Enterprise Development Lecture 9

Distributed Applications
Distributed Databases
Distributed Architectures

Objective

- Covered to date:
  - Application components
  - Resource management / scalability running one instance of a glassfish server
- Focus of this lecture: Scaling up and out on multiple instances
  1. Clusters of glassfish instances
  2. Distributed databases
  3. Distributed architectures

1. Distributed architecture with Glassfish . . .

Glassfish application server can be scaled up by creating a cluster of instances across a number of physical nodes on a network.

Introduction

- Lecture 6 covered resource management within a single instances of a glassfish server.
- For large scale availability, glassfish servers can run in a clustered environment, supporting:
  - Multiple instances of service-providing entities (i.e. Java EE applications).
  - Scaling to larger deployments by adding application server instances to clusters in order to accept increasing service loads.
  - Be able to fail over to another server instance so that service is not interrupted if one instance fails.
    - If a process makes changes to the state of a user's session, session state must be preserved across process restarts.

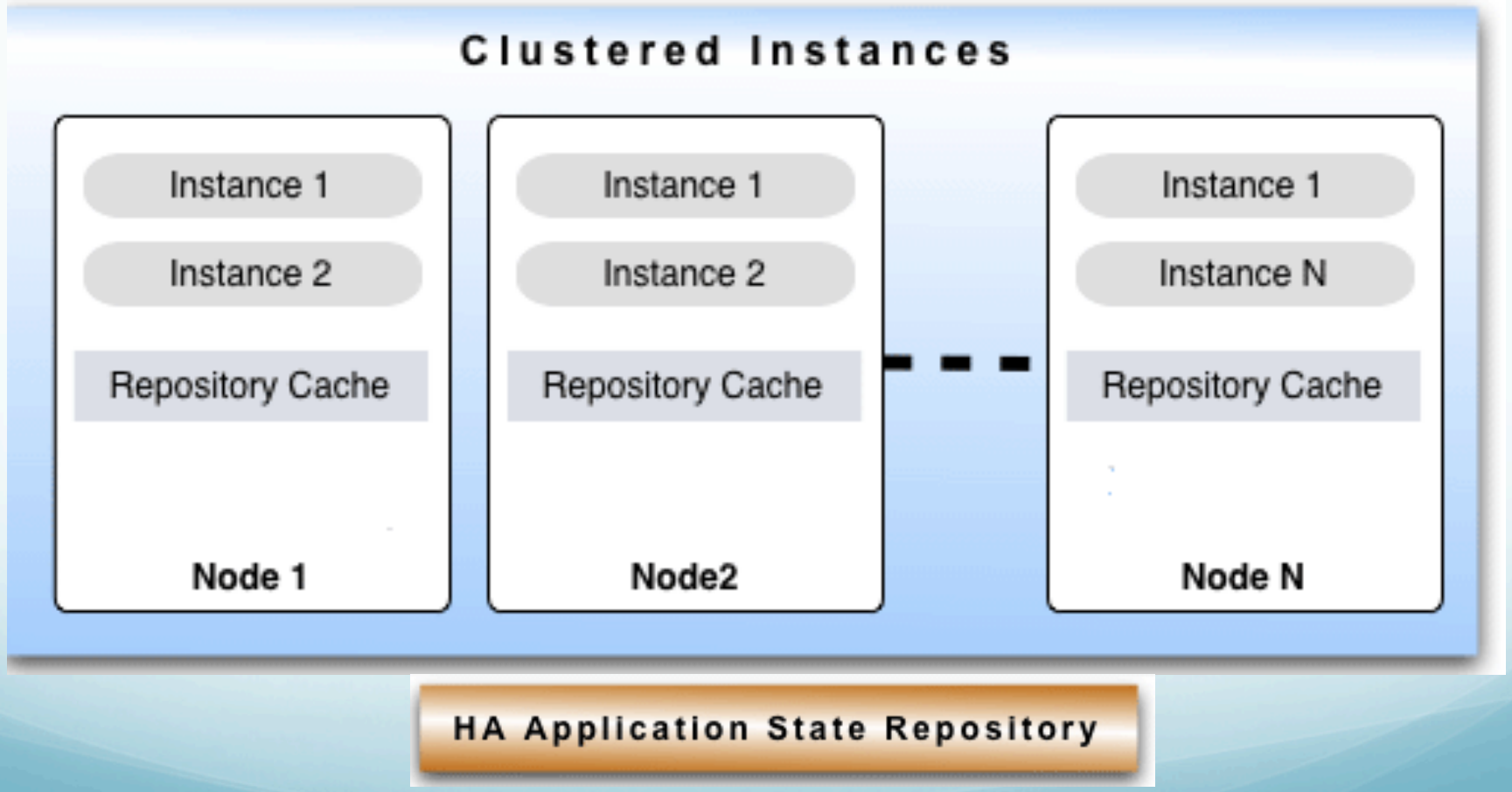
Scalability in Glassfish

GlassFish application server supports the following server-side entities:

- **Server Instance** – An instance of a glassfish server.
- **Node** – A node is a configuration of the GlassFish software that exists on every physical host where one or more server instances run.
- **Cluster** – A cluster is a logical entity that determines the configuration of the server instances that make up the cluster.

Session data is replicated across the cluster in a distributed cache, to provide a High-Availability Application State Repository.

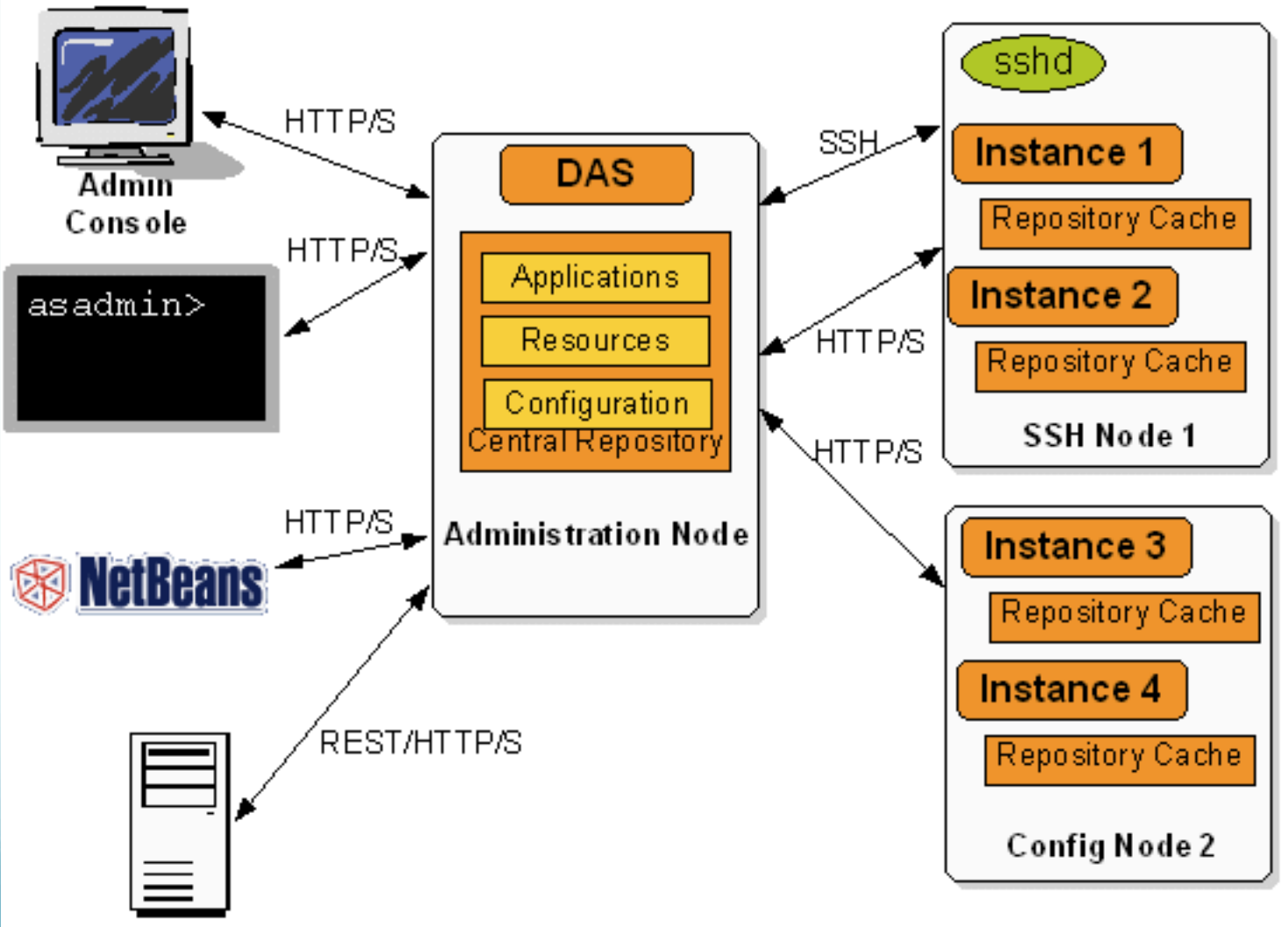
Scalability in Glassfish



Managing clustered instances:

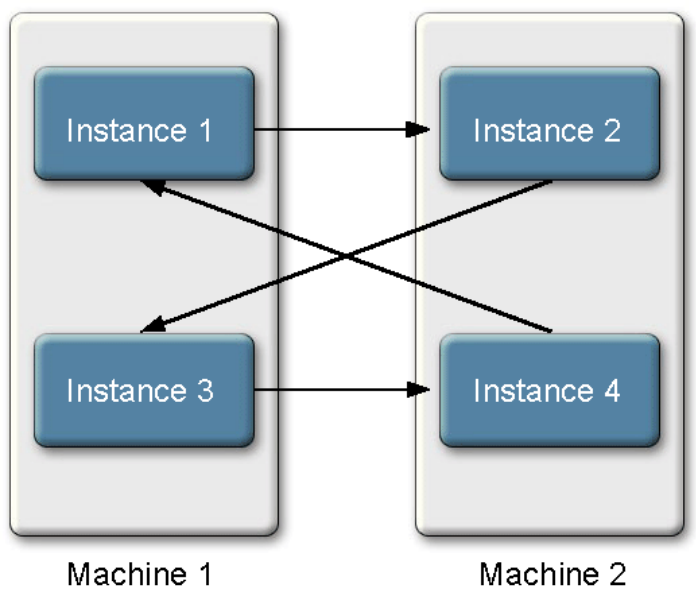
- Clusters and their instances are organized into *administrative domains*, and managed by the Domain Administration Server (**DAS**), which usually runs on a separate node.
- The cluster is administered from the Glassfish admin console of the DAS.

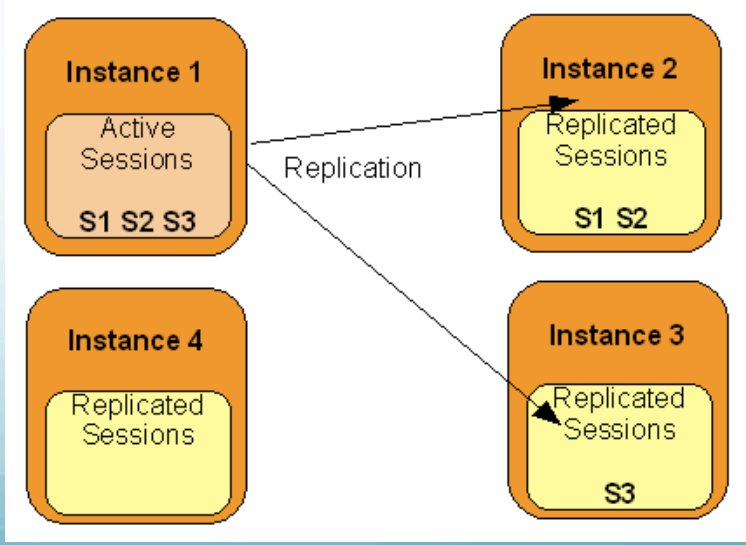
Managing clustered instances:



Reliability and Failover

- Clustered instances are organized so that all memory state data is replicated on other instances.
  - This can be organised in a simple ring topology. Each member in the ring sends memory state data to the next member in the ring, its replica partner, and receives state data from the previous member. As state data is updated in any member, it is replicated around the ring.
  - More recent versions of Glassfish uses a consistent hash algorithm to determine which instance should replicate the individual sessions of another. Sessions from one instance are distributed among the other instances in the cluster.





Group Management Services (GMS)

- Glassfish manages dynamic cluster membership using Group Management Services (GMS). GMS covers:
  - Cluster-wide (one to many) or point-to-point (one to one) messaging
  - Cluster membership change notifications and cluster state
    - Send periodic heartbeats to instances.
  - Recovery in case of failure(s)

Setting up a cluster:

1. Create a cluster: In the Glassfish admin console, select the cluster node, and click new to create a cluster and give it a name.
2. Create a node: Nodes can be set up on the admin console under Nodes.
3. Add instances to a cluster: Once a cluster has been created, instances can be added by selecting the instances link at the of the page, and then select new.

Deploying applications

- Applications are deployed to the cluster (DAS server).
- A load balancer distributes the work amongst cluster instances.

GlassFish clusters scale up by adding more instances to nodes, and scale out by adding more nodes

2. Distributed Databases

Clusters of glassfish instances allow applications to scale, but does not cover the data tier, i.e. databases themselves.

This section looks at distributed databases

Section Objective

- Why have distributed data? How does it arise?
- What are the issues arising from data being distributed.
- Solutions
  - Global transactions manager
  - 2 phase locking
  - 2 phase commit
  - Database clusters

Distributed data

- RDBMS have supported distributed data for a number of years. This can include:
  - **Tables** within a database located on different machines, and managed by different DBMS's
  - **Rows** within a table split across a network
  - **Columns** within a table split across a network
- The data can be fully partitioned (no replication) or partially partitioned (some data is replicated across a number of nodes on a network)

studentID	Name	Course	Registration date	CAO points . .
B00000123	JK	BN402	11th Sept 2009	350
B00002345	DM	BN402	11th Sept 2009	240
B00034560	GP	BN402	11th Sept 2009	420

Why is data distributed

1. Data can be distributed by design: top down approach
   - Design the database as a single ERD, and split tables/data ensuring that data is located close to the users that access it most.
   - Not common practice for online transaction processing (OLTP).

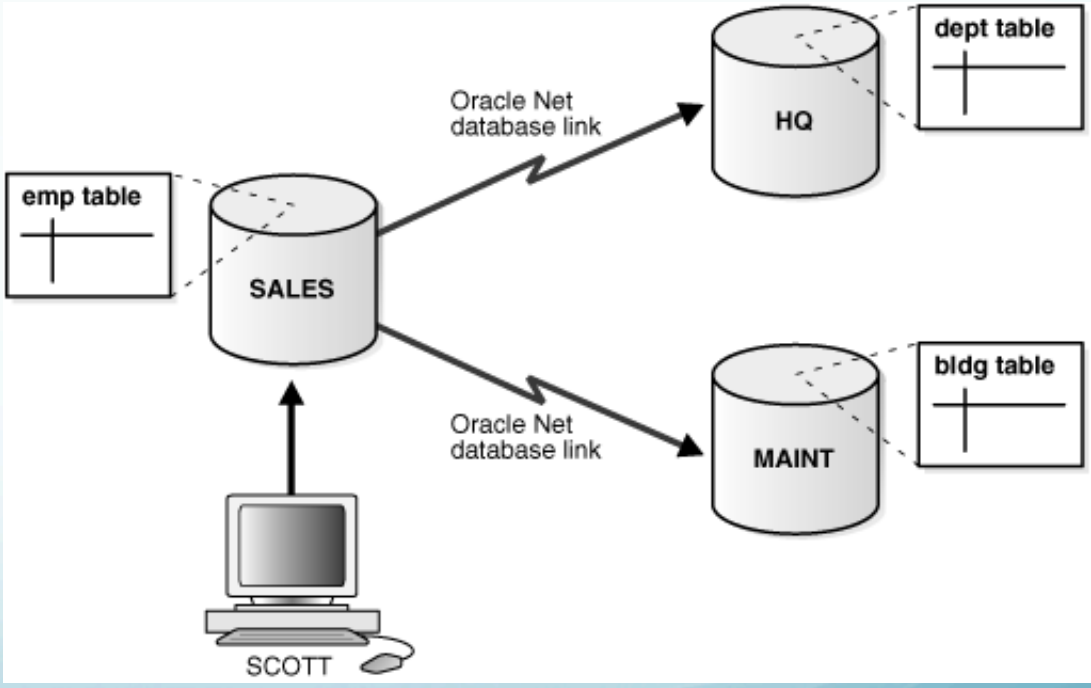
2. Data can be distributed automatically by a DBMS operating in cluster mode.
   - Tables rows are split across nodes in the cluster
   - This is **HOMOGENEOUS** distributed data

Why is data distributed?

3. Alternatively, distributed data can evolve from merging existing data sources: bottom up approach

• e.g. a new application requires access to a number of existing data sources dispersed across a global enterprise.

•This is **HETERGENEOUS** distributed data



Issues arising

- Maintaining **ACID** properties poses a challenge when **updating** data in a distributed database.
- We will look at these challenges next. . .

