup systems
Coordinator Selection

- **Backup coordinator**
  - site which maintains enough information locally to assume the role of coordinator if the actual coordinator fails
  - executes the same algorithms and maintains the same internal state information as the actual coordinator (but does not take actions affecting other sites)
  - assumes the role of coordinator when its failure is detected
  - allows fast recovery from coordinator failure but involves overhead during normal processing.

- **Election algorithms**
  - used to elect a new coordinator in case of failures
  - Example: Bully Algorithm - applicable to systems where every site can send a message to every other site.
Bully Algorithm

- If site $S_i$ sends a request that is not answered by the coordinator within a time interval $T$, assume that the coordinator has failed. $S_i$ tries to elect itself as the new coordinator.

- $S_i$ sends an election message to every site with a higher identification number, $S_i$ then waits for any of these processes to answer within $T$.
  - If no response within $T$, assume that all sites with number greater than $i$ have failed, $S_i$ elects itself the new coordinator.
  - If answer is received, $S_i$ begins time interval $T'$, waiting to receive a message that a site with a higher identification number has been elected.
    - If no message is sent within $T'$, assume the site with a higher number has failed; $S_i$ restarts the algorithm.

- After a failed site recovers, it immediately begins execution of the same algorithm.
  - If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number.
Trading Consistency for Availability
What is Consistency?

- Consistency in Databases (ACID):
  - Database has a set of integrity constraints
  - A consistent database state is one where all integrity constraints are satisfied
  - Each transaction run individually on a consistent database state must leave the database in a consistent state

- Consistency in distributed systems with replication
  - **Strong consistency**: a schedule with read and write operations on a replicated object should give results and final state equivalent to some schedule on a single copy of the object, with order of operations from a single site preserved
  - **Weak consistency** (several forms)
Availability

- Traditionally, availability of centralized server
- For distributed systems, availability of system to process requests
  - For large system, at almost any point in time there’s a good chance that
    - a node is down or even
    - Network partitioning
- Distributed consensus algorithms will block during partitions to ensure consistency
  - Many applications require continued operation even during a network partition
    - Even at cost of consistency
Brewer’s CAP Theorem

- Three properties of a system
  - **Consistency** (all copies have same value)
  - **Availability** (system can run even if parts have failed)
    - Via replication
  - **Partition-tolerance** (network can break into two or more parts, each with active systems that can’t talk to other parts)

- Brewer’s CAP “Theorem”: You can have **at most two** of these three properties for any system

- Very large systems will partition at some point
  - Choose one of consistency or availability
  - Traditional database systems choose consistency
  - Most Web applications choose availability
    - Except for specific parts such as order processing
Replication with Weak Consistency

- Many systems support replication of data with weak degrees of consistency (i.e., without a guarantee of serializability)
  - i.e. $Q_R + Q_W \leq S$ or $2*Q_W < S$
  - Usually only when not enough sites are available to ensure quorum
    - But sometimes to allow fast local reads
  - Tradeoff of consistency versus availability or latency

- **Key issues:**
  - Reads may get old versions
  - Writes may occur in parallel, leading to inconsistent versions
    - Question: how to detect, and how to resolve
      - Version vector scheme, Section 25.5.4
Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent.

- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service.

- Known as **BASE** (Basically Available, Soft state, Eventually consistent), as opposed to ACID.
  - **Soft state**: copies of a data item may be inconsistent.
  - **Eventually Consistent** – copies become consistent at some later time if there are no more updates to that data item.
Availability vs Latency

- CAP theorem only matters when there is a partition
  - Even if partitions are rare, applications may trade off consistency for latency
    - E.g. PNUTS allows inconsistent reads to reduce latency
      - Critical for many applications
    - But update protocol (via master) ensures consistency over availability
- Thus there are two questions:
  - If there is partitioning, how does system tradeoff availability for consistency
  - else how does system trade off latency for consistency
Distributed Query Processing
For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.

