Distributed Data Management

Wolf-Tilo Balke
Christoph Lofi
Institut für Informationssysteme
Technische Universität Braunschweig
http://www.ifis.cs.tu-bs.de
• Hermann Maurer
  – 17:00 in IZ160
  – Emeritus Uni Graz
  – Written more than 20 books and over 600 scientific article

– Topic: “You have seen nothing yet…”
  • Visions and outcomes from ever-accelerating innovation cycles
• **Additional** constraints and **cost factors** compared to “classic” query optimization
  – Network **costs**, network **model**, **shipping** policies
  – **Fragmentation & allocation** schemes
  – Different optimization goals
    • **Response time vs. resource consumption**

• **Basic techniques try to prune** unnecessary accesses
  – Generic query reductions
This lecture only covers very basic techniques

- In general, distributed query processing is a very complex problem
- Many and new optimization algorithms are researched
  - Adaptive and learning optimization
  - Eddies for dynamic join processing
  - Fully dynamic optimization
  - …

Recommended literature

4.0 Introduction

4.0 Classic Transaction Processing
4.1 Distributed Transaction Processing
   – Distributed Two-Phase Commit
4.2 Distributed Two-Phase Locking
4.3 Byzantine Agreements
4.4 Outlook: Web Age Transactions
Most early commercial databases have been used in banking and financial sector

- Financial Transaction:
  - “Agreement between a buyer and seller to exchange an asset for payment”
    - Not good: No payment, no asset, no agreement,…

- Database transaction
  - A group / workflow of coherent operations accessing and updating a database to perform a complex task
4.0 Transactions

• Automatic teller machines (ATM)
  – User Interaction
    • Insert your card and input PIN code
    • Select amount
    • Take card and cash
  – Basic business workflow
    • Authenticate user
    • Ask for requested amount
    • Query for available balance (read operation): if balance is too low shred card and abort…
    • Else deduce amount from balance (write operation)
    • Return card and dispense cash
• Travel agency
  – User interaction
    • “I want to go on vacations to Hawaii in the first week of May”
  – Basic business workflow
    • Check for flight availability during the week (read operation)
    • Check for hotel accommodation availability during the week (read operation)
    • Align dates for flights and hotels, shift it around a little for best prices
    • Reserve suitable room from hotel (write operation)
    • Buy flight ticket from airline (write operation)
While processing workflows severe problems can occur

- Even if we assume that individual workflows are always sensible and correct

Examples

- What if the ATM catches fire after withdrawing your money, but before dispensing it..?!
- What if you found the perfect flight and hotel, but while your flight is booked somebody else takes the last hotel room?
4.0 Transactions

• The previous examples require the concept of transactions
  – A transaction is a finite set of operations (workflow, program) that has to be performed in a certain order, while ensuring certain properties

• The properties are concerned with
  – **Integrity**: transactions can always be executed safely, especially in concurrent manner, while ensuring data integrity
  – **Fail Safety**: transactions are immune to system failures
4.0 Transactions

• What are **transactions** in databases?
  – A database stores a **data**
  – There are **consistency constraints** defined on the data
    - **Structural constraints**
      - Unique primary keys, correct foreign key relationships, correct data types, etc.
    - **Semantic constraints**
      - All additional rules ensuring a “correct” system state from an application point of view
  – If all constraints are fulfilled, the database is in an **consistent state**
4.0 Transactions

• A transaction is a database program (usually multiple queries) which **reads** and **modifies** data
  
  – A transaction should ensure **database consistency**
    • i.e. the transaction **transforms** the database from one consistent state to another consistent state
      – May be inconsistent during the execution of the transaction

– Furthermore, **transactional consistency** should be ensured
  • i.e. multiple transactions must be able to run without collisions
**4.0 Transactions**

- **Transaction Operations**
  - Transactions are an *interface contract* of an transaction-enabled server
    - **Start**: Starts an transaction, followed by a finite sequence of operations of a workflow or program
    - **Commit**: Executes all operations since transaction begin and ends the transaction
    - **Rollback**: Cancels the current transaction and reverts to the initial consistent state
• **Flat Transactions** are transaction which have a single start and commit point

  \[ S_1 \rightarrow \text{start} \rightarrow \text{operations} \rightarrow \text{operations} \rightarrow \text{commit} \rightarrow S_2 \]