ACID - recap

- **Atomicity.** A transaction is an atomic unit of processing. It is either performed entirely or not at all.
- **Consistency.** Correct execution of a transaction must take the database from one consistent state to another, without necessarily preserving consistency at all intermediate points.
- **Isolation.** Transactions are isolated from one another. A transaction should not make its updates visible to another until it has committed.
- **Durability**. Once a transaction has committed, the changes it has made are never lost because of subsequent failure.

Commit & Rollback

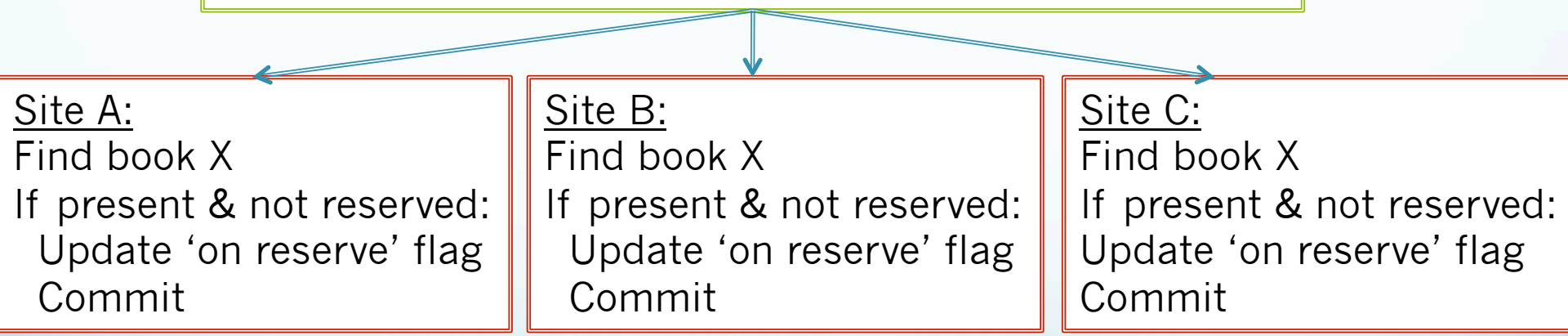
Locking

Redundancy & Recovery

Example – searching for a book

Global query: search for two copies of book X, and place them on reserve.
Translate to:

Search site A for book X, if present place one reserve.
Search site B for book X, if present place one reserve.
While still looking . ..
    Search site C for book X, if present place one reserve
    etc.



Site A:
Find book X
If present & not reserved:
  Update 'on reserve' flag
  Commit

Site B:
Find book X
If present & not reserved:
  Update 'on reserve' flag
  Commit

Site C:
Find book X
If present & not reserved:
Update 'on reserve' flag
Commit

Serialisable schedule

- A **schedule** of tasks is the list of reads and writes a particular DBMS is executing.
- The serial schedule (each transaction is completed before the next transaction begins) maintains acid properties, but is not efficient as transactions can not operate concurrently.
- Interleaved schedules are preferable, but only if their effect on the database is the same as if the transactions ran serially

Serial schedule

DBMS Y:
T1:Read Y in table 3
T1:Update Y in table 3
T2: Read Y in table 3
T2: Update Y in table 3

Interleaved schedule

DBMS Y:
T2:Read Y in table 3
T1:Read Y in table 3
T1:Update Y in table 3
T2:Update Y in table 3

Serialisable schedule

An interleaved schedule is said to be serialisable, if it is equivalent to a serial schedule.

Two schedules are equivalent **if and only if**

1. Each read operation reads the same value in each schedule
2. The final database state is the same for both schedules.

The following slides focus on ensuring global transactions running over a distributed database are serialisable.

Isolation & locking strategies

Isolation: A transaction should not make its updates visible to another until it has committed.

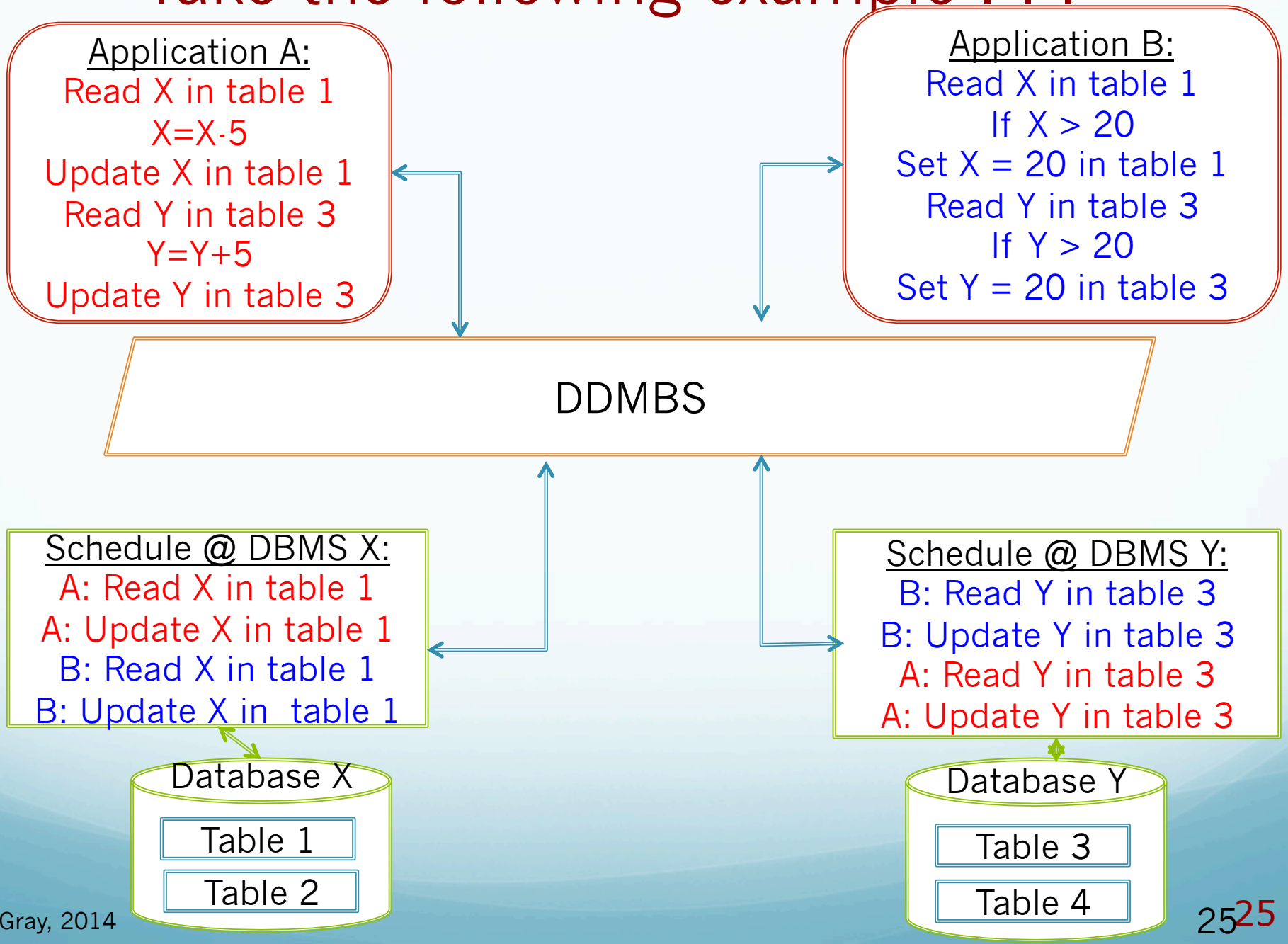
- At a local level, isolation is achieved by using locking – either pessimistic or optimistic.
  - Locking would prevent the two schedules of database operations shown below

Schedule @ DBMS Y:
T1: Read Y in table 3
T2: Read Y in table 3
T1: Update Y in table 3

Schedule @ DBMS Y:
T1: Read Y intable 3
T2: Update Y in table 3
T1: Update Y in table 3

- However locking is not sufficient to guarantee the isolation of distributed transactions, the following slides will explain why . . .

Take the following example . . .



25

Example contd.

- Suppose the values of X and Y are as follows before either transaction runs:
  - X=21 Y=18

What is the final value of X given the schedule of transactions at DMBS X?

What is the value of Y given the schedule of transactions at DMBS Y?

If the transactions ran in isolation (application B couldn't start until application A completed), would the values of X & Y be different?

Isolation property has been violated

Example continued . . .