In a distributed system, other issues must be taken into account:

- The cost of a data transmission over the network.
- The potential gain in performance from having several sites process parts of the query in parallel.
Query Transformation

- Translating algebraic queries on fragments.
  - It must be possible to construct relation $r$ from its fragments
  - Replace relation $r$ by the expression to construct relation $r$ from its fragments

- Consider the horizontal fragmentation of the account relation into
  
  $account_1 = \sigma\ branch\_name = \text{"Hillside" } (account)$
  
  $account_2 = \sigma\ branch\_name = \text{"Valleyview" } (account)$

- The query $\sigma\ branch\_name = \text{"Hillside" } (account)$ becomes
  
  $\sigma\ branch\_name = \text{"Hillside" } (account_1 \cup account_2)$
  
  which is optimized into

  $\sigma\ branch\_name = \text{"Hillside" } (account_1) \cup \sigma\ branch\_name = \text{"Hillside" } (account_2)$
\[ \sigma \text{branch\_name} = \text{“Hillside”} \ (account_1) \cup \sigma \text{branch\_name} = \text{“Hillside”} \ (account_2) \]

- Since \(account_1\) has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.

- Apply the definition of \(account_2\) to obtain
  \[ \sigma \text{branch\_name} = \text{“Hillside”} \ (\sigma \text{branch\_name} = \text{“Valleyview”} \ (account)) \]

- This expression is the empty set regardless of the contents of the \(account\) relation.

- Final strategy is for the Hillside site to return \(account_1\) as the result of the query.
Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented:
  \( \text{account} \bowtie \text{depositor} \bowtie \text{branch} \)
- \( \text{account} \) is stored at site \( S_1 \)
- \( \text{depositor} \) at \( S_2 \)
- \( \text{branch} \) at \( S_3 \)
- For a query issued at site \( S_I \), the system needs to produce the result at site \( S_I \)

\( \text{account} \bowtie \text{depositor} \bowtie \text{branch} \) ?
Possible Query Processing Strategies

- Ship copies of all three relations to site $S_1$ and choose a strategy for processing the entire query locally at site $S_1$.
- Ship a copy of the `account` relation to site $S_2$ and compute $temp_1 = \text{account} \bowtie \text{depositor}$ at $S_2$. Ship $temp_1$ from $S_2$ to $S_3$, and compute $temp_2 = temp_1 \bowtie \text{branch}$ at $S_3$. Ship the result $temp_2$ to $S_1$.
- Devise similar strategies, exchanging the roles $S_1$, $S_2$, $S_3$
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site
Semijoin Strategy

- Let $r_1$ be a relation with schema $R_1$ stored at site $S_1$
- Let $r_2$ be a relation with schema $R_2$ stored at site $S_2$
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at $S_1$. 
  1. Compute $temp_1 \leftarrow \prod_{R_1 \cap R_2} (r_1)$ at $S_1$.
  2. Ship $temp_1$ from $S_1$ to $S_2$.
  3. Compute $temp_2 \leftarrow r_2 \bowtie temp_1$ at $S_2$.
  4. Ship $temp_2$ from $S_2$ to $S_1$.
  5. Compute $r_1 \bowtie temp_2$ at $S_1$. This is the same as $r_1 \bowtie r_2$. 

\[ temp_1 = \prod_{R_1 \cap R_2} (r_1) \]
\[ temp_2 = r_2 \bowtie temp_1 \]
The semijoin of $r_1$ with $r_2$, is denoted by:

$$r_1 \bowtie r_2$$

it is defined by:

$$\Pi_{R_1} (r_1 \bowtie r_2)$$

Thus, $r_1 \bowtie r_2$ selects those tuples of $r_1$ that contributed to $r_1 \bowtie r_2$.

In step 3 above, $temp_2 = r_2 \bowtie r_1$.

For joins of several relations, the above strategy can be extended to a series of semijoin steps.
Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation $r_i$ is stored at site $S_i$. The result must be presented at site $S_1$.

- $r_1$ is shipped to $S_2$ and $r_1 \bowtie r_2$ is computed at $S_2$; simultaneously $r_3$ is shipped to $S_4$ and $r_3 \bowtie r_4$ is computed at $S_4$.