  – A flat transaction **failing** returns to its start state

\[ S_1 \rightarrow \text{start} \rightarrow \text{operations} \rightarrow \text{operations} \rightarrow \text{rollback} \]
• Furthermore, **nested transactions** can be defined
  – Multiple commit and start points (**subtransactions**)
    • Simple case: transaction **chains**
    • Complex case: **workflows**

  – What happens in case of failure?
    • Ops2 fails: revert to $s_1$ or to $s_2$? Different options!
• The **history** of **transaction** management in DBMS can be classified into several “**ages**”

• **“Stone Age”**
  – Application had to care for fail safety themselves
  – No transactions

• **“Classic History”**
  – The age of the great DB pioneers
  – Rise of the relational model, introduction of SQL, development of transaction management
  – Focus on data integrity
  – **Flat transactions**
  – **System R** and **ACID**
  – **RDB2**!
• “Middle Ages”
  – Rise of complex business applications
  – Distributed databases
  – Relaxation of ACID principles
    • Tailoring for long-running transactions
    • Less strict transaction models
    • More flexible transaction model
      – Simple nested transactions: chains, sagas, etc.
  – Distributed Data Management
4.0 Transactions

• “Renaissance”
  – Workflow Management Systems
    • Workflows: complex nested transactions
    • Departure from flat transactions
  – Focus on workflow integrity and execution reliability

• “Modern Times”
  – Web Transactions
    • Especially, web service transactions
  – Long-running, loosely-coupled workflows on potentially very unreliable functions provided by autonomous parties
The dominant paradigm in classic transaction processing is the **ACID** paradigm

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

---

4.0 ACID

• **Atomicity**
  – Any transaction is either executed **completely, or not at all**
  – From outside view, the transaction has **no** observable **intermediate state**

• **Consistency Preservation**
  – Transactions lead from one **consistent state** of the data instance to another
    • Constraints are not violated after the transaction
• **Isolation**
  – Transactions are isolated from others, i.e. even in a concurrent scenario transactions do not interfere with each other.
  – **Parallel** execution of transactions has the same effect than **serial execution**

• **Durability**
  – Once **committed**, data changes performed by a transaction **survive** subsequent **system failures**
Possible problems:

- Atomicity
  - Dirty Read
- Consistency
  - Inconsistent read
- Isolation
  - Lost Update
  - Phantom Read
- Durability
  - Data loss due to system crash
• How can we deal with these problems?
  – **Transaction Protocols**!

• For understanding transactions protocols, we will need two important concepts
  – **Schedules**
    • A “plan” containing the execution order of the operations of different transactions
    • Also, schedule also denotes when locks are obtained or released
  – **Locks**
    • Flags which can be attached to data items to signal that it is already in use and may / may not be used by another operation
• For a set of concurrently executed transactions:
  – A schedule is a sequence of operations from different transactions
    • Usually, read or write operations
  – A schedule is called serial if operations of different transactions are not mixed, i.e. executed in serial order
    • Obviously, serial schedules are pose no problems wrt. to transactional consistency
    • Also, no parallelism possible
  – **Big aim**: Find schedules which behave like serial schedules but do allow for parallelism
4.0 Transactions

• Example (without starts and commits)
  – \( T_1 := r(x) \ r(y) \ w(u) \ w(x) \)
  – \( T_2 := r(p) \ r(q) \ w(p) \)
  – \( T_3 := r(z) \ w(z) \)

• Schedule
  – \( S := r(x) \ r(y) \ r(p) \ r(z) \ w(u) \ r(q) \ w(x) \ w(z) \ w(p) \)

• Serial schedule
  – \( S := r(p) \ r(q) \ w(p) \ r(x) \ r(y) \ w(u) \ w(x) \ r(z) \ w(z) \)
4.0 Transactions

• How can we find schedules which “behave” safely?
  – i.e. equivalent to a serial plan?