- The schedule of sub-transactions (part of a global/distributed transactions) at each local site is serial, so there is no conflict at local level with the sub-transactions running on each site.
- However when viewed at a global level, the effect of the two transactions is NOT the same as if each transactions executed serially:
  - At DBMS X, transaction B is seeing updates made by application A before the entire transaction is complete (the part of the transaction send to DBMS Y has yet to run).
  - The same applies to transaction A at site B.

2 phase locking

- Locking will guarantee the serialisability of transactions at local level.
- However as we saw from the earlier examples, locking at local DBMS sites does not guarantee that schedules of global distributed transaction is serialisable.
- A solution is to delegate lock requests for the entire transaction to a Global Transaction Manager, and use 2 Phase Locking.

2 phase locking

No lock can be released until all locks that the transaction will need have been acquired.

- With 2 phase locking, locking is done in two phases:
  - Phase 1: all locks are required
  - Phase 2: all locks are released
- Phase 2 can't start until phase 1 is complete, hence all locks must be acquired before any are released.
- Recap on locks: transactions can be granted two types of locks:
  - **Read lock**: a number of transactions can hold read locks on a data item at the same time.
  - **Write lock**: only one transaction can have a write lock, and it will only be granted if there are no read locks on the transaction

2 phase locking in action

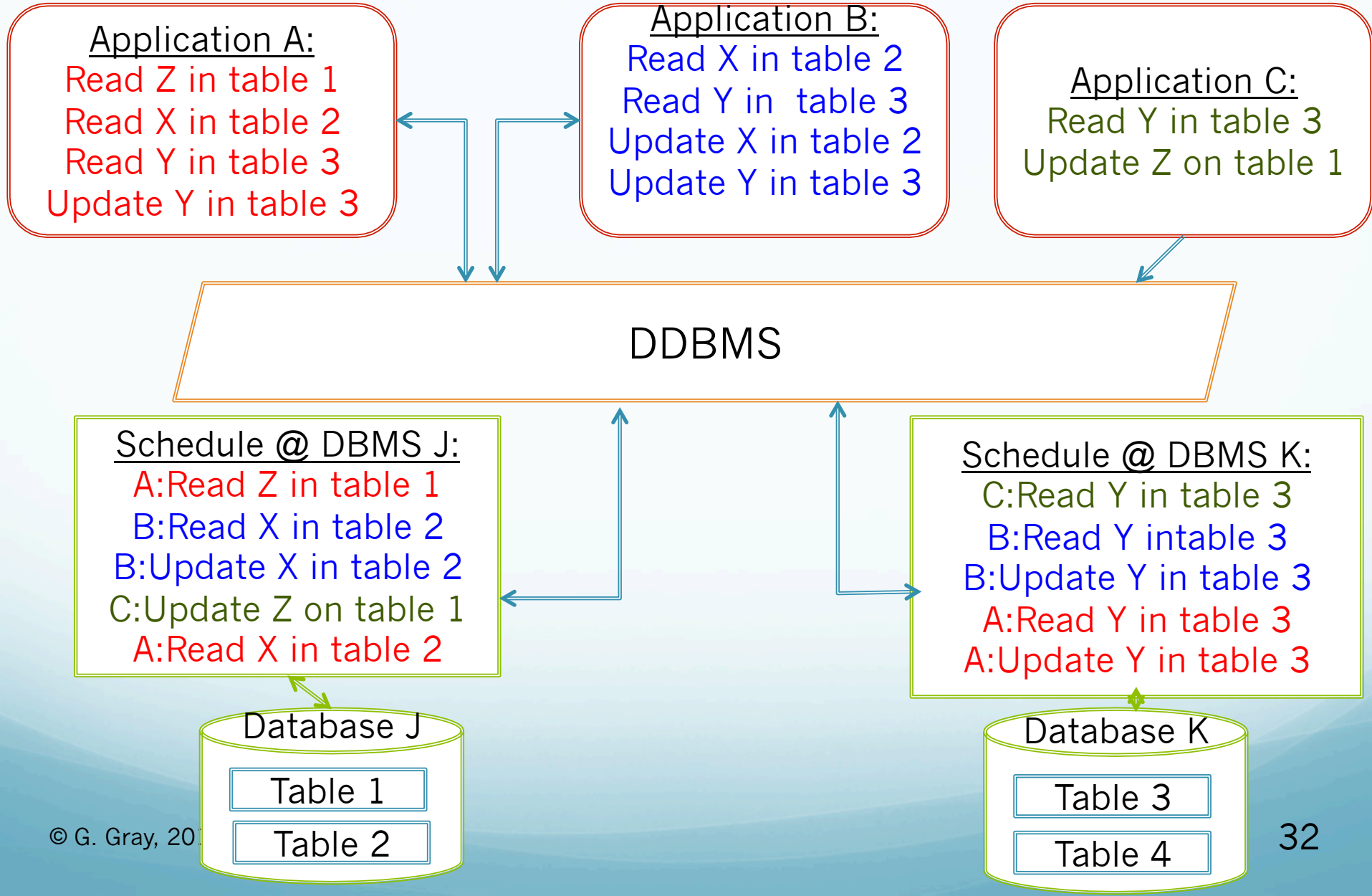
from slide 15

Transaction A	Transaction B	locks
Read X in table 1		Acquire read lock on X
Write X in table 1		Acquire write lock on X
	Read X in table 1	Request for read lock denied – wait for transaction A to complete
	. . On hold . . .	
Read Y in table 3		Acquire read lock on Y
Write Y in table 3		Acquire write lock on Y
Commit		Release all locks
	Transaction B can now execute	
	Read X in table 1	Acquire read lock on X
	. . . .	

Atomicity & Durability

Commit & Rollback

Take the following example . . .



The global transaction schedule :

Transaction A	Transaction B	Transaction C	locks
Read Z in table 1		Read Y in table 3	Acquire read lock on Z and Y
	Read X in table 2		Acquire read lock on X
	Update X in table 2		Acquire write lock on X
	Transaction B @ DBMS J ready to commit	Write Z in table 1	Request for write lock denied.
Read X in table 1		Waiting on transaction A . . .	Request for read lock denied.
Waiting on transaction B	Read Y in table 3		Request for read lock denied.
	Waiting on transaction C . . .		
	DBMS decides to roll back transaction B		.

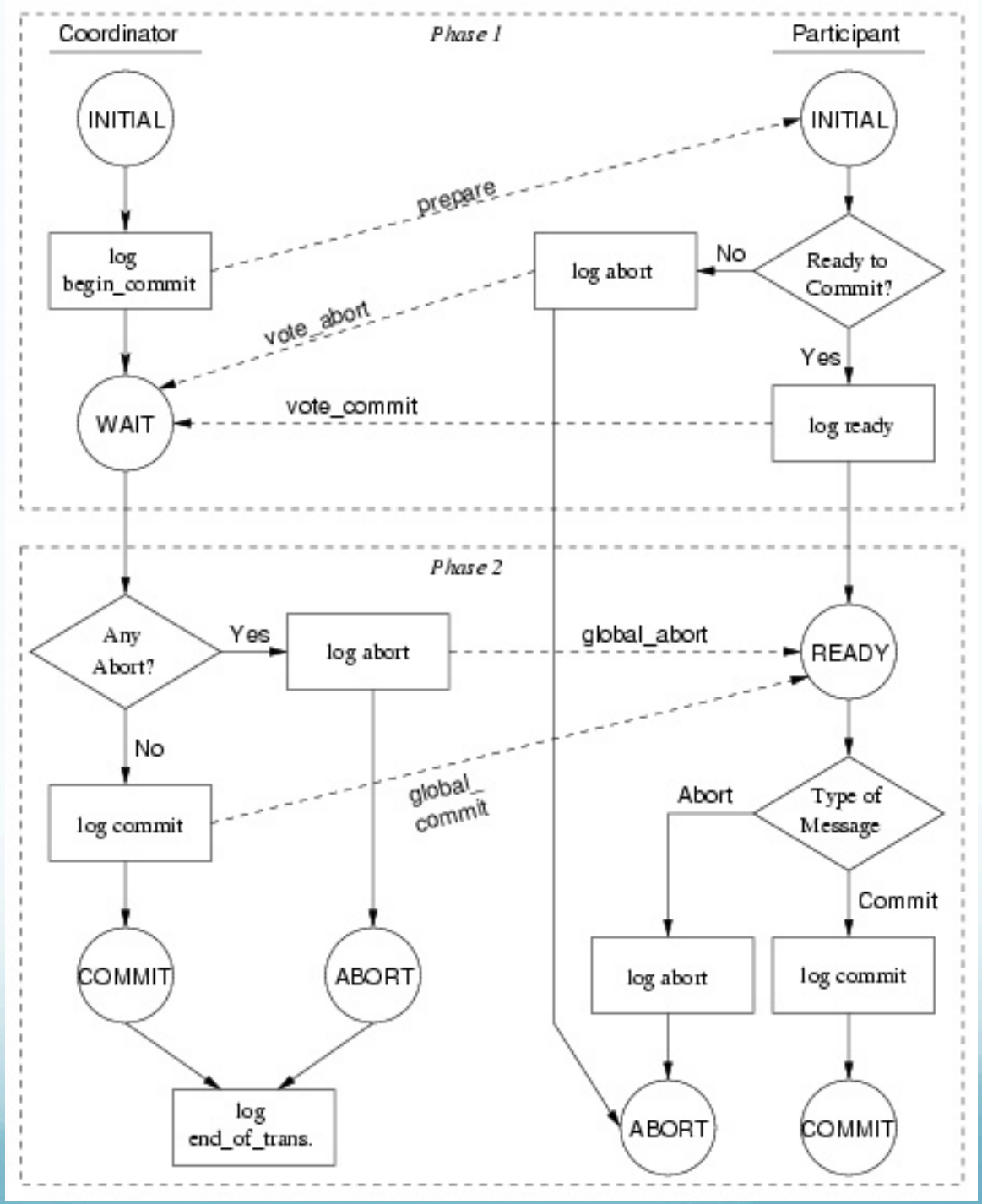
Commit and Rollback

Two problems arise with this example:

1. How does each local DBMS know deadlock has occurred?
2. Assuming one DBMS does elect to rollback a transaction, how do other sites know about the rollback?
   - Suppose DBMS Y elects transaction B for roll back.
   - As far as DBMS X is aware, transaction B's local operations completed normally and it's ready to commit.

Solution: 2-phase commit

Full two phase commit



2 phase commit

- A global transaction is initiated at some site by the Global Transaction Manager at that site (the coordinator).
- The coordinator consults the Data Dictionary to determine the location of the required data, and divides the global transaction into a series of subtransactions for each participant site.
- These subtransactions are sent to the participant sites, where they are processed by the Local Transaction Manager at each site.
- When the subtransaction has finished processing, it cannot yet issue a commit, since it represents only a part of the global transaction. The participant sends a message to the coordinator indicating that it is ready to commit (or abort) the subtransaction. It is, however, free to rollback.

Voting phase

2 phase commit

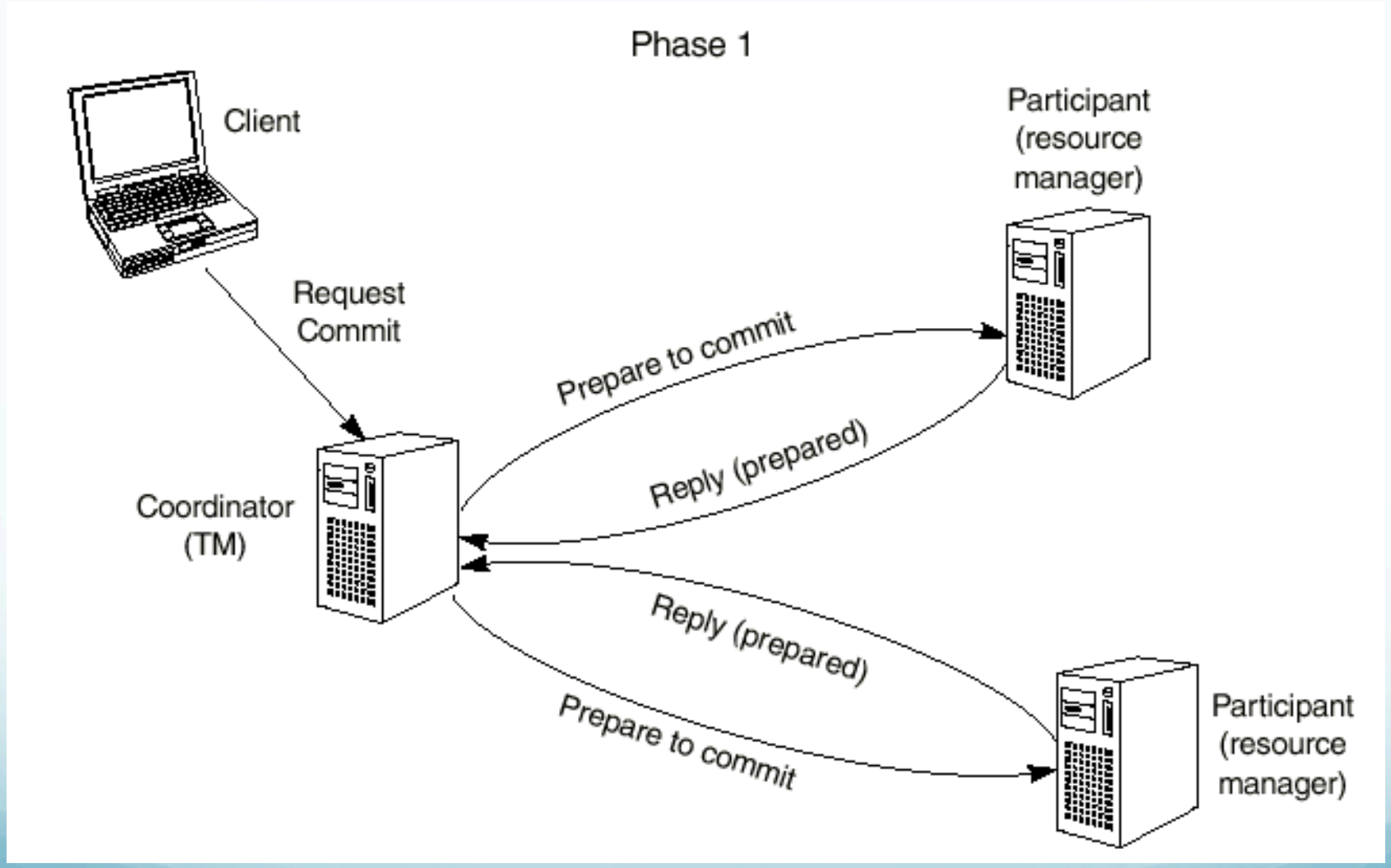
Decision phase

- The coordinator gathers all the responses from all participants and decides whether to commit or abort the global transaction. It can only commit if each of the participants has voted to commit. If one or more vote to abort (or fail to respond within a certain time), the coordinator instructs all participants to abort
- The decision is sent to all participants, and they take the appropriate action to commit or abort, depending on the decision of the coordinator.