- $S_2$ ships tuples of $(r_1 \bowtie r_2)$ to $S_1$ as they produced; $S_4$ ships tuples of $(r_3 \bowtie r_4)$ to $S_1$.

- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at $S_1$, $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at $S_2$ and the computation of $(r_3 \bowtie r_4)$ at $S_4$.
Heterogeneous Distributed Databases

- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms.
- Data models may differ (hierarchical, relational, etc.).
- Transaction commit protocols may be incompatible.
- Concurrency control may be based on different techniques (locking, timestamping, etc.).
- System-level details almost certainly are totally incompatible.
- A multidatabase system is a software layer on top of existing database systems, which is designed to manipulate information in heterogeneous databases.
  - Creates an illusion of logical database integration without any physical database integration.
Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - Applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces
    - Technical difficulties and cost of conversion
    - Organizational/political difficulties
      - Organizations do not want to give up control on their data
      - Local databases wish to retain a great deal of autonomy
Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g. data types, precision,
  - Character sets
    - ASCII vs EBCDIC
    - Sort order variations
- Agreement on units of measure
- Variations in names
  - E.g. Köln vs Cologne, Mumbai vs Bombay
Query Processing

- Several issues in query processing in a heterogeneous database
  - Schema translation
    - Write a *wrapper* for each data source to translate data to a global schema
    - Wrappers must also translate updates on global schema to updates on local schema
  - Limited query capabilities
    - Some data sources allow only restricted forms of selections
      - E.g. web forms, flat file data sources
    - Queries have to be broken up and processed partly at the source and partly at a different site
  - Removal of duplicate information when sites have overlapping information
    - Decide which sites to execute query
  - Global query optimization
Mediator Systems

- **Mediator** systems are systems that integrate multiple heterogeneous data sources by providing an integrated global view, and providing query facilities on global view.
  - Unlike full fledged multidatabase systems, mediators generally do not bother about transaction processing.
  - But the terms mediator and multidatabase are sometimes used interchangeably.
  - The term *virtual database* is also used to refer to mediator/multidatabase systems.
Local transactions are executed by each local DBMS, outside of theMDBS system control.

Global transactions are executed under multidatabase control.

Local autonomy - local DBMSs cannot communicate directly to synchronize global transaction execution and the multidatabase has no control over local transaction execution.

- local concurrency control scheme needed to ensure that DBMS’s schedule is serializable
- in case of locking, DBMS must be able to guard against local deadlocks.
- need additional mechanisms to ensure global serializability
Local vs. Global Serializability

- The guarantee of local serializability is not sufficient to ensure global serializability.
  - As an illustration, consider two global transactions T1 and T2, each of which accesses and updates two data items, A and B, located at sites S1 and S2 respectively.
  - It is possible to have a situation where, at site S1, T2 follows T1, whereas, at S2, T1 follows T2, resulting in a nonserializable global schedule.

- If the local systems permit control of locking behavior and all systems follow two-phase locking
  - the multidatabase system can ensure that global transactions lock in a two-phase manner
  - the lock points of conflicting transactions would then define their global serialization order.
Cloud Databases
Data Storage on the Cloud

- Need to store and retrieve massive amounts of data
- Traditional parallel databases not designed to scale to 1000’s of nodes (and expensive)
- Initial needs did not include full database functionality
  - Store and retrieve data items by key value is minimum functionality
    - **Key-value stores**
  
- Several implementations
  - Bigtable from Google,
  - HBase, an open source clone of Bigtable
  - Dynamo, which is a key-value storage system from Amazon
  - Cassandra, from FaceBook
  - Sherpa/PNUTS from Yahoo!
Key Value Stores

- Key-value stores support
  - **put(key, value)**: used to store values with an associated key,
  - **get(key)**: which retrieves the stored value associated with the specified key.