• **Pessimistic Protocols**
  – Assume that error conditions will occur and prevent any problems beforehand
  – Spend some effort to create “safe” schedules
    • “Standard” approach for databases
    • e.g. **two phase locking**

• **Optimistic Protocols**
  – Assume everything will usually be fine and fix damage if something goes wrong
  – Just schedule something and see what happens
• We will focus on **locking protocols**
  
  – **Pessimistic approach** using locks to avoid transactional inconsistencies
  
  – Simplified: If a transaction needs some data, it **obtains** a lock on it
    
    • Any other transaction may not use the item
    • Other transaction must wait until lock is released
  
  – If the item is not used anymore, the lock is **released**
    
    • Other transaction may continue using the item
• Two types of locks
  – **Read locks:**
    • Read locks can be **shared**
    • Multiple read locks on the same item may be issued to different transactions
      – Parallel reading!
  – **Write locks**
    • Write locks may not be shared
      – Only one simultaneous write!
    • A write lock **cannot** be obtained if the item is already read-locked by any other transaction
    • If the same transaction already holds a read lock, the lock can be promoted to a write lock
• Most commercial database systems rely on two-phase locking
  – Two-phase locking means that for each transaction all necessary locks are acquired before the first lock is released
• When operation **accesses** data item within transaction
  – If item *isn't locked*, then server **locks** and proceeds
  – If item is held in a conflicting lock by another transaction, transaction must **wait** till lock released
  – If item is held by non-conflicting lock, lock is **shared** and operation proceeds
  – If item is already locked by same transaction, lock is **promoted** if possible

• When transaction **commits** or **aborts**, locks are **released**
Two-phase locking protocols are a simple way to generate only serializable schedules

\[ S := \text{lock}(x) \; \text{r}(x) \; \text{lock}(y) \; \text{r}(y) \; \text{lock}(p) \; \text{r}(p) \; \text{w}(p) \; \text{unlock}(p) \; \text{w}(x) \; \text{unlock}(x) \; \text{unlock}(y) \]

- Transactions blue and green interleaved
- Still deadlocks, must be prevented!
  - RDB2!

Dining Philosophers Problem!

- Edward Dijkstra, 1965
- [http://ccl.northwestern.edu/netlogo/models/run.cgi?DiningPhilosophers.790.571](http://ccl.northwestern.edu/netlogo/models/run.cgi?DiningPhilosophers.790.571)
• Philosophers sitting around a round table
  – Each philosopher has a bowl of rice (or spaghetti) and one chopstick (or fork)
    • But you need two chopsticks (forks!!) to eat
  – Idea: Just grab two chopsticks and start
    • The others just wait until you are done
  – But what happens if everybody simultaneously grabs one chopstick?
    • Deadlock!
• **Conservative locking** (also called static locking or preclaiming) acquires all necessary locks before its first read or write
  
  – Restricts concurrency
  – Improves chances of successful commit
  
  – Only possible in restricted scenario, since read/write sets must be declared to the scheduler **in advance**
• **Strict two-phase** locking holds all exclusive locks until the respective transaction terminates

  – Based on the notion that a running transaction may always need further locks
  – Output are only strict schedules that are also interesting for recovery
4.0 Transactions

• Summary “classic” transaction management
  – Flat transactions
  – Most commonly, locking protocols are used
  – Usually, full ACID properties are delivered
    • Only smaller transactions supported
    • Transactions have to be executed fast
      – Too many locks!
    • Limited degree of flexibility
4.1 Distributed Transactions

• Base idea for distributed transaction management: Just **generalize** known algorithms for distributed environments

• **Problems:**
  – Transaction may run **longer** and span **multiple nodes**
    • **Network** communication is slow
    • Should operations performed on one node lock resources on other nodes?
    • When somebody really needs a DDBMS, he usually has more complex queries and transactions
      – More powerful transaction models needed?
4.1 Distributed Transactions

– More potential failure sources
  • Node failures
  • Connection failures
  • Message corruption

– No global system time
  • Most time-stamp-based protocols won’t work

– Agreement problems
  • If multiple nodes participate in one transaction, how can all nodes agree on a commit?

– Replication may have been used
  • Is it safe to assume that all replicated fragments contain the same data?
• Problem: replication consistency
  – What happens, if a fragment is replicated multiple times?
  – Mutually consistent data states
    • All copies of a given data item have identical values
    • Also called one-copy equivalence
  – In some cases it may be beneficial to sacrifice one-copy equivalence and allow the replicas to diverge
    • Eventually, all replicas are synchronized
    • So called eventually consistent approaches
4.1 Distributed Transactions

• In any case: transaction operations have to be **distributed** over different nodes
  – **Data** and **resources** are distributed!

