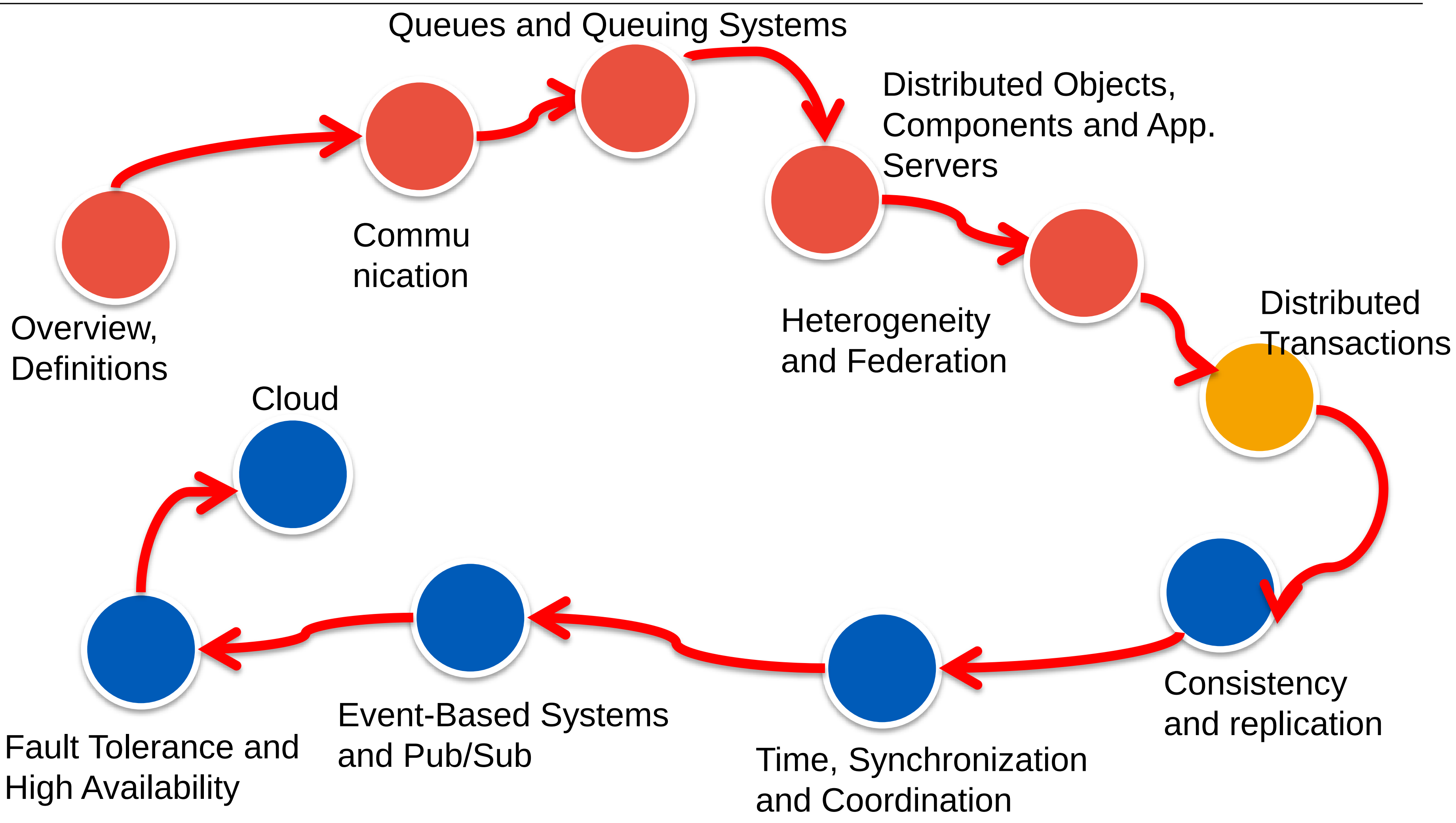
Middleware
6. Distributed Transaction Processing

TECHNISCHE UNIVERSITÄT DARMSTADT

A. Buchmann
Wintersemester 2014/2015

DVS

Topics


Queues and Queuing Systems
Distributed Objects, Components and App. Servers
Commu nication
Overview, Definitions
Heterogeneity and Federation
Distributed Transactions
Cloud
Consistency and replication
Fault Tolerance and High Availability
Event-Based Systems and Pub/Sub
Time, Synchronization and Coordination

Topics

- Distributed Transaction Processing
- Transaction models
- Commit Protocols: 2PC, 3PC
- Transaction Processing Monitors
- Transactional RPC
- Mobile Environments

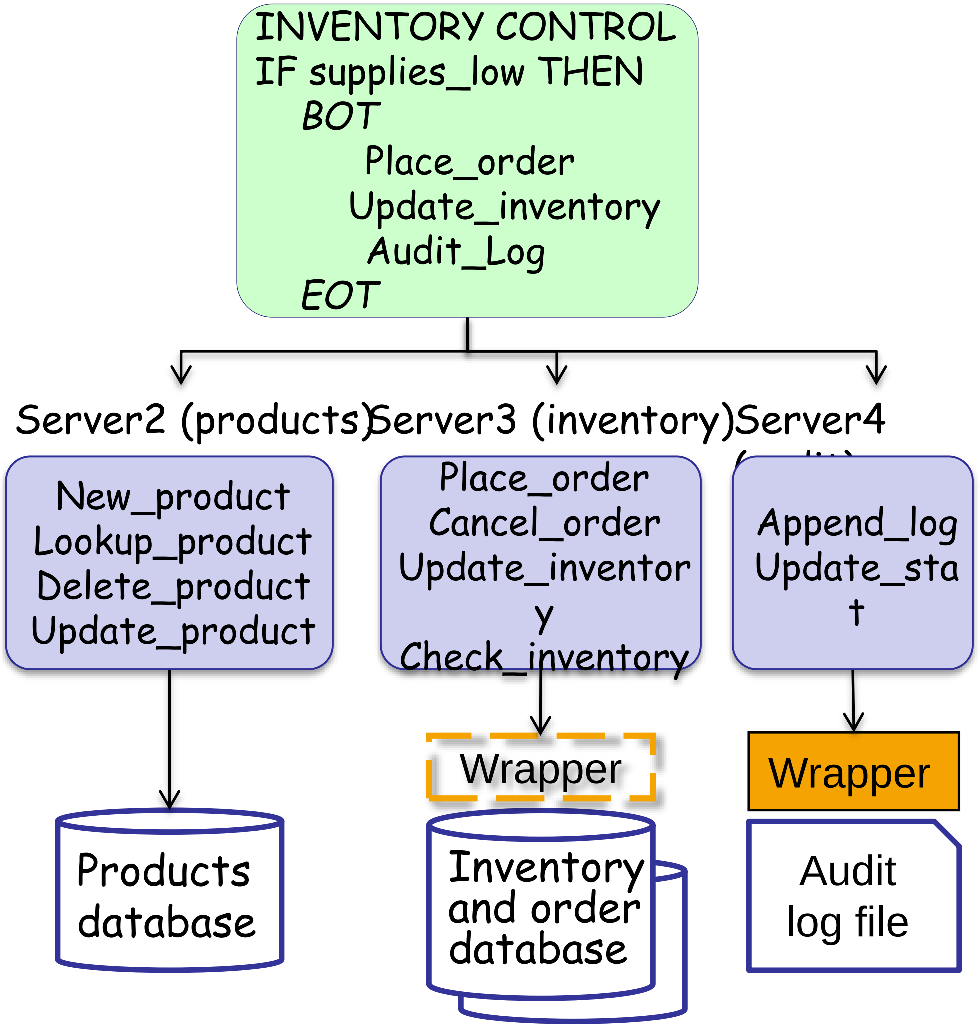
Reading for THIS Lecture

- The slides for the lecture are based on material from:
  - M. Tamer Özsu and Patrick Valduriez. 2011 **Principles of Distributed Database Systems** (3nd Ed.). Prentice-Hall.
    - Chapter 10, 12
  - Ahmed K. Elmagarmid (Ed.). 1992 **Database Transaction Models for Advanced Applications**. Morgan Kaufmann
    - Chapter 5: A.Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos, F.Manola. **A transaction model for active distributed object systems**.
  - Jim Gray and Andreas Reuter. 1992. **Transaction Processing: Concepts and Techniques**. Morgan Kaufmann.
    - Chapter 4, Sect. 5.3
  - Philip Lewis, Arthur Bernstein, Michael Kifer. 2001. **Databases and Transaction Processing: An Application-Oriented Approach** (1st ed.). Addison-Wesley
    - Chapter 21
  - G. Alonso, F. Casati, H. Kuno, V. Machiraju. 2004 **Web Services: Concepts, Architecture and Applications. Springer Verlag**
    - Chapter 2

Distributed Transactions

Created with wordle.net based on:
P. Bernstein. Middleware. CACM, Feb. 1996

Issues with Distributed Transactions



- In a single DB →Tx guarantees
- However:
- Data resides in different data stores
  - Not all are DB → Guarantees:
  - Atomicity - Recovery
  - Isolation - Concurrency control
- ACID guarantees do not apply
  - Operation spans multiple DB
  - Operation accesses non DB data
    - Server state
    - Files
    - Devices

Issues in a distributed scenario

- Transaction **structure** and transaction **models**
  - Very Relevant, but **not** necessarily coupled to distributed scenarios
- **Consistency** – how can global consistency be ensured?
  - Across multiple DB (data stores)
  - Replication
- **Distributed concurrency control**
- **Distributed commit protocols**
  - Recovery
- Do we **really always** need ACID transactions?
  - Atomicity of a print job that encounters an error half way through?

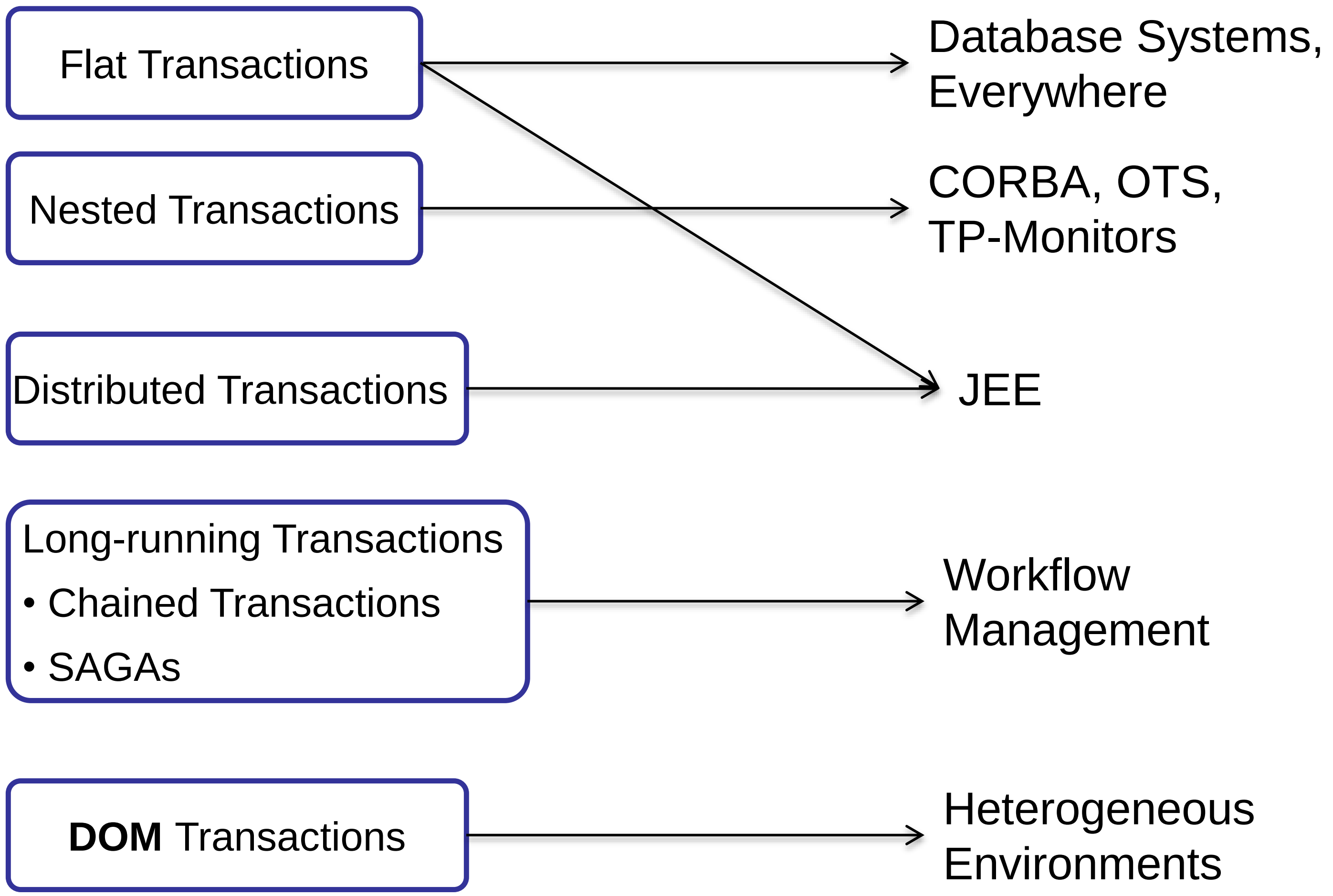
Transaction Models

Created with wordle.net based on:
P. Bernstein. Middleware. CACM, Feb. 1996

Transaction Models

- Flat Transactions
  - Savepoints
- Distributed Transactions
- Nested Transactions
  - Open Nested, Closed Nested
- Multi-level Transactions
- Long-running Transactions
  - Chained Transactions
  - SAGAs
- The **DOM** model

Use of Transaction Models



Flat Transaction

- Consists of:
  - Computation on local variables
    - not seen by DBMS
  - Access to DBMS using call or statement level interface
- No internal structure
- Accesses a single DBMS
- Adequate for simple applications
- Abort causes the execution of a program that restores the variables updated by the transaction to the state they had when the transaction first accessed them.
- Used in JEE (JTS) and most other middleware frameworks