- Some systems such as Bigtable additionally provide range queries on key values

- Multiple versions of data may be stored, by adding a timestamp to the key
Records in many big data applications need to have a flexible schema
- Not all records have same structure
- Some attributes may have complex substructure
- XML and JSON data representation formats widely used
- An example of a JSON object is:

```json
{
    "ID": "22222",
    "name": {
        "firstname": "Albert",
        "lastname": "Einstein"
    },
    "deptname": "Physics",
    "children": [
        { "firstname": "Hans", "lastname": "Einstein" },
        { "firstname": "Eduard", "lastname": "Einstein" }
    ]
}
```
Partitioning and Retrieving Data

- Key-value stores partition data into relatively small units (hundreds of megabytes).
- These partitions are often called tablets (a tablet is a fragment of a table).
- Partitioning of data into tablets is dynamic:
  - as data are inserted, if a tablet grows too big, it is broken into smaller parts
  - if the load (get/put operations) on a tablet is excessive, the tablet may be broken into smaller tablets, which can be distributed across two or more sites to share the load.
  - the number of tablets is much larger than the number of sites
    - similar to virtual partitioning in parallel databases
- Each get/put request must be routed to the correct site
- **Tablet controller** tracks the partitioning function and tablet-to-site mapping
  - map a get() request to one or more tablets,
  - Tablet mapping function to track which site responsible for which tablet
PNUTS Parallel Storage System Architecture

- Requests
- Master copy of partition table/tablet mapping
- Tablet controller
- Tablets
- Tablet servers
- Routers
Distributed Directory Systems
Directory Systems

- Typical kinds of directory information
  - Employee information such as name, id, email, phone, office addr, ..
  - Even personal information to be accessed from multiple places
    - e.g. Web browser bookmarks

- White pages
  - Entries organized by name or identifier
    - Meant for forward lookup to find more about an entry

- Yellow pages
  - Entries organized by properties
  - For reverse lookup to find entries matching specific requirements

- When directories are to be accessed across an organization
  - Alternative 1: Web interface. Not great for programs
  - Alternative 2: Specialized directory access protocols
    - Coupled with specialized user interfaces
Directory Access Protocols

- Most commonly used directory access protocol:
  - LDAP (Lightweight Directory Access Protocol)
  - Simplified from earlier X.500 protocol

- Question: Why not use database protocols like ODBC/JDBC?
- Answer:
  - Simplified protocols for a limited type of data access, evolved parallel to ODBC/JDBC
  - Provide a nice hierarchical naming mechanism similar to file system directories
    - Data can be partitioned amongst multiple servers for different parts of the hierarchy, yet give a single view to user
      - E.g. different servers for Bell Labs Murray Hill and Bell Labs Bangalore
  - Directories may use databases as storage mechanism
LDAP: Lightweight Directory Access Protocol

- LDAP Data Model
- Data Manipulation
- Distributed Directory Trees
LDAP Data Model

- LDAP directories store **entries**
  - Entries are similar to objects
- Each entry must have unique **distinguished name (DN)**
- DN made up of a sequence of **relative distinguished names (RDNs)**
- E.g. of a DN
  - `cn=Silberschatz, ou=Bell Labs, o=Lucent, c=USA`
- Standard RDNs (can be specified as part of schema)
  - `cn`: common name  `ou`: organizational unit
  - `o`: organization  `c`: country
- Similar to paths in a file system but written in reverse direction
Entries can have attributes
- Attributes are multi-valued by default
- LDAP has several built-in types
  - Binary, string, time types
  - Tel: telephone number   PostalAddress: postal address

LDAP allows definition of **object classes**
- Object classes specify attribute names and types
- Can use inheritance to define object classes
- Entry can be specified to be of one or more object classes
  - No need to have single most-specific type
LDAP Data Model (cont.)