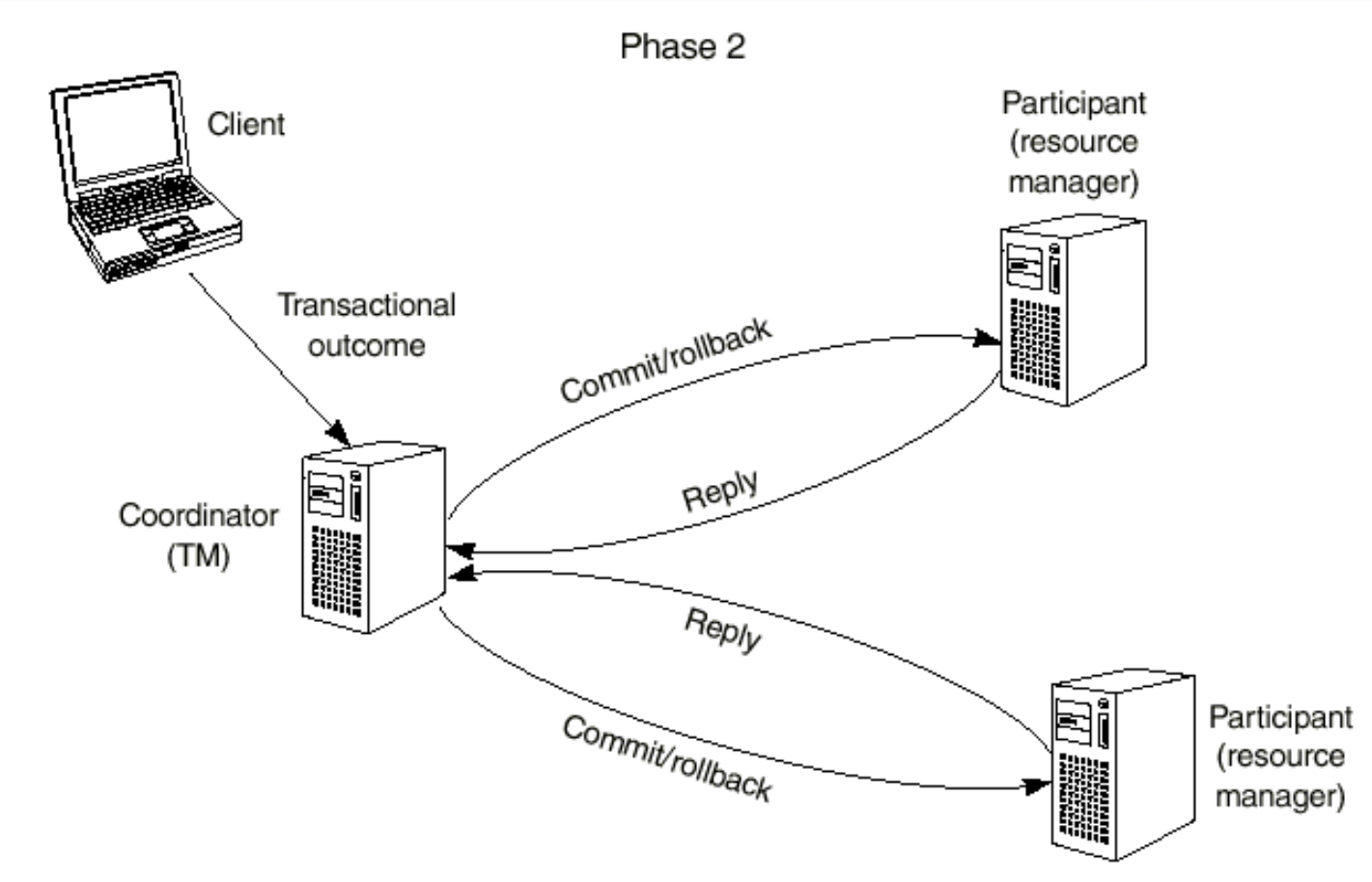
2 phase commit

- The protocol is called Two-Phase Commit (2PC) because of its two phases:
  1. a **voting phase**, in which the coordinator asks the subtransactions whether or not they are prepared to commit the transaction.
  2. and a **decision phase**, in which the coordinator receives responses and makes a decision.

Voting phase



Decision phase



Past exam question:

Explain the term **heterogeneous distributed data.** Why, in practice, are distributed databases often heterogeneous?

5 marks

Discuss the role of a **global transaction manager** in guaranteeing transaction atomicity in a distributed database.

8 marks

Redundancy & Recovery

Clustering

- Databases can be installed in cluster mode.
  - Data is replicated across data nodes in a cluster.
- To get full redundancy, it should be run over at least three servers, one running the management node, and two storing data nodes.

Terminology

- Data node: Stores data for a portion of the database. Each data node should be located on a separate computer.
- Node group: Data nodes are grouped into node groups. The number of data nodes in a node group depends on the number of replicas needed:
  - 1 primary copy, 0 replicas – each data node is itself a node group
  - 1 primary copy, 1 replica: two data nodes per node group
  - 1 primary copy, 2 replica's: three data nodes per node group
    - Etc
- Partition: master copy of the data
- Replica: slave copy of the data

Example:

1 replica of each partition; 2 nodes per cluster

Each node group in a cluster must have the same number of data nodes.

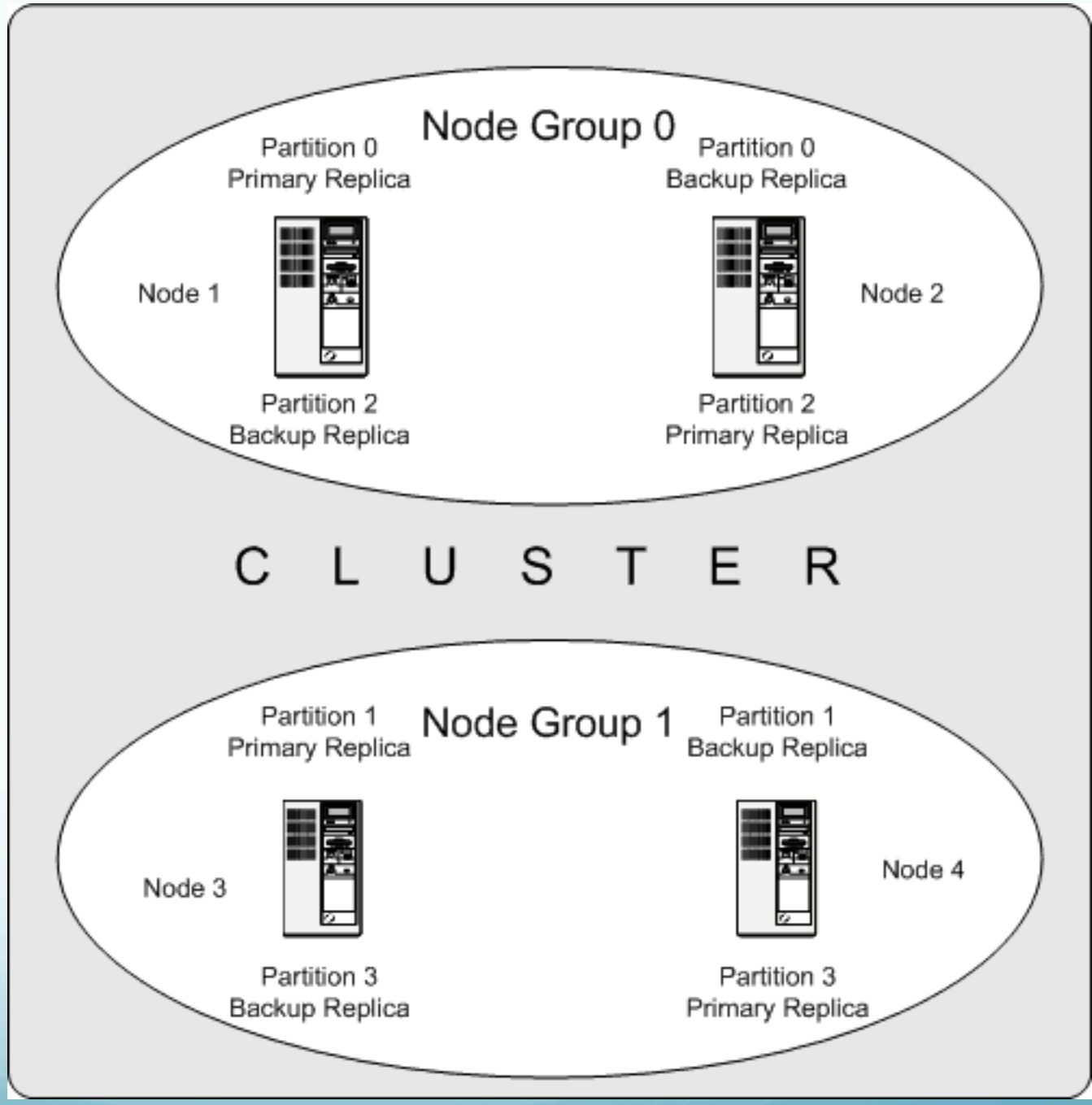
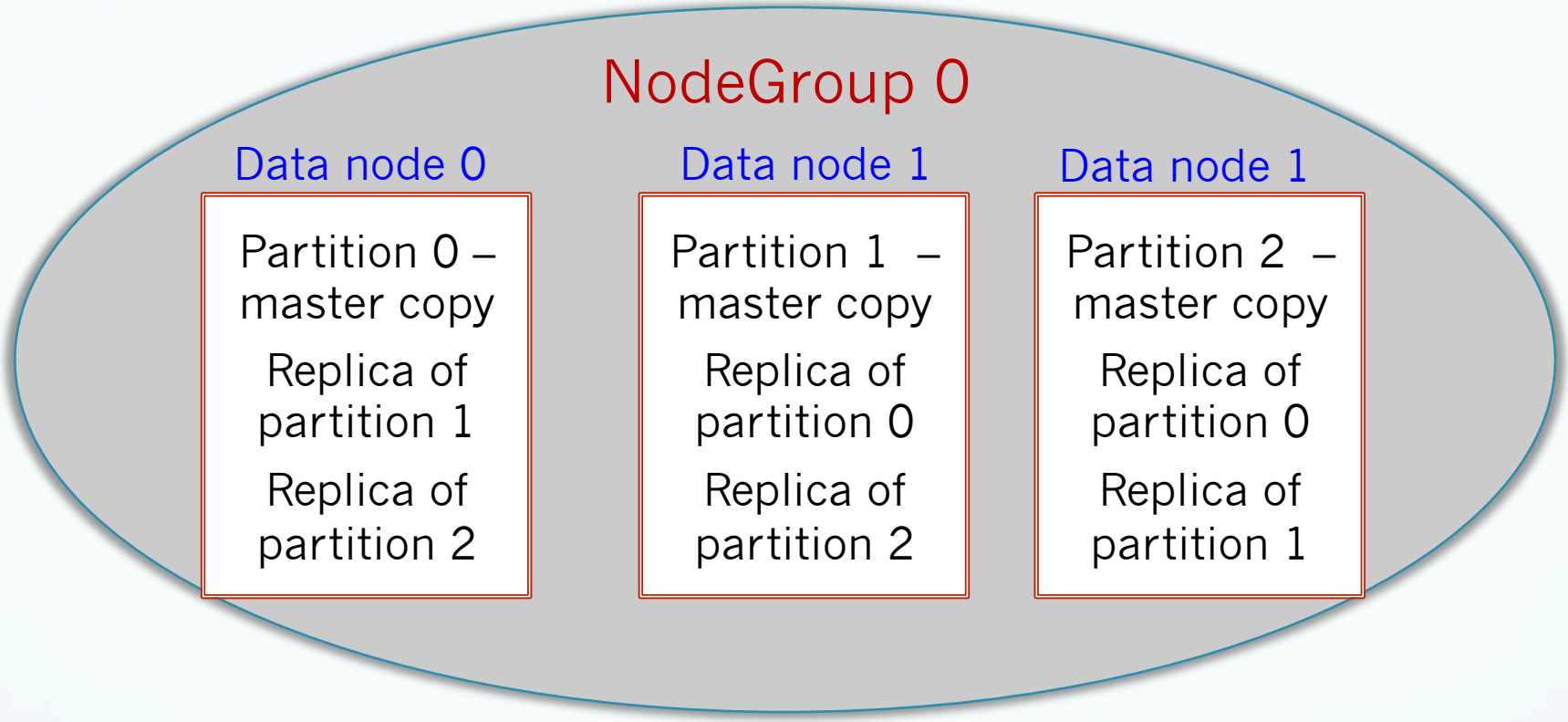