• **Example**: simple flat transaction
    • **Op1** and **Op2** are executed at node 1
    • **Op3** is executed at node 2
    • **Op4** is executed at node 3
• Basic idea
  – Use a **central** transaction controller handling everything
    • Granting and releasing **locks**
    • Generation of schedules
    • **Abort**ing and commit**ing** transactions
4.1 Distributed Transactions

• Obviously, the central controller needs full access to all relevant node system resources
  – This is usually only the case in homogenous distributed databases
    • What happens in inhomogeneous, e.g. federated databases?

• Base idea: Delegate responsibility
  – Local DDBMS should be responsible for execution of transactions
  – Mediator layer of the DDBMS supervises local execution (middleware transaction manager)
4.1 Distributed Transactions

• Use **nested transactions** for distributing transactions to nodes!
  
  – Split the transaction hierarchically into multiple **smaller transaction** spanning just one node each
    
    • **Transaction trees**!
  
  – Each node handles its own transaction **locally**
    
    • Additional operations: Vote-Commit and Ready-to-Commit / Ready-to-Abort
  
  – Transaction manager just **moderates** and alone decides on **final commit or rollback**
4.1 Distributed Transactions

- **Original** transaction

- **Final state** broken down on sub-transactions
  - T1: Start – Op1 – Op2 – Commit
  - T2: Start – Op3 – Commit
  - T3: Start – Op4 – Commit

- **Transaction tree**
  - Sub-Transactions could even be further split into sub-sub transactions
4.1 Distributed Transactions

• Executing the transaction tree
  – Execution is initiated from the root transaction
  – Child transactions are recursively started
    • If child transactions are independent, they can be executed in parallel
      – Good for performance!
    • Dependent transactions must be executed sequentially
    • As soon as one child transaction fails, all others child transactions also have to be aborted or rolled-back
      – Failures propagate to the root
      – A single failure forces the whole tree to abort!
4.1 Distributed Transactions

• Assume there are no communication or node failures
  – Declare a **single** node as ‘commit coordinator’ (CC)
    • Only the CC will decide about **global** commit/abort
    • The CC initiates a **voting phase** among all nodes
  – Every participating node decides **locally** about safe commit or necessary abortion of its local transaction
    • If asked, it will send either Ready-to-Commit, or Ready-to-Abort
    • Once a decision has been sent it may not be reversed
    • In the state **Ready-to-Commit** recovery and commit both have to be possible (Redo/Undo log files!)
4.1 Distributed Transactions

• Commit coordinator

send out Begin-Vote

receive at least one Vote-Abort

Decision: Abort

send to all Abort

Aborted

receive all Vote-Commit

Decision: Commit

send to all Commit

Committed
4.1 Distributed Transactions

- Participating transactions

- Received **Begin-Vote**
  - **send** Vote-Commit
  - **send** Vote-Abort

- **Aborted**
  - **receive** Abort

- **Ready-to-Commit**
  - **receive** Commit

- **Committed**
4.1 Distributed Transactions

• Now also consider network and node failures
  – What if a node does not respond to the **Begin-Vote**?
  – What if a node does not receive further **information** from the coordinator?

• Two new phases and new messages
  – **Voting Phase** followed by **Decision Phase**
  – **Time-Out** and **Help-Me**
4.1 Distributed Transactions

• These considerations result in the Two-Phase-Commit Protocol

• The coordinator starts the voting phase and collects votes...
  – If at least one vote did not arrive after a predefined time interval the coordinator declares a time-out and decides for global abort
• If any **participant** in Ready-to-Commit state does not hear from the coordinator…

  – It declares a **time-out** and sends out **Help-Me** messages to other participating nodes
    • If some other node has **committed**, it must have come from the coordinator thus it is safe to commit
    • If some other node has **aborted**, it is safe to abort
    • If some other node has **not yet voted**, it may (after knowing there is at least one time-out) immediately **Vote-Abort** and thus kill the global transaction
    • If all other nodes are also ready to commit, Help-Me does not help
4.1 Distributed Transactions

- Commit coordinator

send out Begin-Vote

Wait

receive all Vote-Commit

Decision: Commit

send to all Commit

Aborted

send to all Abort

Decision: Abort

receive at least one Vote-Abort
Or time-out

Committed
4.1 Distributed Transactions

- Participating nodes

![Diagram showing the flow of transactions in a distributed system. The diagram includes states such as Start decision, Ready-to-Commit, Aborted, Blocked, Recover, and Commited. Messages include Begin-Vote, Vote-Commit, Vote-Abort, and Help-Me.](image)
In the previous slides, we assumed that sub-transactions are handled locally by the nodes. Works fine as long as the sub-transactions are independent. If not, no parallelism easily possible. Same problem as with transaction schedules in central DBMS! Same solutions possible?