- Entries organized into a **directory information tree** (DIT) according to their DNs
  - Leaf level usually represent specific objects
  - Internal node entries represent objects such as organizational units, organizations or countries
  - Children of a node inherit the DN of the parent, and add on RDNs
    - E.g. internal node with DN `c=USA`
      - Children nodes have DN starting with `c=USA` and further RDNs such as `o` or `ou`
    - DN of an entry can be generated by traversing path from root
  - Leaf level can be an alias pointing to another entry
    - Entries can thus have more than one DN
      - E.g. person in more than one organizational unit
LDAP Data Manipulation

- Unlike SQL, LDAP does not define DDL or DML
- Instead, it defines a network protocol for DDL and DML
  - Users use an API or vendor specific front ends
  - LDAP also defines a file format
    - LDAP Data Interchange Format (LDIF)
- Querying mechanism is very simple: only selection & projection
LDAP Queries

- LDAP query must specify
  - Base: a node in the DIT from where search is to start
  - A search condition
    - Boolean combination of conditions on attributes of entries
      - Equality, wild-cards and approximate equality supported
  - A scope
    - Just the base, the base and its children, or the entire subtree from the base
  - Attributes to be returned
  - Limits on number of results and on resource consumption
  - May also specify whether to automatically dereference aliases
- LDAP URLs are one way of specifying query
- LDAP API is another alternative
LDAP URLs

- First part of URL specifies server and DN of base
  - ldap://aura.research.bell-labs.com/o=Lucent,c=USA
- Optional further parts separated by ? symbol
  - ldap://aura.research.bell-labs.com/o=Lucent,c=USA??sub?cn=Korth
- Optional parts specify
  1. attributes to return (empty means all)
  2. Scope (sub indicates entire subtree)
  3. Search condition (cn=Korth)
C Code using LDAP API

```c
#include <stdio.h>
#include <ldap.h>
main() {
  LDAP *ld;
  LDAPMessage *res, *entry;
  char *dn, *attr, *attrList[] = {"telephoneNumber", NULL};
  BerElement *ptr;
  int vals, i;
  // Open a connection to server
  ld = ldap_open("aura.research.bell-labs.com", LDAP_PORT);
  ldap_simple_bind(ld, "avi", "avi-passwd");

  ... actual query (next slide) ...

  ldap_unbind(ld);
}
```
C Code using LDAP API (Cont.)

```c
ldap_search_s(ld, "o=Lucent, c=USA", LDAP_SCOPE_SUBTREE,
    "cn=Korth", attrList, /* attrsonly*/ 0, &res);
/*attrsonly = 1 => return only schema, not actual results*/
printf("found%d entries", ldap_count_entries(ld, res));
for (entry=ldap_first_entry(ld, res); entry != NULL; 
    entry=ldap_next_entry(ld, entry)) {
    dn = ldap_get_dn(ld, entry);
    printf("dn: %s", dn); /* dn: DN of matching entry */
    ldap_memfree(dn);
    for(attr = ldap_first_attribute(ld, entry, &ptr); attr != NULL;
        attr = ldap_next_attribute(ld, entry, ptr))
    {
        // for each attribute
        printf("%s:", attr); // print name of attribute
        vals = ldap_get_values(ld, entry, attr);
        for (i = 0; vals[i] != NULL; i++)
            printf("%s", vals[i]); // since attrs can be multivalued
    }
}
ldap_msgfree(res);
```
LDAP API (Cont.)

- LDAP API also has functions to create, update and delete entries.
- Each function call behaves as a separate transaction.
  - LDAP does not support atomicity of updates.
Distributed Directory Trees

- Organizational information may be split into multiple directory information trees
  - **Suffix** of a DIT gives RDN to be tagged onto to all entries to get an overall DN
    - E.g. two DITs, one with suffix \( o=Lucent, c=USA \) and another with suffix \( o=Lucent, c=India \)
  - Organizations often split up DITs based on geographical location or by organizational structure
  - Many LDAP implementations support replication (master-slave or multi-master replication) of DITs (not part of LDAP 3 standard)
- A node in a DIT may be a **referral** to a node in another DIT
  - E.g. \( ou=\textit{Bell Labs} \) may have a separate DIT, and DIT for \( o=\textit{Lucent} \) may have a leaf with \( ou=\textit{Bell Labs} \) containing a referral to the Bell Labs DIT
  - Referrals are the key to integrating a distributed collection of directories
  - When a server gets a query reaching a referral node, it may either
    - Forward query to referred DIT and return answer to client, or
    - Give referral back to client, which transparently sends query to referred DIT (without user intervention)
End of Chapter