Diagram from this url



- Example: 2 replicas of each partition, so 3 data nodes per node group.
- There is no default value for **NoOfReplicas**; the recommended value is 2 for most common usage scenarios - one master copy & one slave copy. The max value in MySQL is 4.

Partitions

- Partitions are created automatically by the managing node (not shown in diagrams) depending on the number of data nodes available, and the number of replicas required.
- Database tables are partitioned horizontally.

E#	Name	Address	Salary	Title	D#
E001	Smith	Dublin	15000	Manager	D1
E002	Jones	Bombay	25000	Analyst	D1
E003	Doody	New York	5000	Analyst	D2
E004	Doe	Bombay	9000	Salesman	D7
E005	Elmasri	Dublin	28000	Secretary	D4

E#	Name	Address	Salary	Title	D#
E001	Smith	Dublin	15000	Manager	D1
E002	Jones	Bombay	25000	Analyst	D1
E003	Doody	New York	5000	Analyst	D2

E#	Name	Address	Salary	Title	D#
E004	Doe	Bombay	9000	Salesman	D7
E005	Elmasri	Dublin	28000	Secretary	D4

Node Failure

- Each node sends periodic 'heartbeat' checks to the other nodes in their cluster.
- If a node fails to respond, the sender of the 'heartbeat' check informs all other nodes in the cluster of the status of the failed node, and the failed node is removed from the cluster.
- Other nodes adjust accordingly, switching their requests to a replica copy of the data.
- When the failed node restarts, it takes the most recent copy of the data, and rejoins the cluster.

Failover

- As long as each node group participating in the cluster has at least one node operating, the cluster has a complete copy of all data and remains viable.
- However, if all nodes in a node group fail the cluster has lost an entire partition and so can no longer provide access to a complete set of all cluster data.

3. Distributed architectures

Clustering application servers and databases allows for scalability of a single system.

The final section of the lecture focuses on how to integrate different systems.

Integration architectures

In a well-designed building, the electrics and plumbing usually keep working no matter how many appliances are switched on (multi-user). Such a building is also capable of extension without having to tear up the blueprints and start again (scalable). Why? Because it has good architectural design.

The same applies to software systems. Software architecture is the backbone of any complex computer system. The architecture encompasses all of the software elements, the relationships between the elements and the user interfaces to those elements. The performance and reliability of a software system are highly dependent upon the software architecture.

Well-designed software architecture can be extended with relative ease to accommodate new applications without requiring extensive infrastructure development.

Ref: http://hubpages.com/hub/Integration-Architecture-Explained

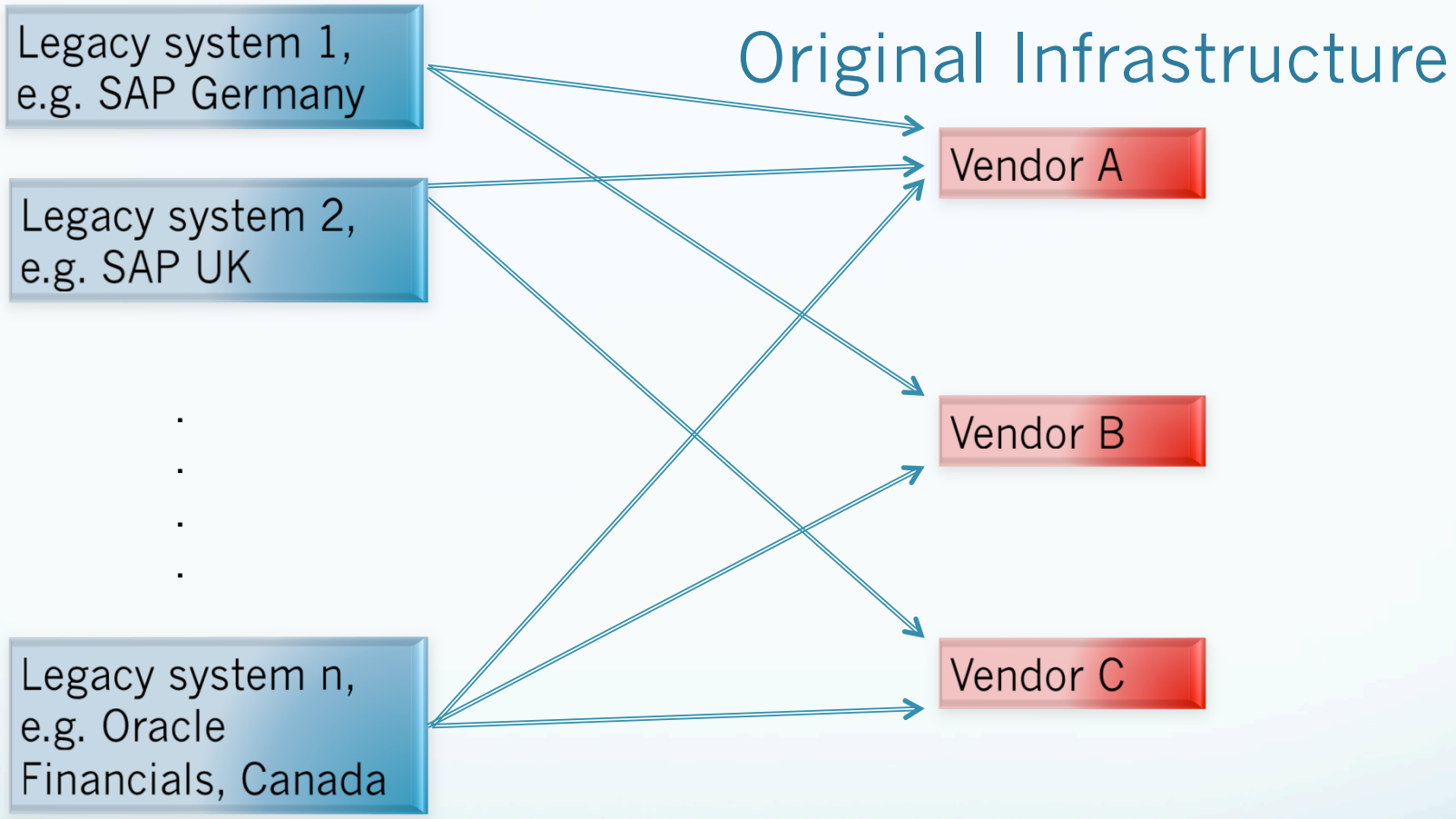
Case study: UK-based transportation company

Royal Wallace, a UK-based transportation company, is a global leader in the rail equipment and servicing industry. Its wide-range of products includes passenger rail vehicles and total transit systems. It also manufactures locomotives, freight cars, bogies, and provides rail-control solutions.

Because of the structure of its business, the Royal Wallace company has a presence in several countries across Europe and in North America, including the USA and Canada.

Each of these country-specific sites has its own dedicated legacy system such as SAP-based systems in UK and Germany, Oracle financials in Canada, and so on.

Case study: UK-based transportation company



SAP AG is a multinational software development and consulting corporation, which provides enterprise software applications and support to businesses of all sizes globally.