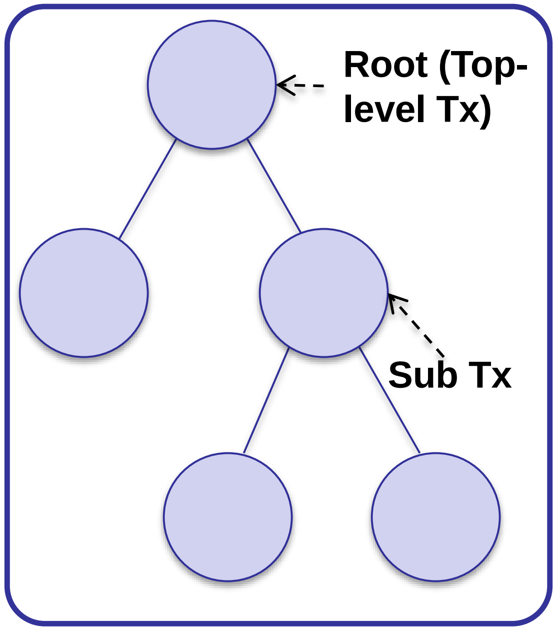
```
begin_transaction()
    ⋮
  EXEC SQL .....
    ⋮
  EXEC SQL .....
    ⋮
 if condition then
    ⋮   abort
 commit
end_transaction()
```

Limitations of Flat Transactions

- Only total rollback (abort) is possible
  - Partial rollback not possible
  - No structure within Tx → see Savepoints, Distributed Tx
- All work lost in case of crash
- Limited to accessing a single DBMS (different opinions across references)
- Entire transaction takes place in a time interval, commit point seen as point in time
  - Resource consumption
  - Not possible to spilt actions on a large dataset
  - Example:
    - Bank with 100 000 accounts → 100 Tx, each processing 1 000 acc.
    - Inconsistencies possible (interest rate increase after 77 931th Tx)

Nested Transactions

Nested Tx

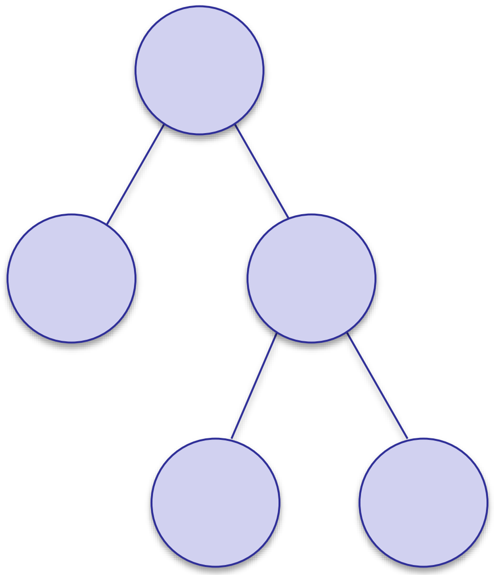


- **Problem**: Lack of mechanisms that allow
  - a top-down decomposition of Tx into subTx
  - individual subTx abort without aborting the entire Tx
    - Not so in Distr. Tx
- **Solution**: Nested Tx → globally isolated atomic
- Parent:
  - creates children to perform subtasks
    - sequentially or concurrently
  - waits until all children complete
  - no communication between parent and children.

J. E.B. Moss. 1985. *Nested Transactions: an Approach to Reliable Distributed Computing. Massachusetts Institute of Technology, Cambridge, MA, USA. (Ph.D Thesis)*

- Each subTx (together with its descendants):
  - is isolated with respect to each sibling (and its descendants).
  - hence, siblings are serializable,
  - but order is not determined i.e. nested Tx is non-deterministic.
- Concurrent nested transactions are serializable.

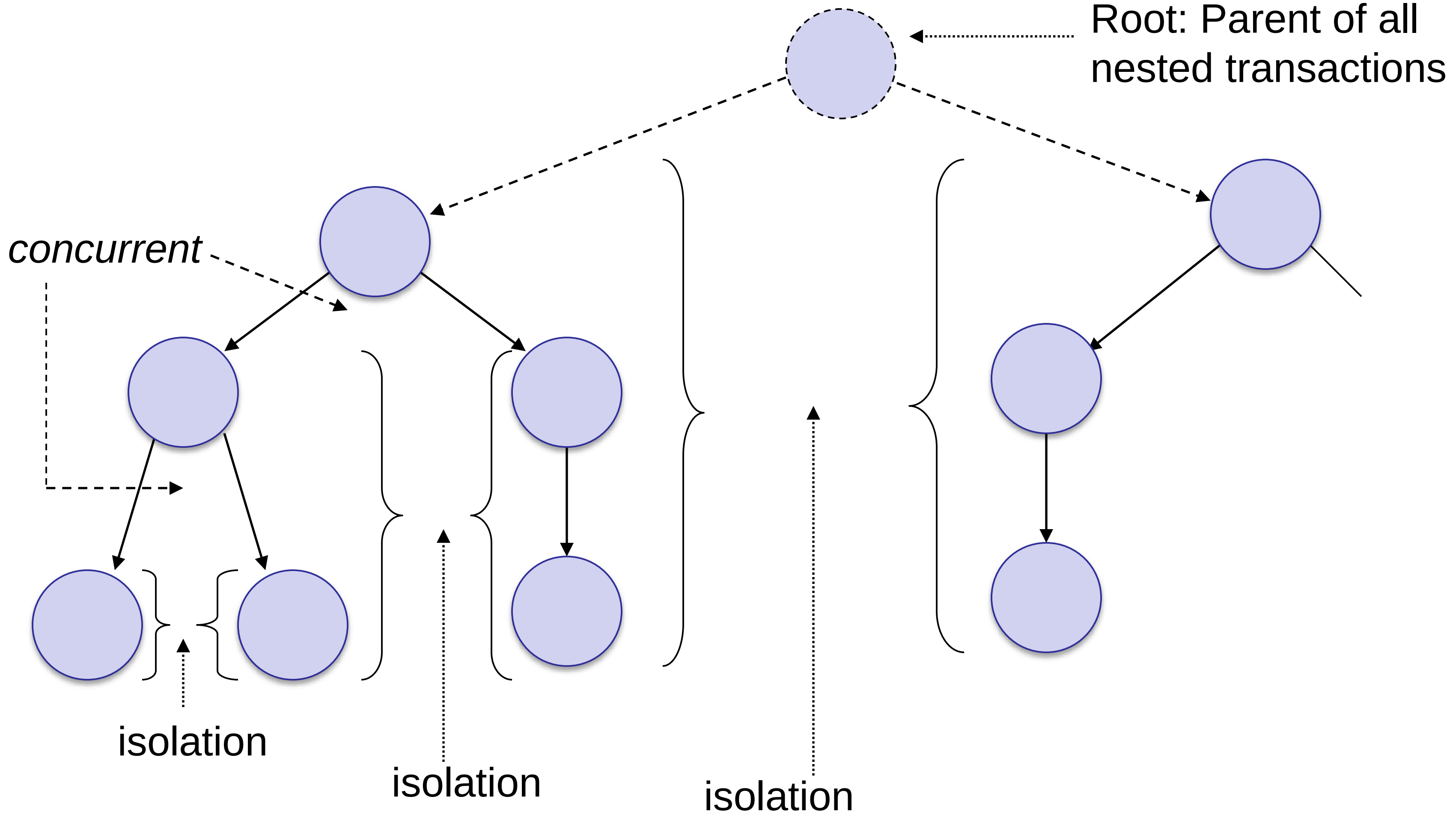
Characteristics of Nested Transactions



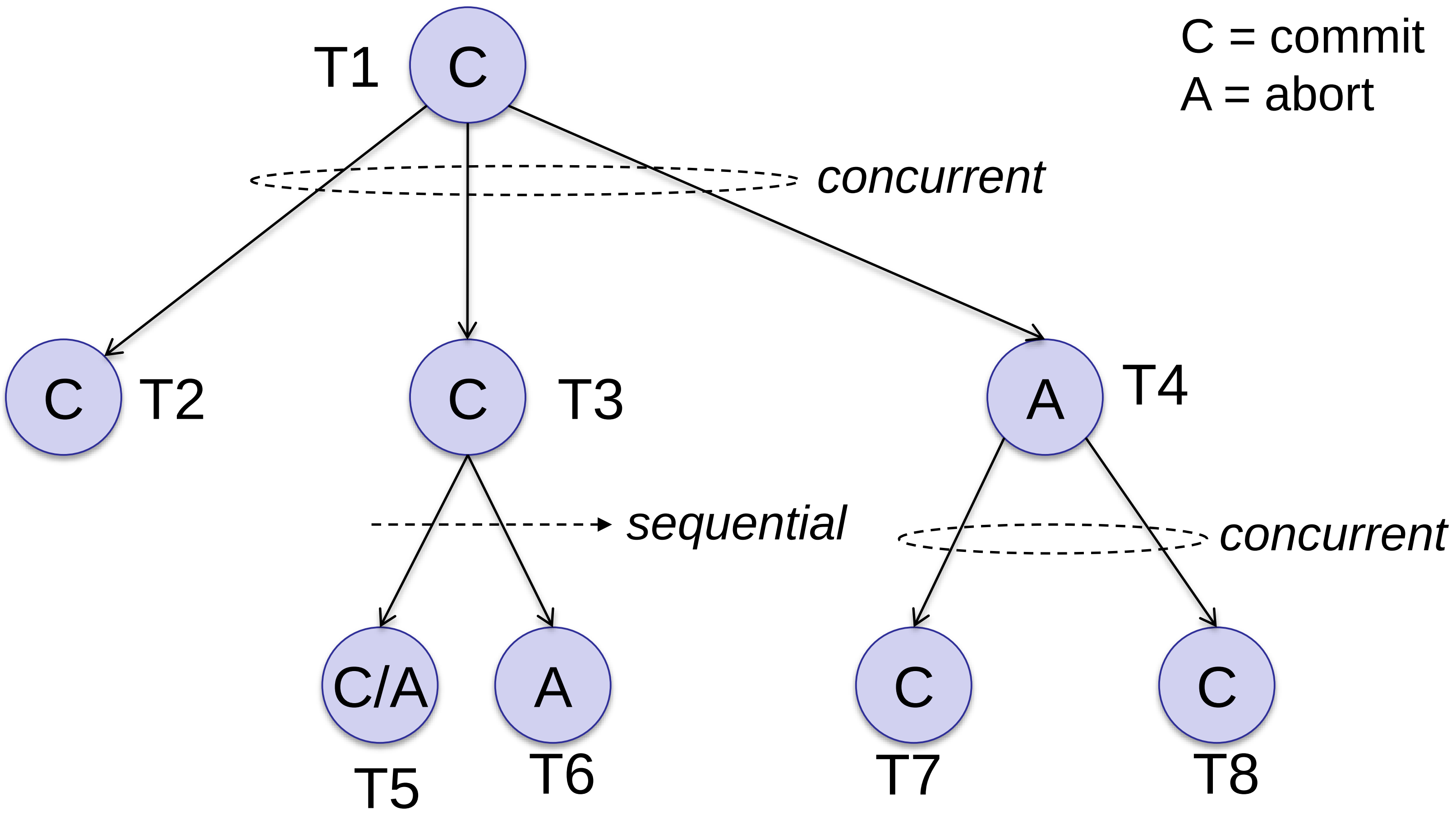
- A subtransaction is atomic:
  - aborts or commits independent of other subTx
- Commit is conditional on commit of parent
  - since child task is a subtask of parent task
- Nested Tx commits when root commits
  - At that point updates of committed subTx are made durable
- Abort causes abort of all subTx's children
- Consistency:
  - Individual subTx are not necessarily consistent,
  - nested Tx as a whole is consistent

Nested Transactions - Example

TECHNISCHE UNIVERSITÄT DARMSTADT



Nested Transaction - Example



C = commit

A = abort

Closed/Open Nested Transactions

- **Closed nested transaction** - atomicity enforced at the top-level
- The semantics of the closed transaction model are as follows:
  - A parent can create children sequentially or concurrently
  - A subTx and all of its descendents execute in isolation with respect to siblings
  - SubTx are atomic
    - Each sub-transaction can commit or abort independently to its parent.
    - **Commitment of subTx is dependant on the commitment of its parent**
    - If a subTx aborts, its operations have no effect. Control returned to parent
  - A subTx is not necessarily consistent. However, the nested transaction is consistent as a whole. (Data vs. Tx consistency)

- **Open nested transactions –** relax ‘top-level atomicity’ requirement
  - allow partial results of subTx to be seen outside transaction.
  - Compensating transactions

Chained Transactions

- Problem: If the system crashes during the execution of a long-running transaction, considerable work can be lost
- Chained Tx: : a new Tx starts after the previous commits or aborts
- Chaining allows a transaction to be decomposed into subTx with ntermediate commit points
- Database updates are made durable at intermediate points → less work is lost in a crash
- Savepoint: explicit rollback to arbitrary savepoint; all updates lost in a crash
- Chaining: abort rolls back to last commit; only the updates of the most recent transaction lost in a crash

S1
S2
S3
commit

$\Rightarrow$

S1;
commit;
S2;
commit;
S3;
commit;

S1; → ***update recs 1 - 1000***
commit;
S2; → ***update recs 1001 - 2000***
commit;
S3; → ***update recs 2001 – 3000***
commit;

Lewis, Bernstein,Kifer. Databases and Transaction Processing: An Application-Oriented Approach

Atomicity and Chaining

- Transaction as a whole is not atomic. If crash occurs
  - DBMS cannot roll the entire transaction back
    - Initial subtransactions have committed,
      - Their updates are durable
      - The updates might have been accessed by other transactions (locks have been released)
  - Hence, the application must roll itself forward
- Roll forward → on recovery the application determines committed work
  - Each subTx must tell successor where it left off
- Communication between successive subTx cannot use local variables
  - lost in a crash
  - Use database to communicate between subTx