Idea: Generalize two phase locking (> D2PL) for a distributed setting!
• **Two Phase Locking (2PL) in a distributed environment**

  – **Remember 2PL:**
    • First obtain all required locks, then release all locks
  – **Several types of parties are involved in a distributed 2PL locking scheme**
    • **Central Lock Manager (LM)**
      – Manages which data is locked by which transaction
    • **Coordination Manager (CM)**
      – Manages the transaction, e.g. obtains locks from LMs and distributes operation to DPs
    • **Data Processors (DP)**
      – Execute a single operation assigned by CMs
• **Types of lock managers used**
  
  – **Centralized 2PL**
    
    • Use a single **central lock manager** for managing all necessary locks
  
  – **Primary Copy 2PL**
    
    • **Multiple lock managers**, each responsible for a certain data partition
  
  – **Distributed 2PL**
    
    • **Every node** may potentially be a **lock manager**
• Careful with replication
  – If data is replicated, this must be known by the **lock managers** and **transaction managers**!
    • Replication Protocol needed!
  – **Simple Version:**
    • If a **lock** on a replicated data item is needed, all copies need to be **locked**
    • If an **update** is performed on a replicated item, the TM needs to issue updates to all copies!
• **Centralized 2-PL**
4.2 D-2PL

- In centralized 2-PL, the **lock manager** is the bottleneck
  - **Scalability issues** with just one lock manager
  - **Central point of failure**
    - No lock manager $\Rightarrow$ No transactions

- **Primary Copy 2-PL** helps by introducing multiple **lock managers**
  - Each lock manager is responsible for defined partitions of the data

- Finally, fully **distributed 2PL** expects a lock manager at each site
  - Especially suited for dealing with **heavy replication**
  - Each lock manager “knows” its own data and reaches agreements with other lock managers
    - Lock managers coordinate replication
• **Enterprise applications** usually involve multiple data sources
  – Transaction may also span **multiple** heterogeneous data sources
    • e.g. book a flight within one system and an hotel in another
  – Need for **federated transaction management**
    • Additional coordination layer necessary, i.e. **transaction manager**
      – Usually provided by an **application server**
    • All participating databases need a common interface for coordinating transactions
      – e.g. XOpen **XA**
4.2 Applications

Applications Clients
ATM bookkeeper...
... travel agency
... bookkeeper

Application Layer

Application Management Layer

DBMS Layer

Applications Clients

ATM

... travel agency

... bookkeeper

app_1

app_n

Transaction Manager

Application Server

DB pages

exposed data

Encapsulated data

Transaction Manager

DBMS

DB pages

Transaction Manager

Encapsulated data

exposed data
4.2 Applications

- **Example:** JTA
- **Java Transaction API**
- Uses **Application Server**
  - e.g. J2EE Server
  - Provides centralized **Transaction Manager**
    - Provided by AppServer
  - **User Transaction** interface for applications
  - XOpen XA Adapter connecting to databases
4.2 Applications

- J2EE **Application Servers** with JTA Transaction Manager Implementations
  - JBoss
  - Apache Geronimo
  - Sun Glassfish
  - Bea WebLogic Server
  - IBM WASCE
  - Oracle Application Server
  - SAP NetWeaver
  - …
4.2 Applications

• Open Group XOpen XA
  – Vendor-spanning standard protocol for Distributed Transaction Processing
  – Each DBMS / data source participating within a transaction needs to support XA
  – Uses Distributed 2-Phase Locking
  – Each DBMS is responsible for maintaining integrity of its own data
    • Centralized transaction manager necessary to coordinate individual commits
4.2 Applications

• Example: JTA-Transactions

```java
UserTransaction ut = envCtx.lookup("jta/UserTransaction");
DataSource ds = envCtx.lookup("jdbc/Datasource");
// note: explicit transaction handling necessary!
ut.begin();
boolean success = false;
try {
    Connection conn = ds.getConnection();
    // do stuff here
}
finally {
    if (success)
        ut.commit();
    else
        ut.rollback();
}
```
• Remember earlier: **transaction trees** for transactions in inhomogeneous systems
  