Case study: UK-based transportation company

In the original architecture, legacy systems acted as independent data silos with no integration or data-sharing between applications.

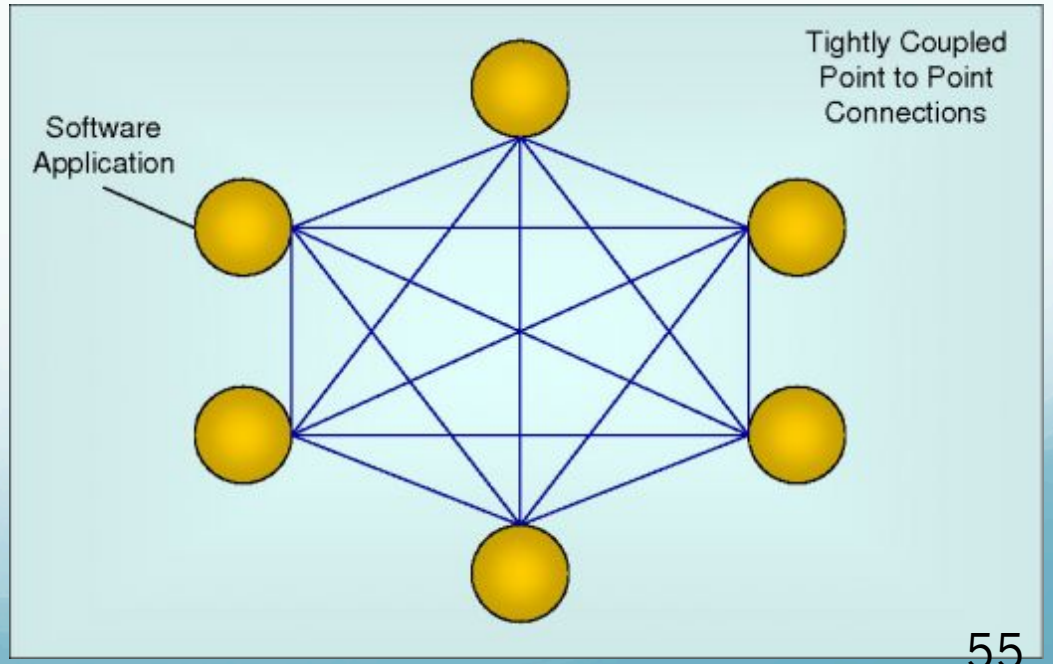
For example, if two different systems are using the same vendor (say, Vendor A), this vendor's information is replicated at both the legacy systems, and purchase orders are sent independently by each site. If the vendor was to notify the company of an address change, chances are the change would be made in the local system, but not replicated in remote sites.

Royal Wallace understands that to improve efficiency, and leverage the benefits in technology advances, it needs to put an integration solution in place.

Ref: http://www.developer.com/print.php/3509766

1. POINT-TO-POINT INTEGRATION ARCHITECTURE

- The original architecture used to support systems integration was called Point-to-Point. It derives its name from the direct, tightly bound connections that are made between applications and is the simplest of the integration architectures.



POINT-TO-POINT INTEGRATION ARCHITECTURE

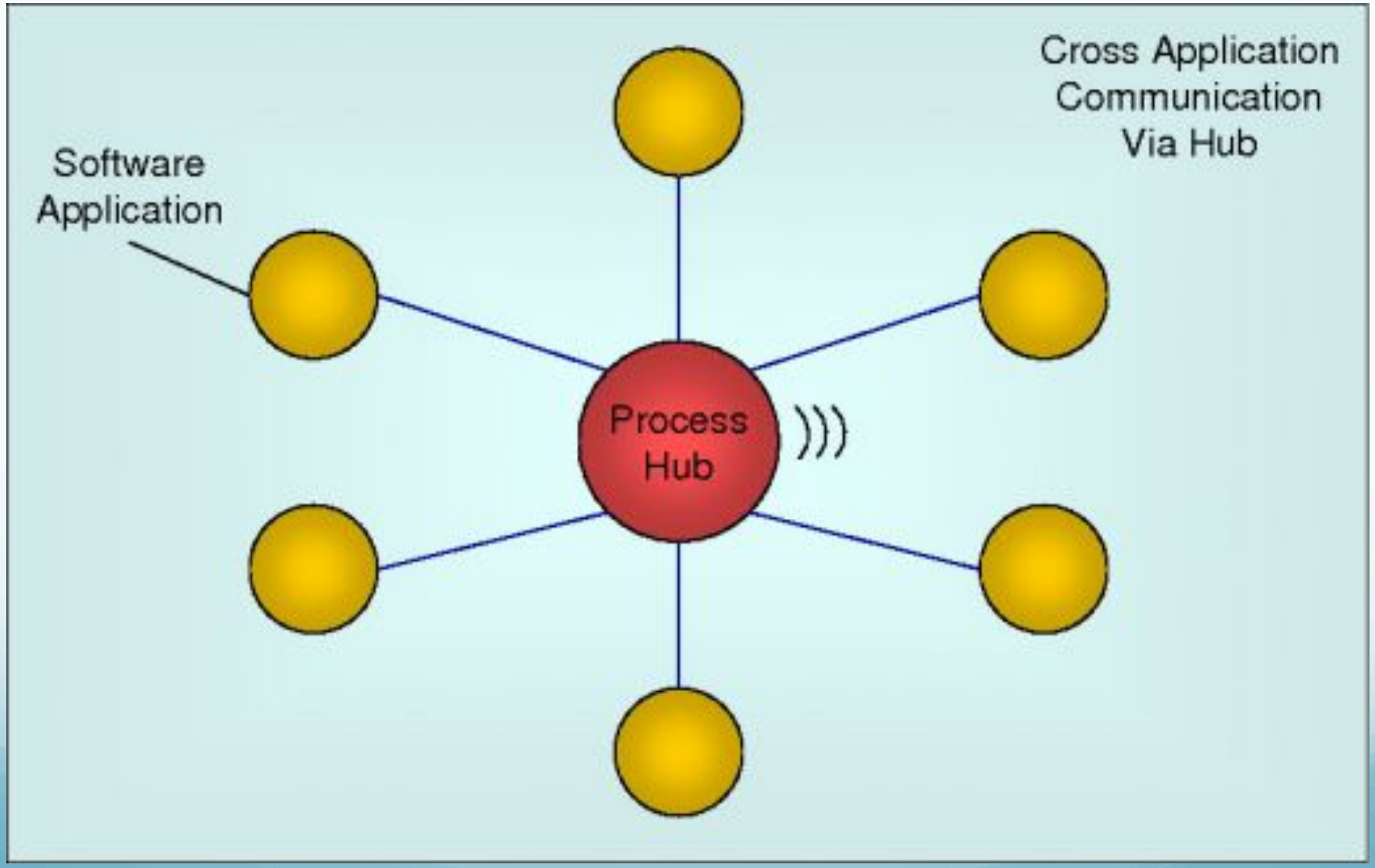
- ✓ The tight coupling in Point-to-Point solutions are fast and efficient. This efficiency is one reason why Point-to-Point solutions are still in use.

- ✖ The down side of Point-to-Point is that, while each individual connection might be relatively straightforward, as the number of applications increases so too does the overall complexity of the environment. This can result in:
  - ✖ high maintenance costs and
  - ✖ a lack of flexibility when it becomes necessary to make changes.

2. HUB AND SPOKE INTEGRATION ARCHITECTURE

- A solution to this lack of scalability was to implement a hub to handle systems integration.
- Acting as a central point of control, the hub dealt with all message processing including routing, splitting and combining of messages, mapping, and so on.
- Hub and Spoke implementations decouple the sending and receiving applications.
- Unlike Point-to-Point, the applications on either side of the hub can be modified independently of each other. Since applications no longer need to perform data mapping, centralized definition and control of business processes could be easily achieved for the first time.

HUB AND SPOKE INTEGRATION ARCHITECTURE



HUB AND SPOKE INTEGRATION ARCHITECTURE

- ✓ The integration environment is less complex as all connections are to and from the hub.
- ✓ Hub and Spoke is the preferred architecture for achieving an easily controlled and managed environment in a medium sized integration project.
- ✖ The initial setup of two-way communications can be challenging. The hub has to coordinate messages flowing between applications.
- ✖ For large implementations, the central hub can become a bottleneck and suffer from poor performance.

Service Oriented Architecture (SOA)

- Next semesters Web Services module will cover SOA.
  - Each component offers its capabilities as a service on the network, using Web Services technologies
  - Its a more loosely coupled solution than Point to Point or Hub and Spoke. Allowing for greater scalability.
  - Works with a variety of implementation frameworks, not just Java EE.