- Chained Tx used in Workflow Management

Sagas

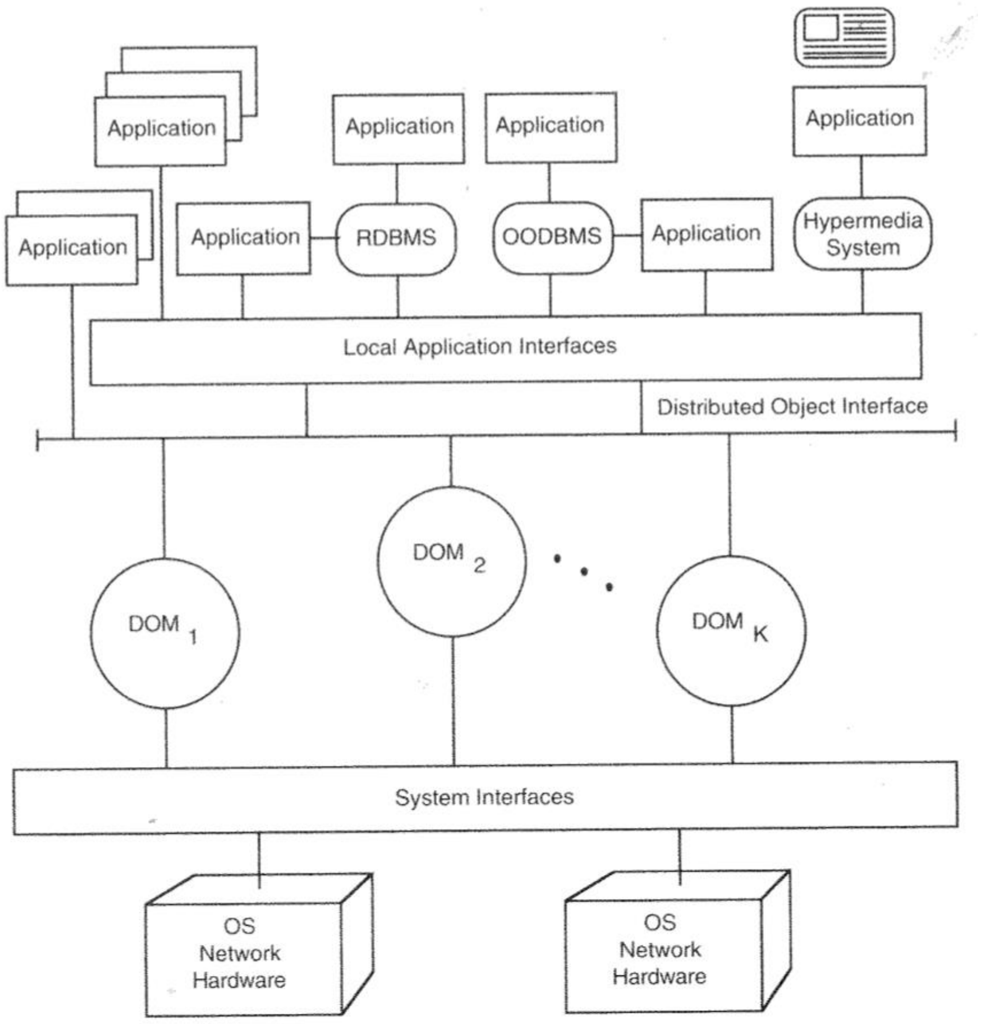
- For each subTx, STi,j in a chained transaction Ti a compensating transaction, CTi,j is designed
- Thus if T1 consisting of 5 chained subTx aborts after the first 3 subTx have committed, the compensation is
  - $ST_{1,1}$ $ST_{1,2}$ $ST_{1,3}$ $CT_{1,3}$ $CT_{1,2}$ $CT_{1,1}$
- Atomicity not guaranteed
  - when a Tx aborts, the value of every item is eventually restored to the value it had before that transaction started
  - However, complete atomicity is not guaranteed
    - Some other concurrent transaction might have read the changed value before it was restored to its original value
- Sagas used in Workflow Management

H. Garcia-Molina, K. Salem. Sagas. *SIGMOD '87*

The DOM Model (Distributed Obj. Management) [A.Buchmann et al. 1990]

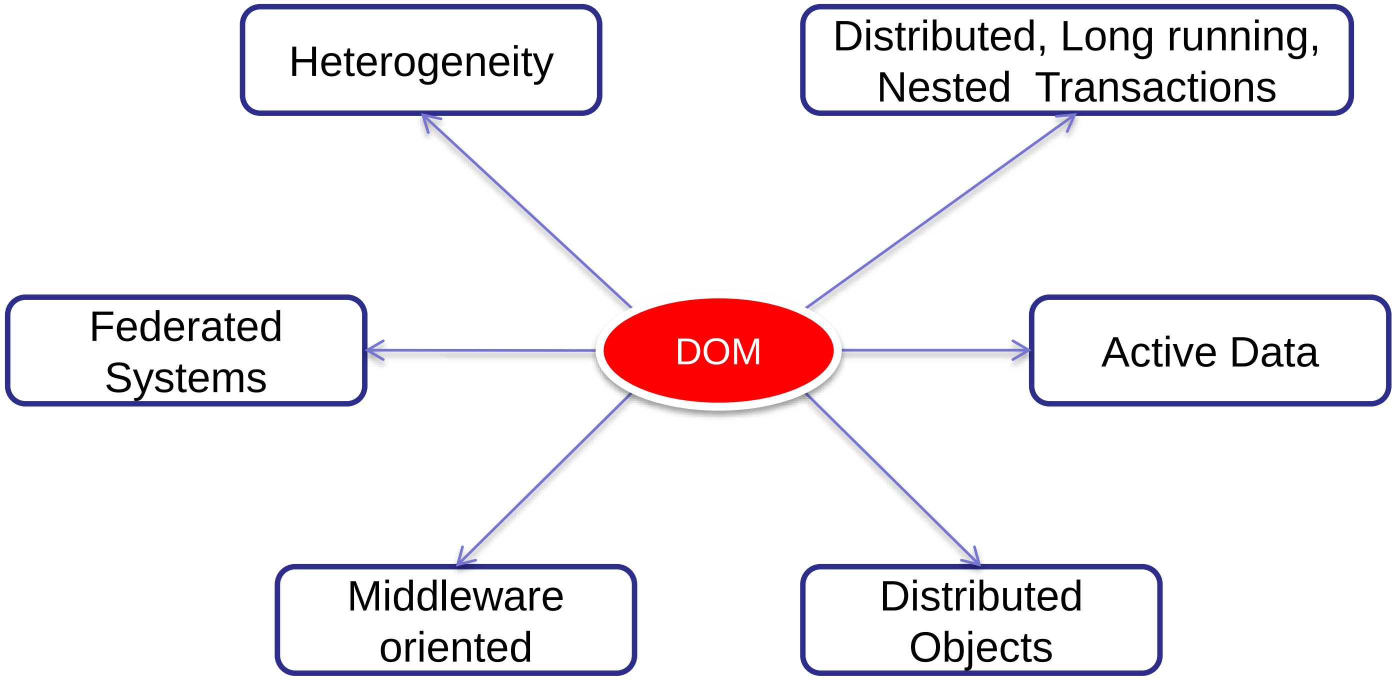
- The DOM model encompasses most other transaction models
  - Flat Tx | Distr. Tx | Nested Tx (Open, Closed) | Multilevel Tx
  - Homogenous and heterogeneous distributed environments (attached systems)
- Nomenclature:
  - **Multi Transactions** (multiTx) – a combination of topTx. Long running Tx accessing external resources (support commit, abort)
    - Comprise other multiTx or Nested topTx
    - **Top Nested Transaction** (topTx) – makes results visible outside multiTx upon commit
      - Closed nested Tx → root = top of the tree through which all commit
      - Compensating transaction ← Logical undo of partial committed results
    - Component Tx of a multiTx can force global abort
      - Vital – abort dependency → upon abort, the parent must abort too
      - Non-vital – no dependencies
    - **Contingency Tx**- executed if primary Tx fails.
      - Used to implement less preferable alternative actions
    - Precedence dependencies among component Tx may exist

Infrastructure for DOM

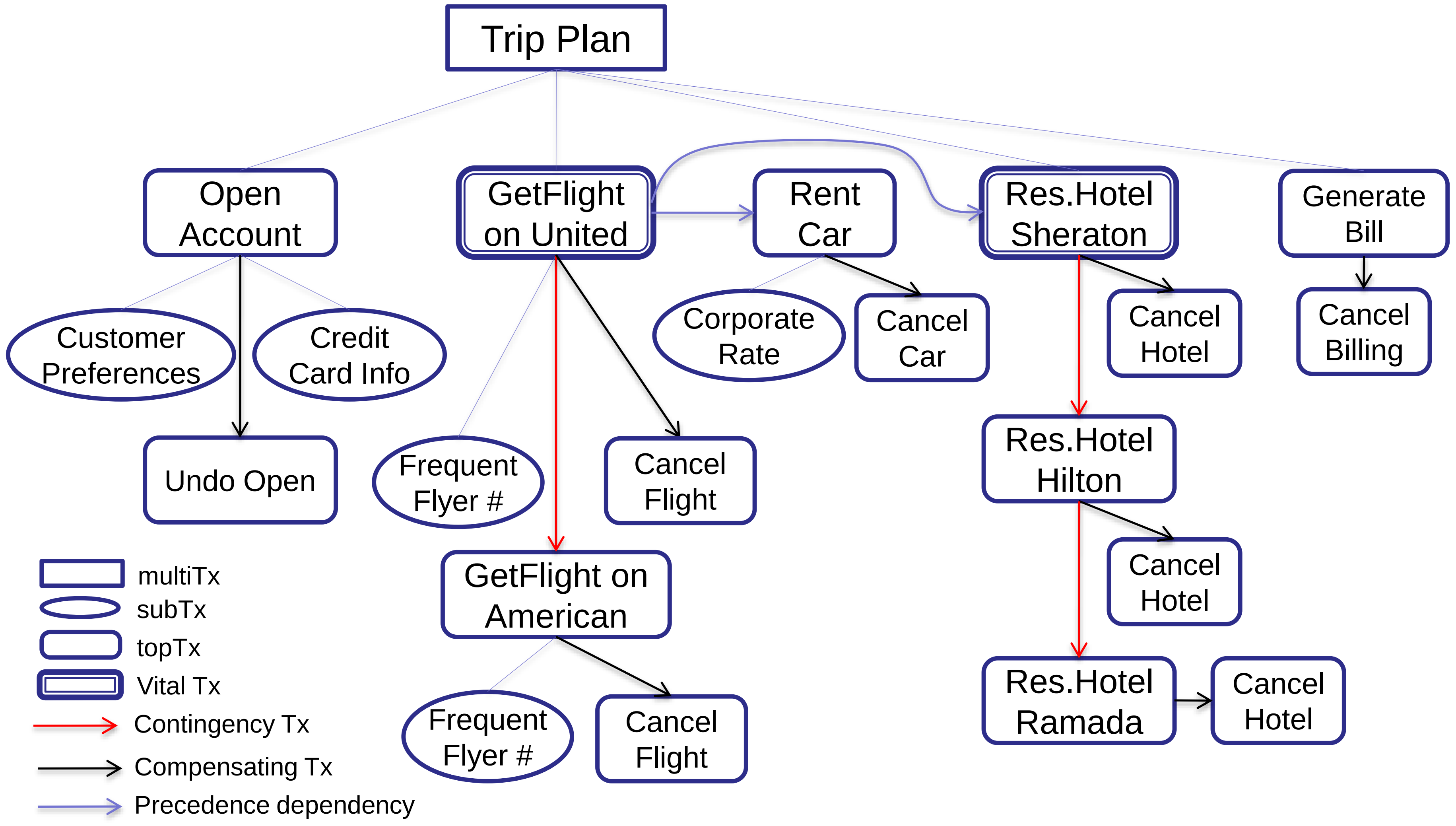


- DOM System – environment for
  - Distributed object application
  - Heterogeneous environment
  - Long running Tx
  - Active functionality
- Object Types
  - Native DOM objects → implemented in attached sys
  - Active objects
- LAI → Local Application interface
  - Interact with attached systems
  - Wrapper!

The DOM Model



Example of DOM a Transaction



Example of DOM a Transaction

- **MutiTx**: TripPlan
- **Nested TopTx**: OpenAccount, GetFlight, RentCar, ReserveHotel, GenerateBill
  - **Compensating Tx**: UndoOpen, CancelFlight, CancelCar, CancelHotel, ...
  - **Vital component Tx**: GetFlight, ReserveHotel
  - **Contingency Tx**: Reserve Hotel Hilton, Reserve Hotel Sheraton

- If TripPlan aborts all component Tx must abort
  - If componentTx already commited → compensating transaction
  - If vital component Tx aborits → multiTx must abort
- Component Tx execute in parallel
  - Precendence constraints specified → mechanism to enforce seq. Execution
  - GetFlight→ ReserveCar
  - Successor cannot start execution or commit before successor → rules

Explanation of a DOM Transaction

- Nested Tx
  - Closed Nested Tx, through which the whole tree commits
  - Explicitly built by user or result of firing rules from other Tx
    - Rules resulting from account preferences
    - The firing mode specified in rule:
      - immediate, deferred – nested Tx
      - detached → parallel execution, no commit dependencies
- Contingency Tx
  - Performs alternative/similar action to the original
  - Executed upon a failure in the original Tx
    - Compensating Tx already executed to undo actions
  - Failure condition needed in parent Tx, triggering event needed in contingencyTx
  - If multiple contingency Tx defined ordering through triggering events

Distributed Transactions

- **Goal**: distributed transaction should be ACID
  - Each subtransaction (subTx) is locally ACID
  - (local constraints maintained, locally serializable)

- In addition the transaction should be globally ACID
  - A: Either all subtransactions (subTx) commit or all abort – global atomicity
  - C: Global integrity constraints are maintained
  - I: Concurrently executing distributed transactions are globally serializable
  - D: Each subtransaction (subTx) is durable

Q: Are global ACID properties really maintained by today's frameworks?