  – Core idea was to distribute **sub-transactions** across nodes
    
    • During voting face, each node may **vote** to either **commit** or **abort**
    
    • A single abort-vote will abort the global transaction
  
  – This idea works fine in distributed databases because we can **trust** all participating nodes and information channels
    
    • e.g. if a node **signals a failure**, something really went wrong, if the **controller decides to rollback** the whole transaction, he is correct to do so
• But what happens *autonomy* of nodes increase?
  – Think loosely coupled federated database!
    • Or even worse: P2P databases!

• Nodes or communication may start to misbehave!
  – **Malicious Behavior**
    • A node may aim at *sabotaging* the whole system just to harm it
      – Some people are just nasty or want to weaken the system for other reasons
      – e.g. claim a failure for each sub-transaction the node was responsible for ⇒ all global transaction involving that node fail

– **Malfunctions**
  • The more the autonomy increases, the more difficult it is to *detect* if a given system behavior is a real answer or a *malfunction*
4.3 Byzantine Agreements

• What to do if trust cannot be assured?
  – Byzantine agreements!
• Byzantium, 1453 AD.
  – Constantinople is the last bastion of the Eastern Roman Empire
  – Sultan Mehmed of the **Ottoman Empire** lies siege to the city with his army of 80,000 soldiers and many more irregulars
    • The city was **heavily fortified**, and held by 7000 knights
    • Strongest fortress of its time!
    • The Ottoman forces camped around the city, each camp led by a **general**
Although outnumbering the defenders, only a \textbf{coordinated attack} from all sides will bring the fortifications down

- \textbf{Coordination} between generals is only possible via messengers

Problem: messengers and/or generals may be \textbf{malicious} and trick the other generals into an uncoordinated attack!

- … even worse, malicious generals may even conspire!
- Uncoordinated attackers will be routed at the walls - battle lost!
• How can all non-malicious generals coordinate an simultaneous attack despite intervention of malicious generals or corrupted messengers?
  – **Byzantine Agreement scheme!**
• But how are dead generals related to computer science?

• **Menlo Park, 1982 AD.**
  – Joint work for NASA, the Army Research Office, and the Ballistic Missile Defense Systems Command
  – **How can multiple concurrent and potentially faulty computation processes reach a reliable shared agreement?**
    • Faulty processes are not just a little “off”, but may produce completely arbitrary results
4.3 Byzantine Agreements

• Initial use case scenario:
  – **Altitude measurement in airplanes / rockets**
    • Altitude measurement is very fault prone
    • Device works and measurement is more or less correct
    • Device does not work - measurement is completely off
      – … however, it cannot be detected if the device really works or not

• Nowadays popularly used for
  – Fault tolerance in multi-core processors
  – Medical devices
  – **Reliable distributed commits**
  – …
• Assumptions
  – Agreement
    • No two “good” generals agree on different outcomes
  – Validity
    • If all “good” generals start with the belief they are ready to attack, then the only possible outcome is to attack
  – Termination
    • All “good” generals eventually decide

• ‘Generals’ could be peers, database nodes, circuit switches, etc.
4.3 Byzantine Agreements

• For what percentage of **malicious nodes** can protocols be designed?
  
  – Triple Modular Redundancy ➔ > 3f nodes
    • Assuming f treacherous generals (malicious peers), we need at least (3f+1) peers to come to an agreement
  
4.3 Byzantine Agreements

• **Counterexample:** a system with only 3 peers
  – Each starts with an initial value (0 or 1)
  – **One peer is malicious**
  – Good nodes need to agree upon value (0 or 1)
  – Nodes act solely based on messages coming in along incident edges

• Assume there exists an algorithm that **allows** good nodes to agree
• Assume that N1 is a good peer
  – Scenario 1: N3 is treacherous
    • N2 relates that it is in state 0 to N1 and N3
    • But N3 relates to N1 that N2 is in state 1
  – Scenario 2: N2 is treacherous
    • N2 relates that it is in state 0 to N1 and that it is in state 1 to N3
    • N3 relates to N1 that N2 is in state 1
• Obviously N1 cannot distinguish the two scenarios
  – In both cases it would have to decide for a value of 0 for the respective loyal peer
• Now look at N3 in scenario 2
• Remember in scenario 2
  N2 is treacherous
    – N2 relates that it is in state 0 to N1 and that it is in state 1 to N3
    – N1 relates to N3 that it is in state 1
• N3 would have to decide for a value of 1 and thus vote with the loyal peer N1
• Contradiction: in scenario 2 N1 and N3 would both be loyal, but would still vote differently
• One peer starts the agreement process by broadcasting its value *(commander)*
  – Whenever a message is supposed to be sent, but a peer does not send it, it is detected, and a default value is assumed