  - We may have to settle for atomicity

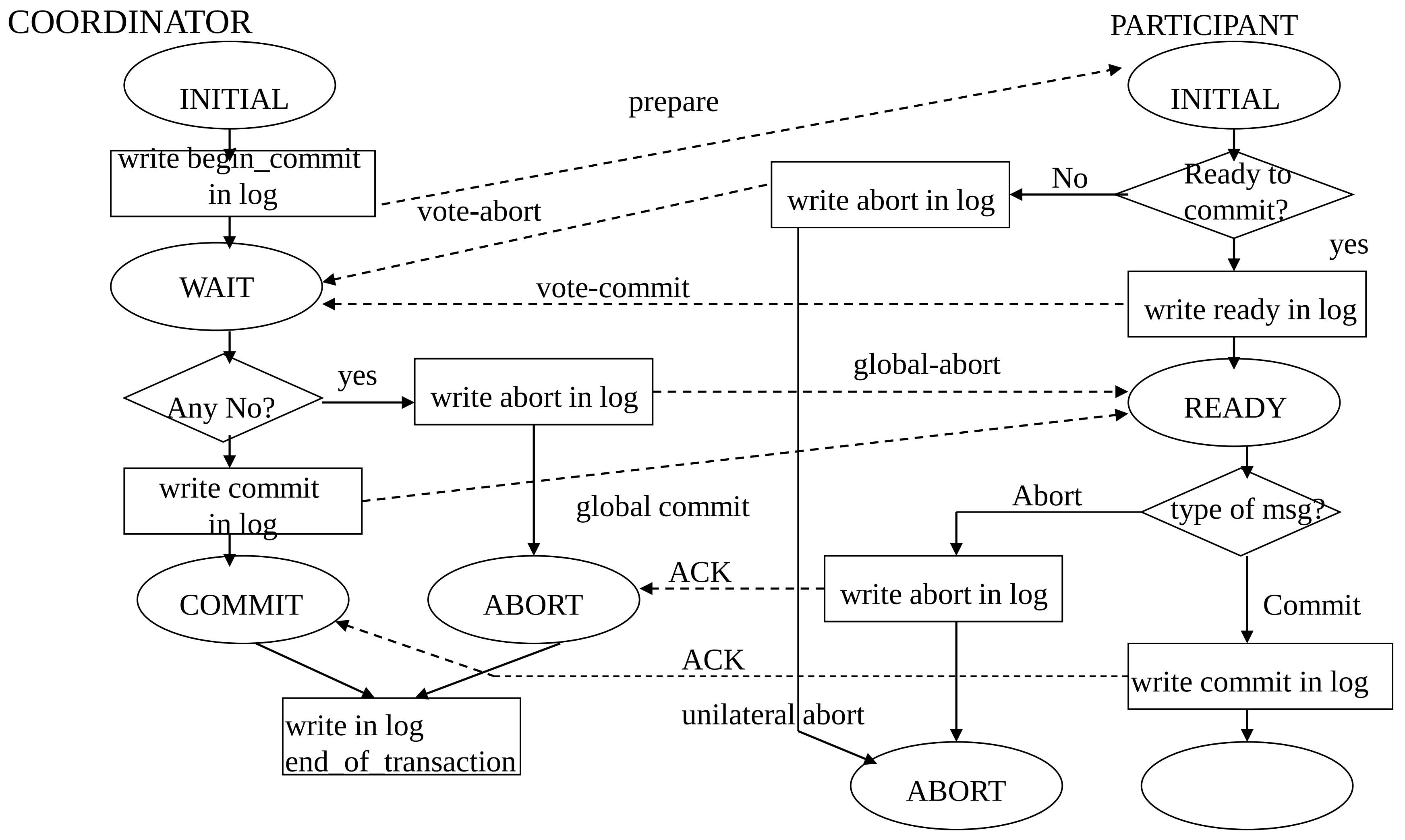
2PC and 3PC

Created with wordle.net based on:
P. Bernstein. Middleware. CACM, Feb. 1996

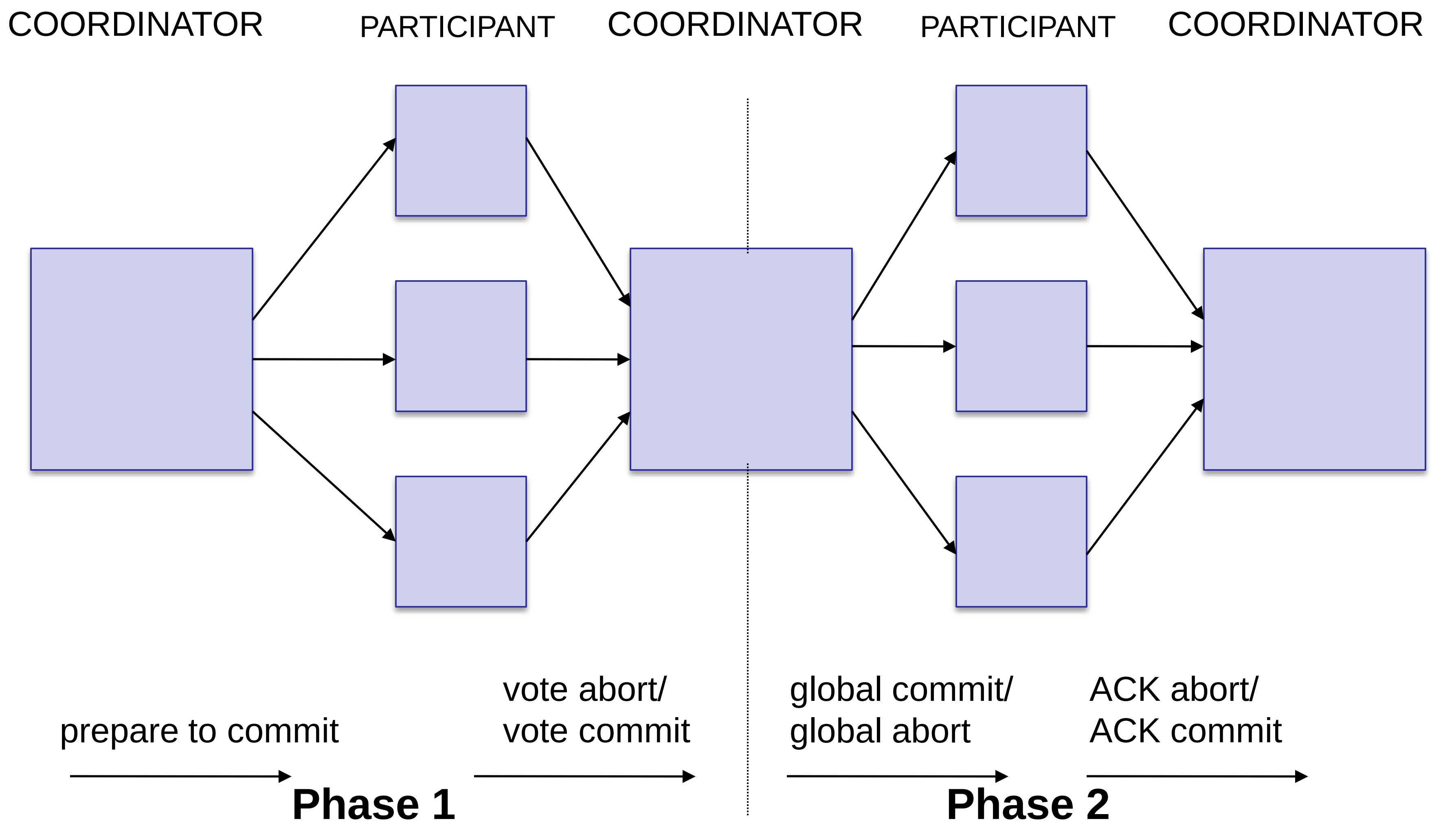
Two-Phase Commit

- Processing of transactions in distributed environments
- Phase 1: Get ready to write results
- Phase 2: Write the results
- Coordinator: Transaction Manager software at site where transaction originates
- Participant: Transaction Manager software at other sites executing the transaction

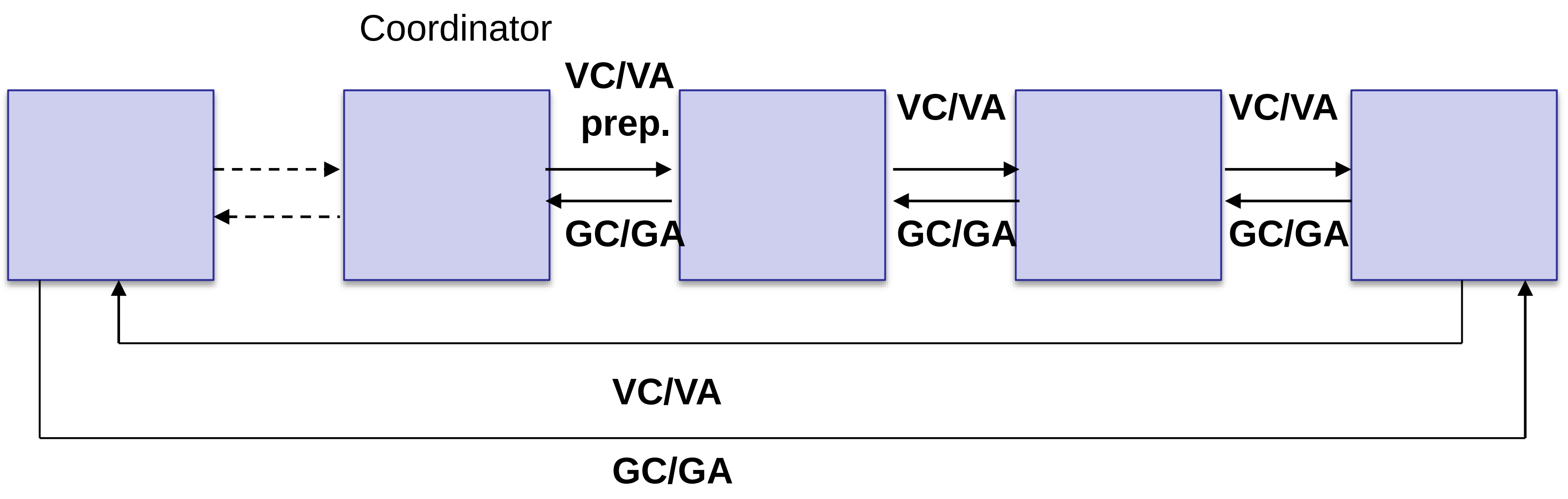
Two Phase Commit (2PC)



Centralized 2PC



Linear 2PC

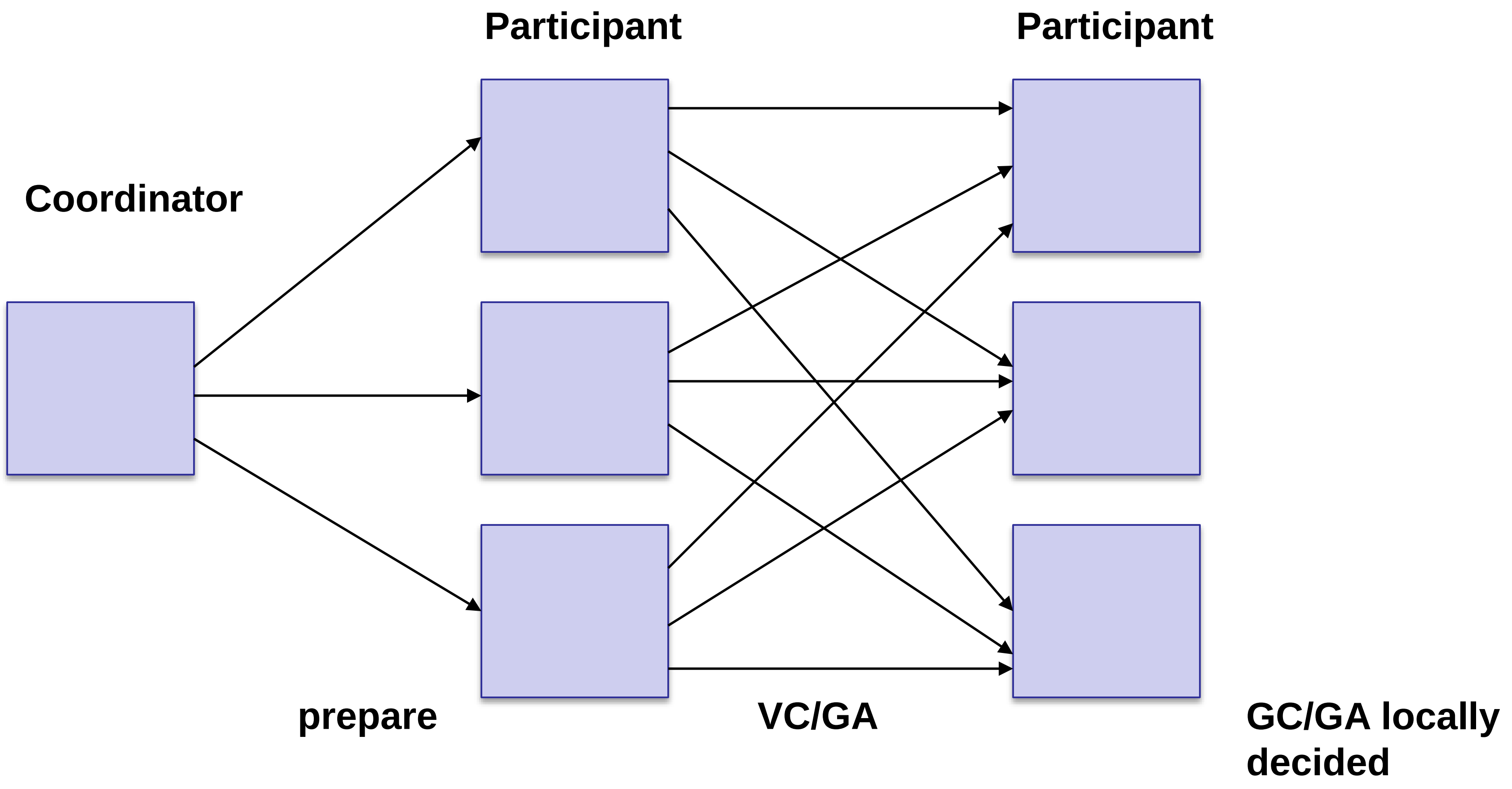


VC= vote commit
VA = vote abort

GC = global commit
GA = global abort

- Assumes knowledge of next node
  - node-list can be transmitted with TX.
- Fewer messages, no parallelism

Distributed 2PC



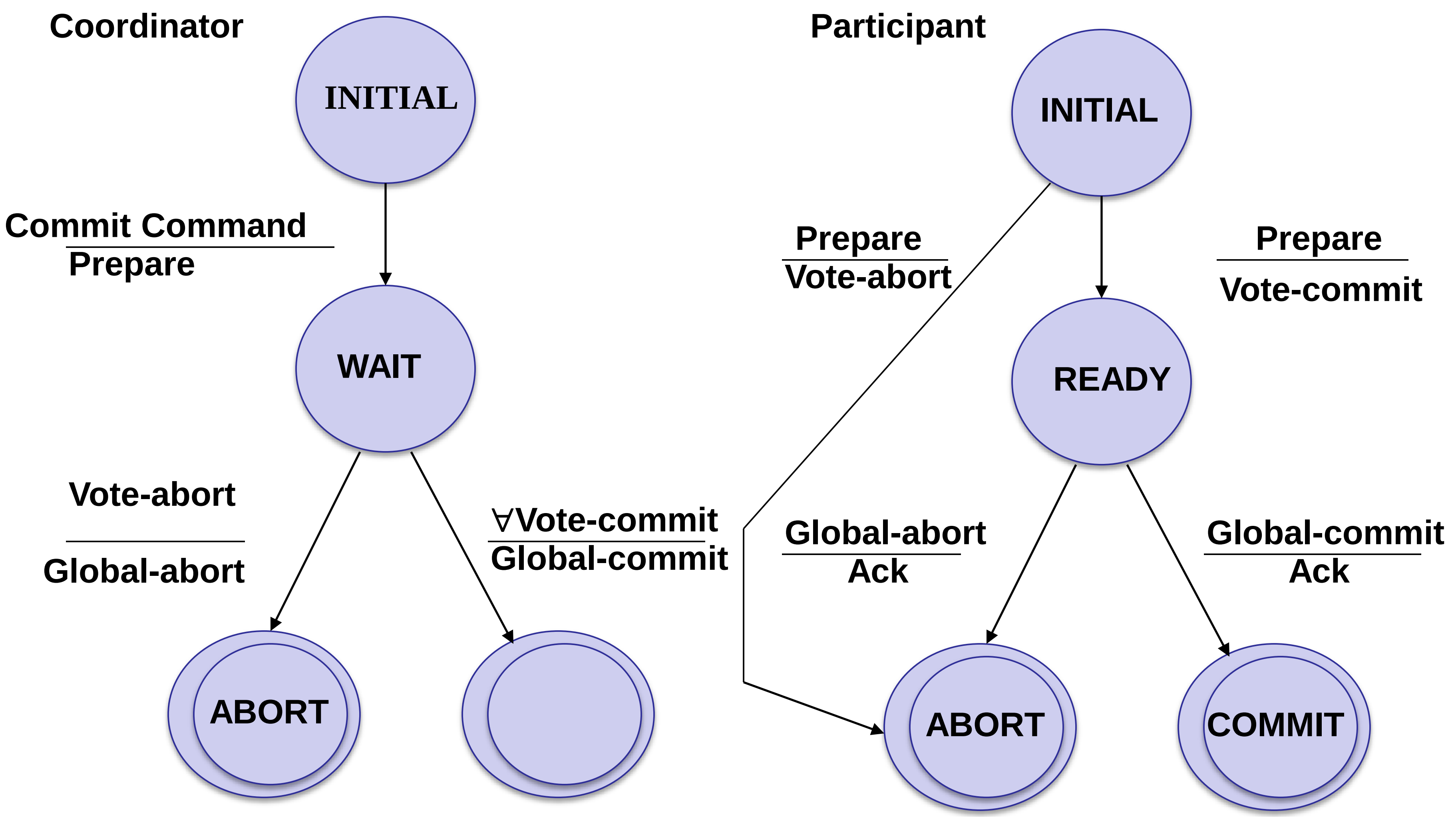
- Each node must know other participants

Communication costs

- Assuming **n** is the number of subordninates
  - or resource managers, or cohorts

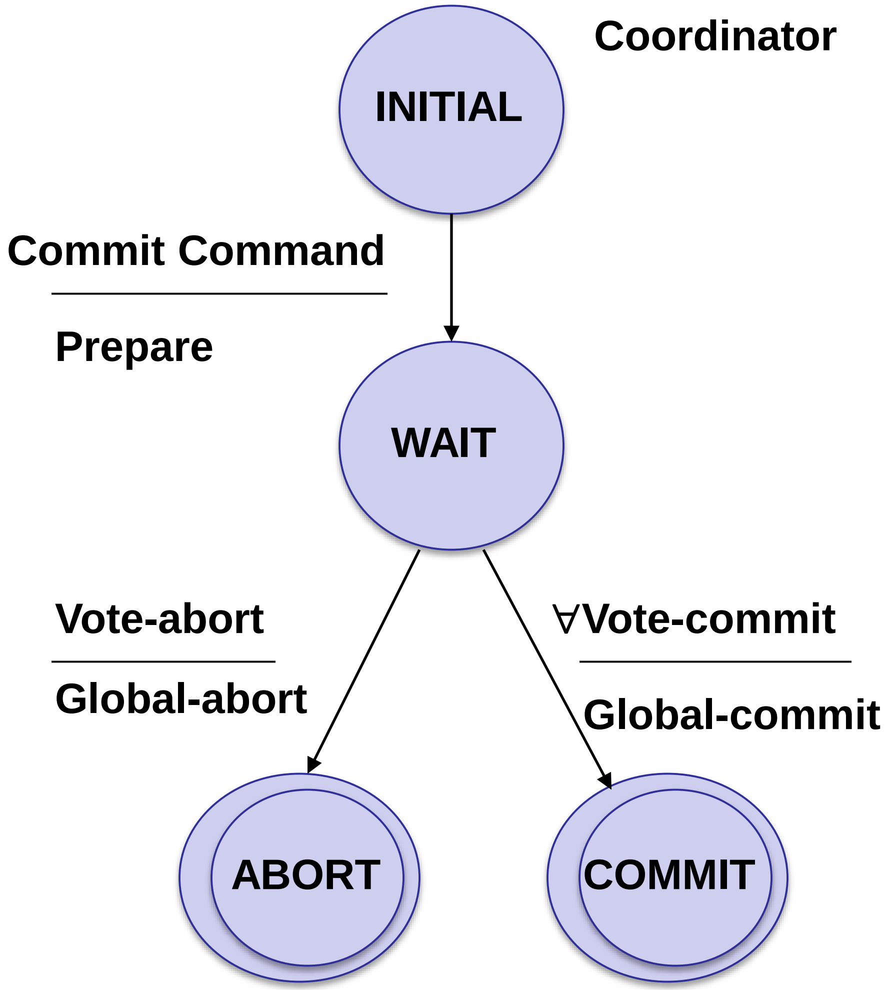
	Messages	**Rounds**
Centralized 2PC	3n	3
Decentralized 2PC	n^2+n	2
Linear 2PC	2n	2n

State Transitions in 2PC



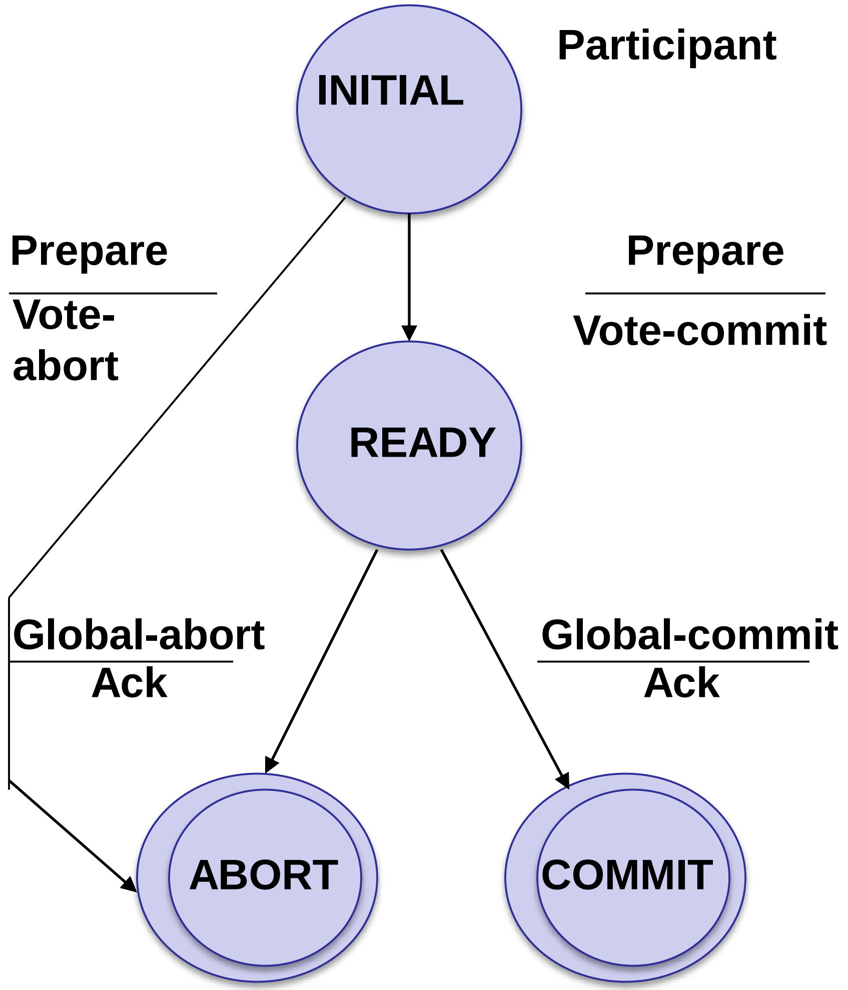
Site Failures 2PC Termination

- Timeout at destination based on problem at source
- Coord. TO in INITIAL
  - who cares
- Coord. TO in WAIT
  - cannot unilaterally commit
  - can unilaterally abort
- Coord. TO in ABORT or COMMIT
  - resend C/A message, stay blocked and wait for acks



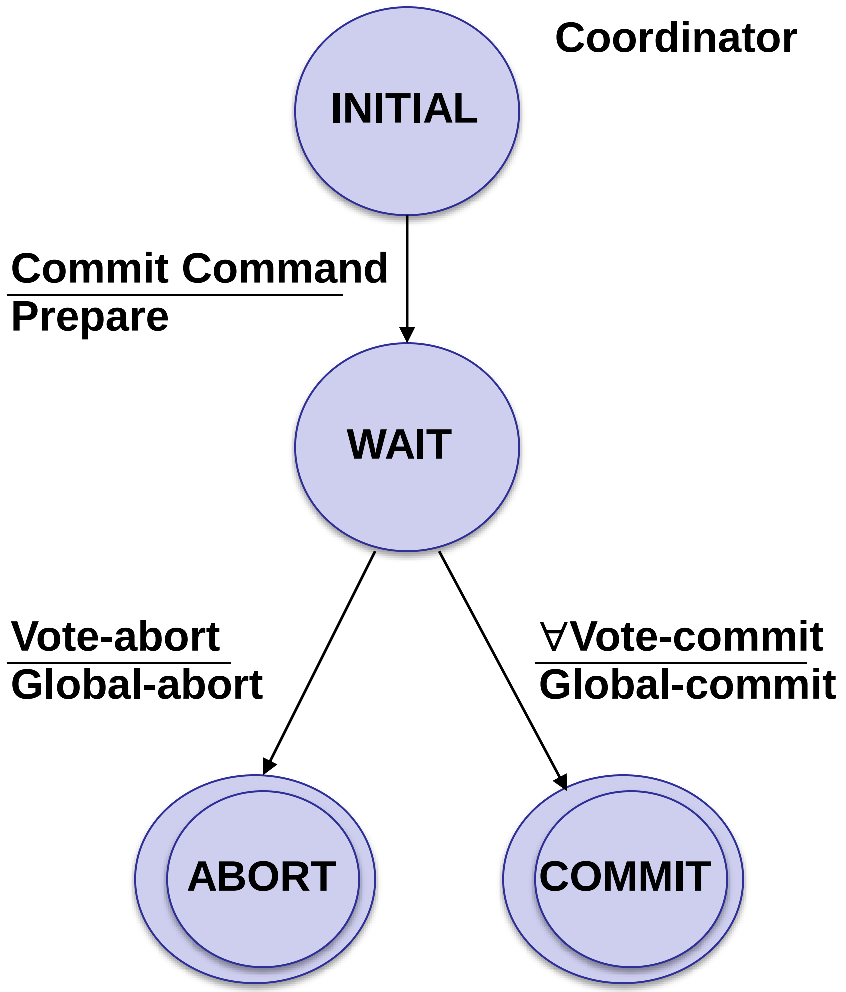
Site Failures - 2PC Termination

- Part. TO in INITIAL
  - coordinator must have failed in INITIAL state, unilaterally abort
- Part. TO in READY
  - stay blocked (can‘t change or unilaterally commit)
- If prepare arrives after unilateral abort
  - ignore, let coordinator TO in WAIT or check log and vote abort