• **Echo the result to all other peers**

• Do this for more peers than can be malicious
  – Algorithm is recursive with \((f + 1)\) levels

• **Bottom case: No traitors**
  – the commander broadcasts its initial value
  – every other process decides on the value it receives
4.3 Byzantine Agreement (n > 3f)

- Idea: Amplify the original message over different channels starting from (f+1) commanders
4.3 Byzantine Agreement (n > 3f)

• echo_broadcast(node C, message m)
  – C sends [initial,C,m] to all nodes
  – Every recipient replies with [echo,C,m] to all and ignores subsequent [initial,C,m’]
  – Upon receiving [echo,C,m] from (n+f)/2 distinct nodes, then a node accepts m from C

• Terminates? Yes — all non-malicious nodes accept (n-f) messages and exit both wait phases.
• If the system is initially proper (all non-malicious nodes have the same value m) then every such node terminates the algorithm with M=m.
4.3 Byzantine Agreement (n > 3f)

\[ C_i : M := M_i \]

for \( k = 1 \) to \( (f+1) \) do

(* Phase 1: SEND *)
broadcast \( M \);
wait to receive \( M \)-messages from \( (n-f) \) distinct processes;
\( proof := \) set of received messages;
\( count(1) := \) number of received messages with \( M = 1 \);
if \( count(1) > (n-2f) \) then \( M := 1 \)
else \( M := 0 \);

(* Phase 2: ECHO *)
echo_broadcast \([M, proof]\);
wait to accept \([M, proof]\)-messages, with a correct proof, from \((n-f)\) distinct processes;
\( count(1) := \) number of accepted messages with \( M = 1 \);
\( \text{Compute}\_\text{new}\_\text{vote}(s_k); \)
if \( (s_k = 0 \) and \( count(1) \geq 1 \)) or \( (s_k = 1 \) and \( count(1) \geq (2f+1) \)) then \( M := 1 \)
else \( M := 0 \);
If the Commander is not malicious (agreement by majority vote)

![Diagram of the Four Generals problem](image-url)
• If the Commander is malicious (no agreement possible)
• Partition nodes into three groups, with at least 1 and at most 1/3 of the nodes in each group

• **Theorem:** A Byzantine agreement can be solved in a network $G$ of $n$ nodes while tolerating $f$ faults if and only if
  
  $-n > 3f \text{ and}$
  
  $-\text{connectivity}(G) > 2f$

• Graph $G$ is $2f$-connected if the removal of $2f$ or more nodes will result in a disconnected graph (or a trivial 1-node graph)
Distributed Transactions

• Adapt methods already known from centralized transactions management
  – But: distributed databases have more potential failure sources
    • Network failures, replication, allocation, node failure, untrustworthiness,…
  – One approach: provide full ACID properties
    • Federated approach: use distributed commits (D2PC)
      – i.e. partition global transaction into sub-transactions
      – Each sub-transaction is executed locally
      – At the end, the coordinator votes if final commit should be performed
        » Sub-transaction either OK or failed
        » One sub-failure ⇒ global failure
– Homogenous approach: **Distributed 2-Phase-Locking (D2PL)**
  
  • Adapt 2PL for distributed usage
  • Distributed lock management necessary
  • Control over internal transaction management of nodes necessary

– Problem: how to deal with **untrustworthy** nodes?
  
  • Important in P2P or in loosely-coupled autonomous settings
    – e.g. web services
  • Nodes may be **malicious** or just **malfunctioning**
  • **Byzantine Agreements**!
    – Nodes **echo** received messages among each other to filter false information and untrustworthy nodes
• Recommended Reading:
• **Peer-Two-Peer Systems**
  – Classification of Peer-To-Peer Systems
  – Decentralized and Centralized P2P
  – Structured and Unstructured P2P
  – Early protocols