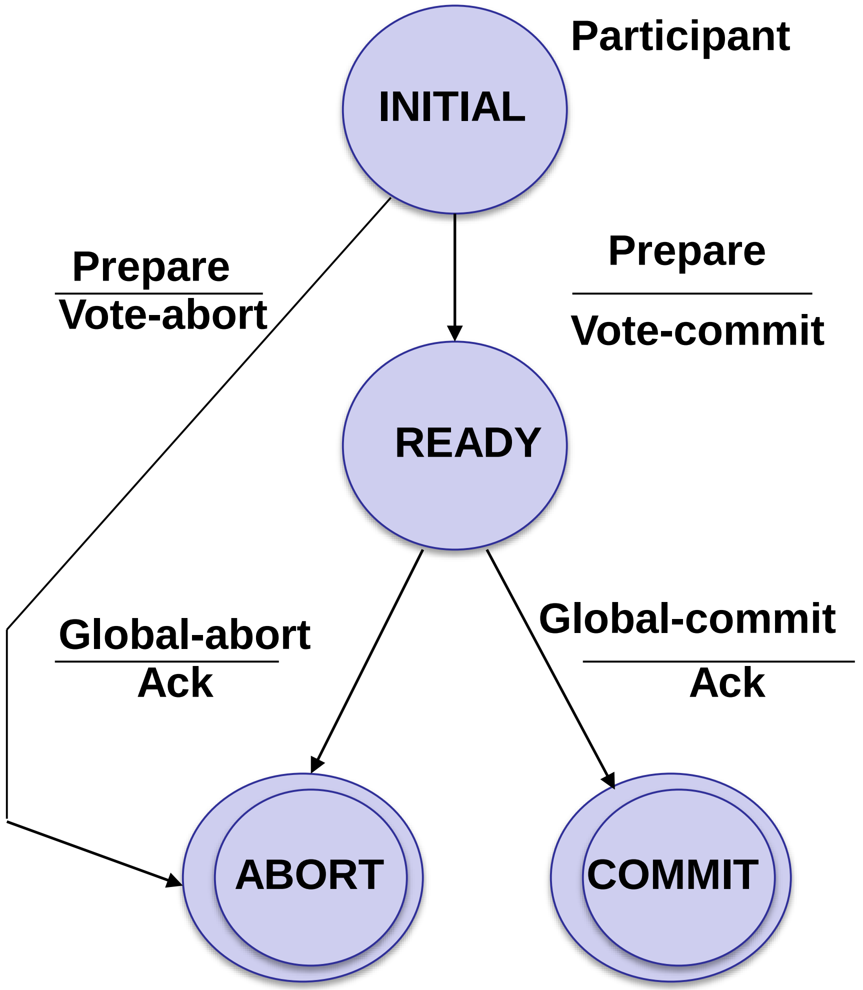
Site Failures - 2PC Recovery

- Failure in INITIAL - start commit proc. upon recovery
- Failure in WAIT - restart commit process upon recovery, resend prepare
- Failure in ABORT or COMMIT - nothing special if all acks were received, else use termination protocol
- Assumption: atomic log write and send, transition after message sent



Site Failures - 2PC Recovery

- Failure in INITIAL (coord. in WAIT, will TO in WAIT)
  - unilaterally abort upon rec.
- Failure in READY
  - coordinator has been informed about local decision
  - treat as TO in READY state, invoke termination prot., stay blocked until global decision is known
- Failure in ABORT or COMMIT
  - nothing special



2PC Recovery Protocols

- Additional cases that arise due to non-atomicity of log and message send actions
- coordinator site fails after writing „begin_commit“ log and before sending „prepare“ message
  - treat as a failure in WAIT state, send „prepare“
- participant fails after writing „ready“ record in log but before „vote commit“ is sent
  - treat as failure in READY state, alternatively, can send „vote_commit“ upon recovery

2PC Recovery Protocols (cont.)

- Participant fails after writing „abort“ record in log but before „vote_abort“ is sent
  - no need to do anything upon recovery
- Coordinator fails after logging its final decision but before sending decision to participants
  - coordinator treats it as failure in COMMIT or ABORT state, participants as TO in READY state
- Participant fails after writing „commit“ record in log but before sending message
  - participant treats it as failure in COMMIT
  - coordinator treats it as TO in COMMIT

Problems with 2PC

- Blocking
  - READY implies that participant waits for coordinator
  - if coordinator fails, site is blocked until recovery
  - blocking reduces availability
- Independent recovery is not possible because of blocking termination protocol
- Known: independent recovery protocols exist only for single site failures, no independent protocols for multiple site failures

Reasons for 2PC problems

- Commit protocol is non-blocking iff
  - it is synchronous within one state transition
  - its state transition diagram contains
    - no state that is adjacent to both a commit and abort state
    - no non-commitable state which is adjacent to a commitable state

- 3PC is non-blocking and fulfills conditions

- 3PC rarely used in practice because of overhead

Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail

- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so
  - Under 2 PC each site is obligated to wait for decision from coordinator
  - Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure.

Phase 2. Recording the Preliminary Decision

- Coordinator adds a decision record (<abort T> or < precommit T>) in its log and forces record to stable storage.
- Coordinator sends a message to each participant informing it of the decision
- Participant records decision in its log
- If abort decision reached then participant aborts locally
- If pre-commit decision reached then participant replies with <acknowledge T>

Phase 3. Recording Decision in the Database

- Executed only if decision in phase 2 was to precommit
- Coordinator collects acknowledgements. It sends <commit T> message to the participants as soon as it receives K acknowledgements.
- Coordinator adds the record <commit T> in its log and forces record to stable storage.
- Coordinator sends a message to each participant to <commit T>
- Participants take appropriate action locally.

Handling Site Failure

- Site Failure. Upon recovery, a participating site examines its log and does the following:
  - Log contains <commit T> record: site executes redo (T)
  - Log contains <abort T> record: site executes undo (T)
  - Log contains <ready T> record, but no <abort T> or <precommit T> record: site consults Ci to determine the fate of T.
    - if Ci says T aborted, site executes undo (T) (and writes <abort T> record)
    - if Ci says T committed, site executes redo (T) (and writes < commit T> record)
    - if c says T committed, site resumes the protocol from receipt of precommit T message (thus recording <precommit T> in the log, and sending acknowledge T message sent to coordinator).

Handling Site Failure (Cont.)

- Log contains <**precommit** *T*> record, but no <**abort** *T*> or <**commit** *T*>: site consults Ci to determine the fate of *T*.
  - if C_i says *T* aborted, site executes **undo** (*T*)
  - if C_i says *T* committed, site executes **redo** (*T*)
  - if C_i says *T* still in precommit state, site resumes protocol at this point
- Log contains no <**ready** *T*> record for a transaction *T*: site executes **undo** (*T*) writes <**abort** *T*> record.

Coordinator – Failure Protocol

1. The active participating sites select a new coordinator, C_{new}
2. C_{new} requests local status of T from each participating site
3. Each participating site including C_{new} determines the local status of T:

- **Committed**. The log contains a < **commit** *T*> record
- **Aborted**. The log contains an <**abort** *T*> record.
- **Ready**. The log contains a <**ready** *T*> record but no <**abort** *T*> or <**precommit** *T*> record
- **Precommitted**. The log contains a <**precommit** *T*> record but no <**abort** *T*> or <**commit** *T*> record.
- **Not ready**. The log contains neither a <**ready** *T*> nor an <**abort** *T*> record.

A site that failed and recovered must ignore any **precommit** record in its log when determining its status.

4. Each participating site records sends its local status to C_{new}

Coordinator Failure Protocol (Cont.)

5. C_{new} decides either to commit or abort *T*, or to restart the three-phase commit protocol:

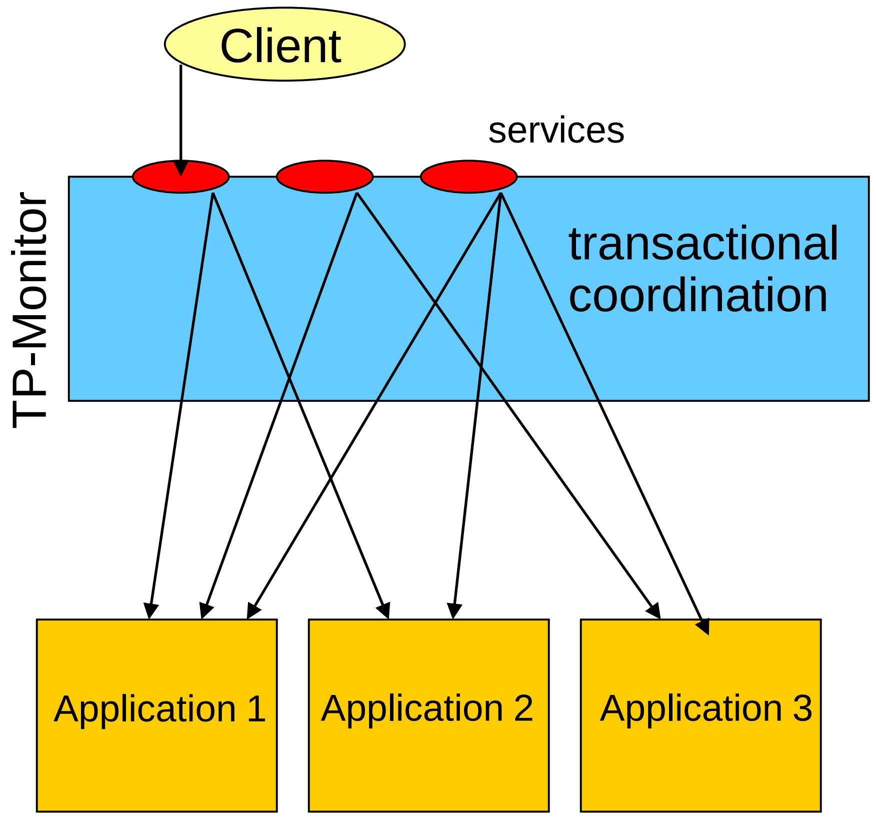
- Commit state for any one participant $\Rightarrow$ commit
- Abort state for any one participant $\Rightarrow$ abort.
- Precommit state for any one participant and above 2 cases do not hold $\Rightarrow$

  A precommit message is sent to those participants in the uncertain state. Protocol is resumed from that point.
- Uncertain state at all live participants $\Rightarrow$ abort. Since at least *n* - *k* sites are up, the fact that all participants are in an uncertain state means that the coordinator has not sent a <**commit** *T*> message implying that no site has committed *T*.

Transactional Processing
TP-Monitors

Created with wordle.net based on:
P. Bernstein. Middleware. CACM, Feb. 1996

TP-Monitors = transactional RPC



- A TP-Monitor allows building a common interface to several applications while maintaining or adding transactional properties.
- A TP-Monitor extends the transactional capabilities of a database beyond the database domain. It provides the mechanisms and tools necessary to build applications in which transactional guarantees are provided.
- TP-Monitors are, perhaps, the best, oldest, and most complex example of middleware.

Gustavo Alonso et al, Web Services. Springer Veralg 2003.

TP-Monitor functionality

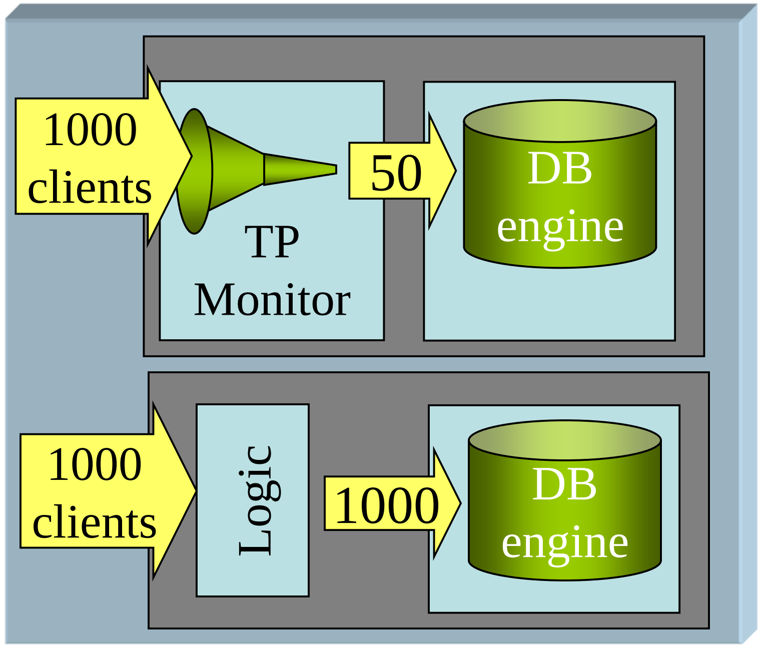
- TP-Monitor functionality is not well defined
  - system dependent.
- A TP-Monitor tries to cover the deficiencies of “all purpose” systems.
  - What it does is determined by the systems it tries to ”improve”.
- A TP-Monitor is an integration tool.
  - Tie heterogeneous system components
  - using a number of utilities that can be mixed depending on the characteristics of each case.
- A TP-Monitor addresses the problems of sharing data from heterogeneous, distributed sources, providing clean interfaces and ensuring ACID properties.
- A TP-Monitor extrapolates the functions of a transaction manager (locking, scheduling, logging, recovery) and controls the distributed execution.
  - As such, TP-Monitor functionality is at the core of the integration efforts of many software producers (databases, workflow systems, CORBA, JEE …).
- A TP-Monitor also controls and manages distributed computations. It performs load balancing, monitoring of components, starting and finishing components as needed, routing of requests, recovery of components, logging of all operations, assignment of priorities, scheduling, etc.

Gustavo Alonso et al, Web Services. Springer Veralg 2003.

TP Monitors

- OLTP functions typically simple, 10 disk I/Os, 10^5 to 10^7 instructions
- Many terminals (clients): 1000 to 100000
- TPMs are typically built according to a three tier architecture

- Automatically manage the entire environment that the business system runs in, including:
  - transactions
  - resource management
  - fault tolerance
  - load balancing
  - communications
  - scheduling
  - Recovery
  - High availability
- procedure oriented
- maximize the reuse of scarce system resources
  - high-volume of transactions



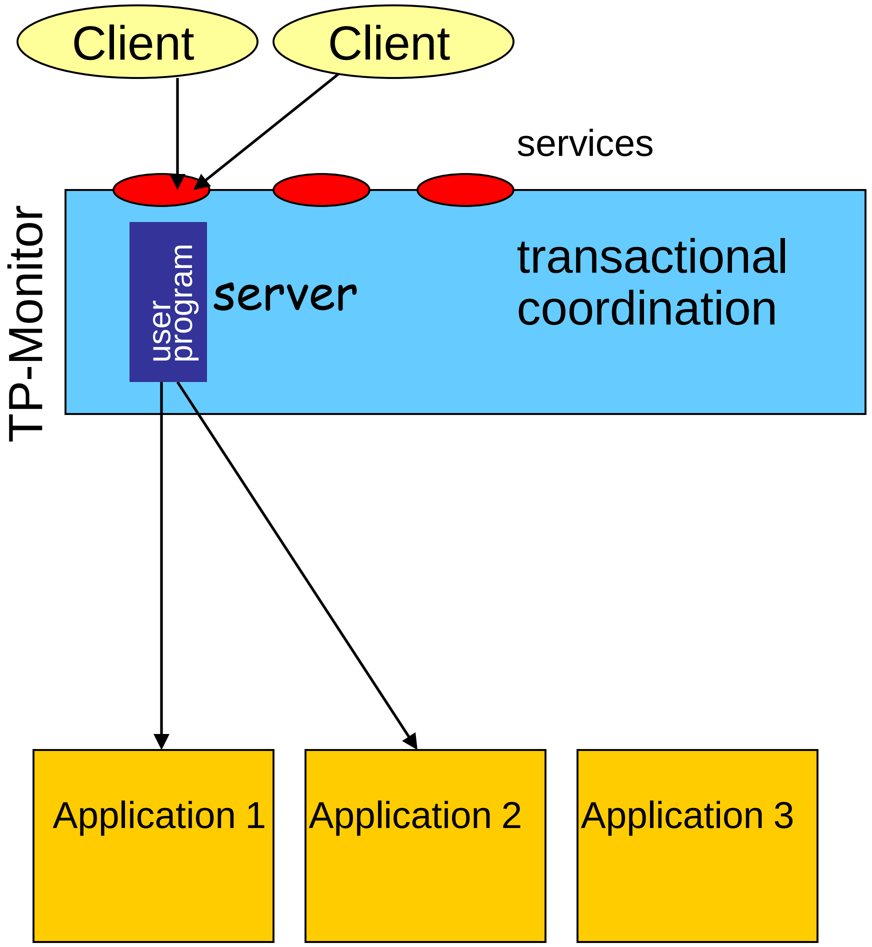
Commercial TPM Products

- **CICS** (IBM)
- IMS (IBM)
- **TUXEDO** (BEA/Oracle) 80%
- ACMS (HP/Digital)
- Encina (HP/Transarc)
- TOP END (AT&T/NCR)
- Pathway/TS (Tandem)
- Adabas TPF (Software AG)

- **Microsoft Transaction Server (Microsoft)**
- **X/Open DTP** standard
- OTS
  - JTS

Recommended reading:http://www.oracle.com/technetwork/middleware/tuxedo/overview/oracle-tuxedo-12c-datasheet-1721265.pdf?ssSourceSiteId=ocomde

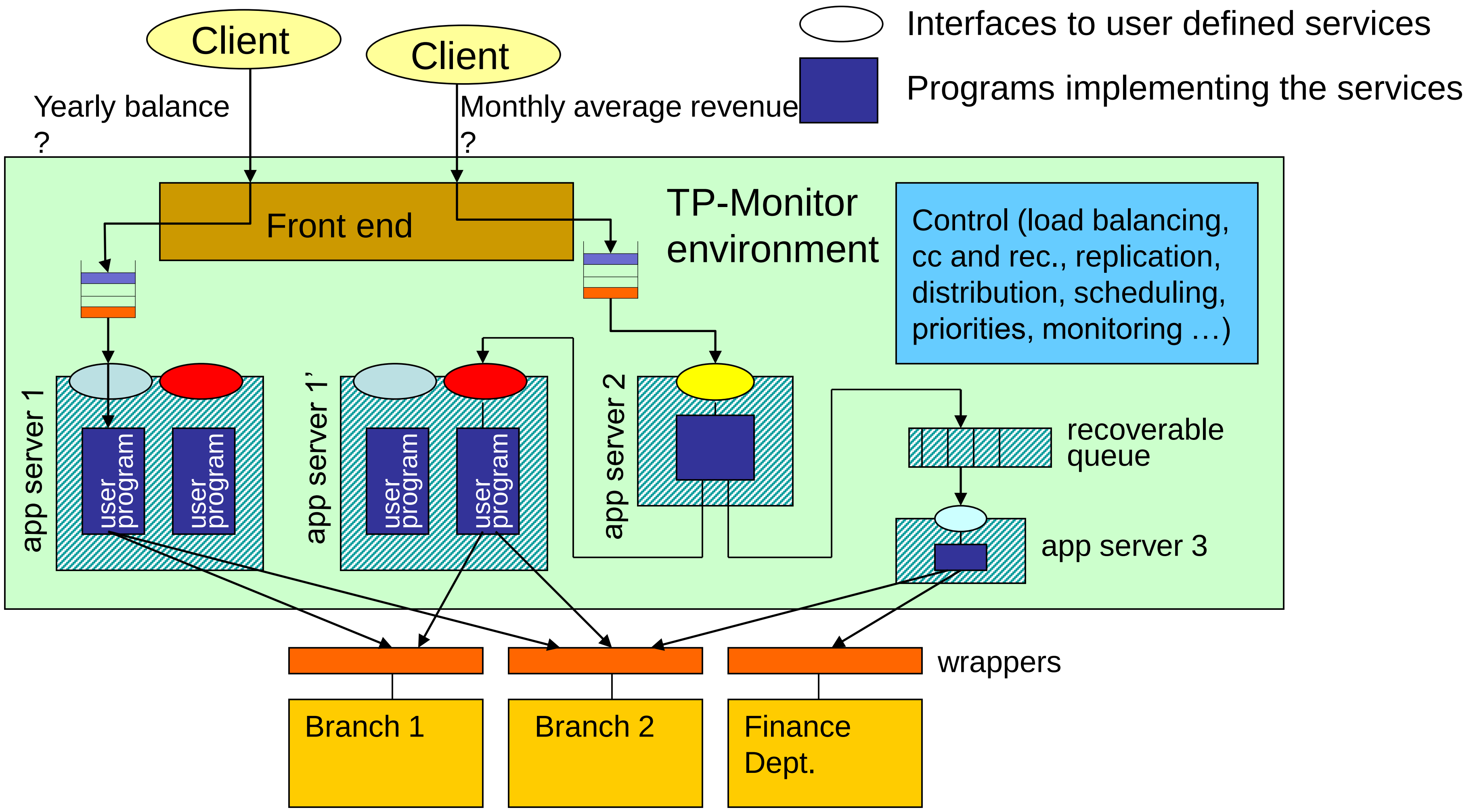
Transactional properties



- The TP-monitor encapsulates the services with transactional brackets
  - atomicity: a service that produces modifications in several components should be executed entirely and correctly in each component or should not be executed at all (in any of the components).
  - isolation: if several clients request the same service at the same time and access the same data, the overall result will be as if they were alone in the system.
  - consistency: a service is correct when executed in its entirety (it does not introduce false or incorrect data into the component databases)
  - durability: the system keeps track of what has been done and is capable of redoing and undoing changes in case of failures.

Gustavo Alonso et al, Web Services. Springer Veralg 2003.

TP-Monitor, generic architecture



Tasks of a TP Monitor

Core services

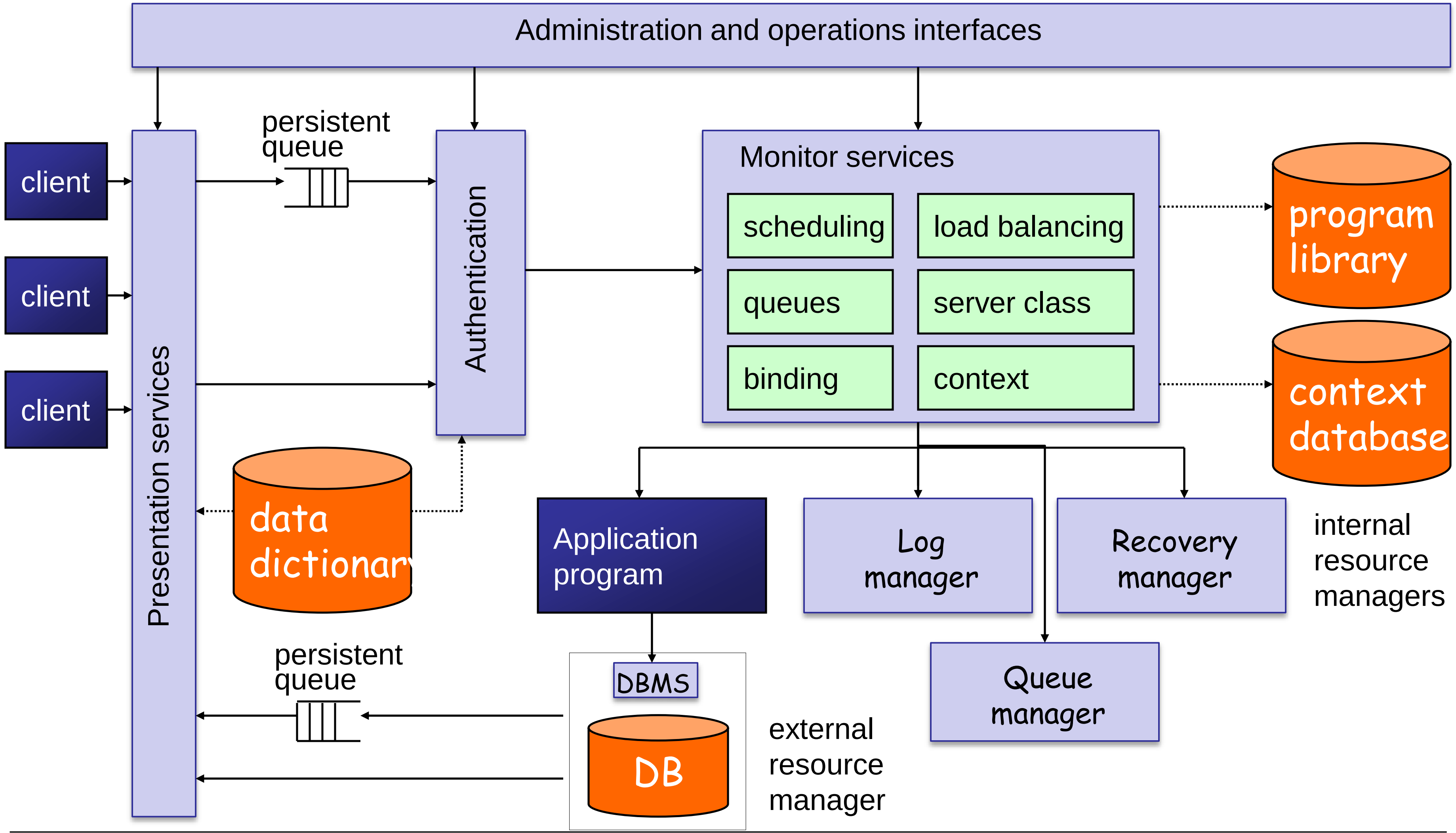
- Transactional RPC: RPC and declarative transactional support
- Transaction manager: runs 2PC and handles recovery operations
- Log manager: records all changes done by transactions so that a consistent version of the system can be reconstructed in case of failures
- Lock manager: a generic mechanism to regulate access to shared data outside the resource managers

Additional services

- Server monitoring and administration: starting, stopping and monitoring servers; load balancing
- Data storage: in the form of a transactional file system
- Transactional queues: for asynchronous interaction between components
- Booting, system recovery, and other administrative chores

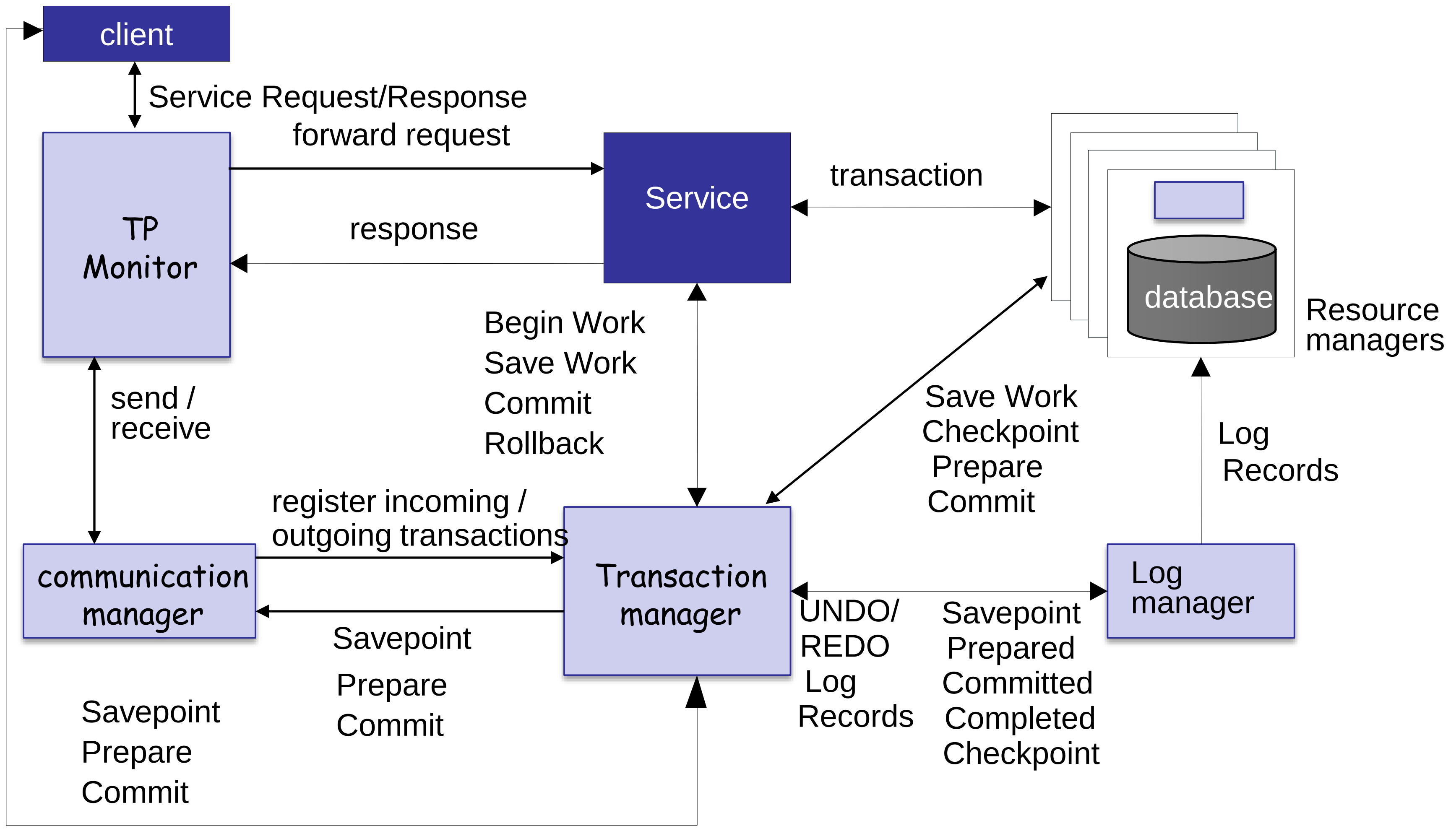
Gustavo Alonso et al, Web Services. Springer Veralg 2003.

TP-Monitor components (generic)



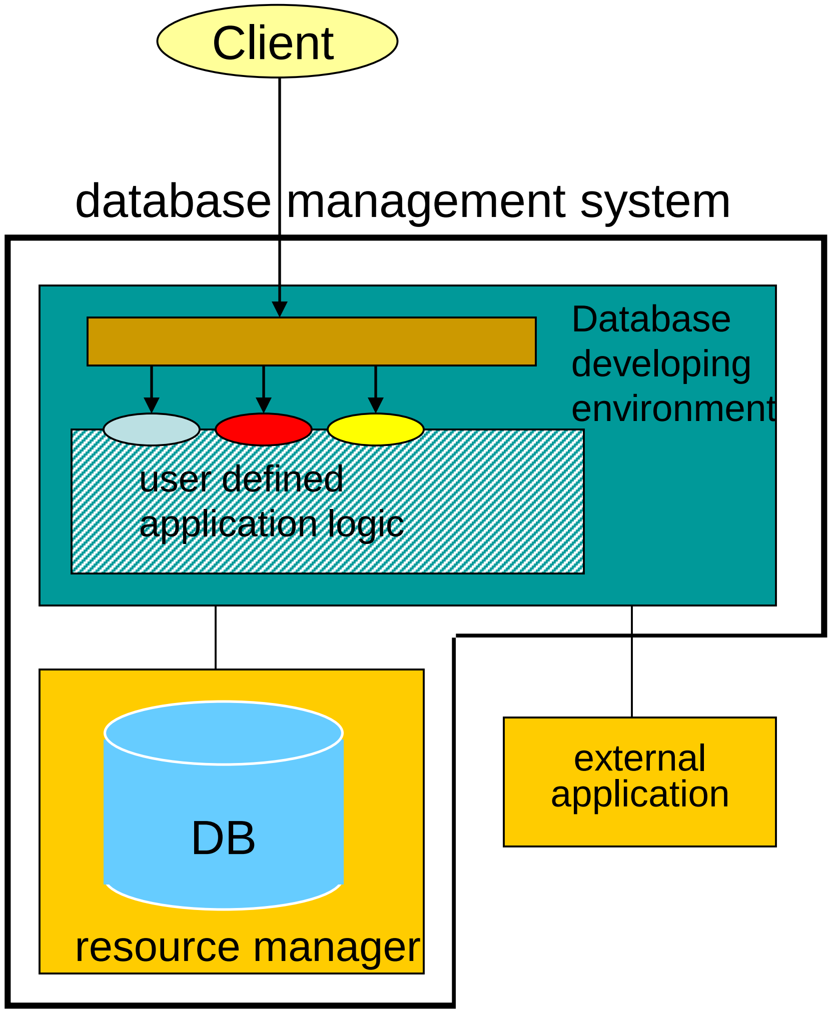
Gray, Reuter. “Transaction Processing” Morgan Kaufmann 1993

Transaction processing components



Gray, Reuter. “Transaction Processing” Morgan Kaufmann 1993

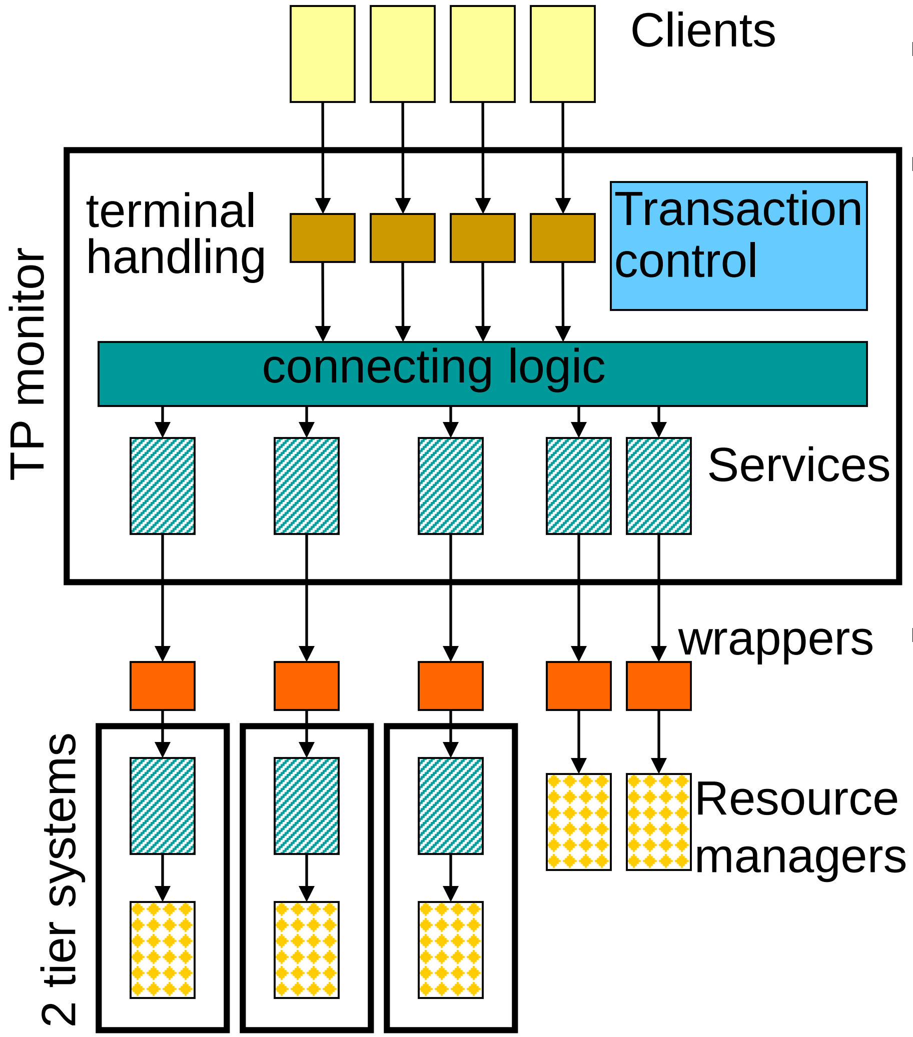
TP-light: databases and the 2 tier approach


Client
database management system
Database developing environment
user defined application logic
DB
resource manager
external application

- Databases are used to manage data
- Simply managing data is not enough
  - One manages data because one has some concrete application logic in mind.
- Why not move the application logic into the database?
  - A 2 tier model proposed with the database providing the tools necessary to implement complex application logic
- These tools include triggers, replication, stored procedures, queuing systems, standard access interfaces (ODBC, JDBC)
- **Scalability Problem: Database Connections!!!**

Gustavo Alonso et al, Web Services. Springer Veralg 2003.

TP-Heavy: 3-tier middleware



- TP-heavy are middleware platforms for developing 3-tier architectures
- A system designer only needs to program the services (which will run within the scope of the TP-Monitor; the services are linked to a number of TP libraries providing the needed functionality), the wrappers, and the clients.
  - TP-Monitors take these components and embed them within the overall system as interconnected components.
- The TP-Monitor provides the infrastructure for the components to work and the tools necessary to build services, wrappers and clients. In some cases, it provides even its own programming language (e.g., Transational-C of Encina).

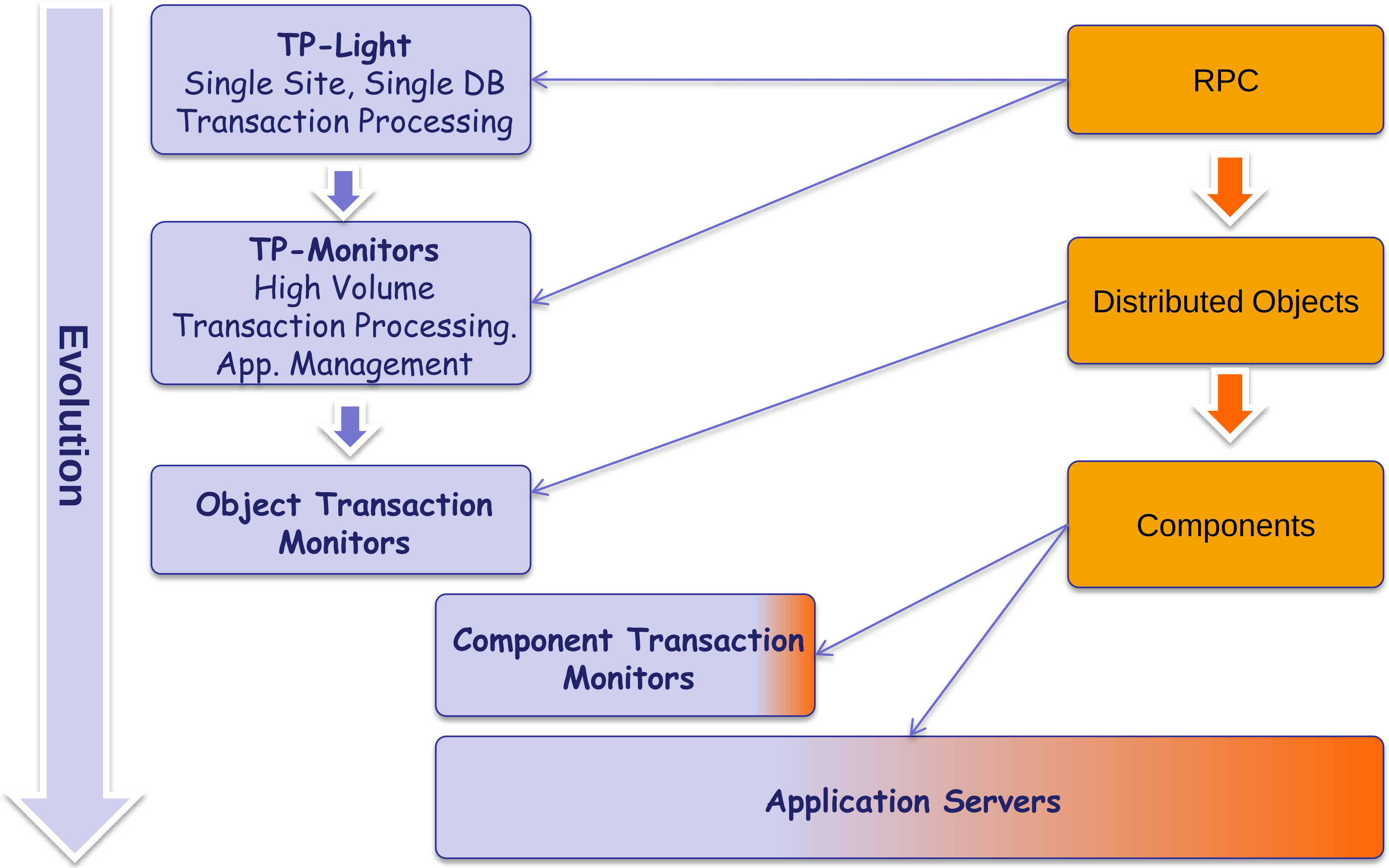
Gustavo Alonso et al, Web Services. Springer Veralg 2003.

TP-Heavy vs. TP-Light = 3 tier vs. 2 tier

- A TP-heavy monitor provides:
  - a full development environment (programming tools, services, libraries, etc.),
  - additional services (persistent queues, communication tools, transactional services, priority scheduling, buffering),
  - support for authentication (of users and access rights to different services),
  - its own solutions for communication, replication, load balancing, storage management ... (most of the functionality of an operating system).
- Its main purpose is to provide an execution environment for resource managers (applications), and do all this with guaranteed reasonable performance (e.g., > 1000 txns. per second).
- Traditional monitors: CICS, Encina, Tuxedo.

- TP-Light is an extension to a database:
  - Originally 1 connection/client, now it is implemented as threads, instead of processes,
  - it is based on stored procedures ("methods" stored in the database that perform a specific set of operations) and triggers,
  - it does not provide a development environment.
- TP-Light became more viable as databases became more sophisticated and provided more services, such as integrating part of the functionality of a TP-Monitor within the database.
- Instead of writing a complex query, the query is implemented as a stored procedure. A client, instead of running the query, invokes the stored procedure.
- Stored procedure languages: Sybase's Transact-SQL, Oracle's PL/SQL.

From TP-Monitors to Application Servers



Summary

- Transaction processing is a key notion in middleware
  - Different applications require different properties → different transaction models exist
  - Systems support and optimize for those
- Transaction models
- TP-Monitors
  - TP-Light, TP-Heavy
  - Layered Architecture
- Are ACID Tx always needed?

Thank You!

TECHNISCHE UNIVERSITÄT DARMSTADT

Questions?

DVS