



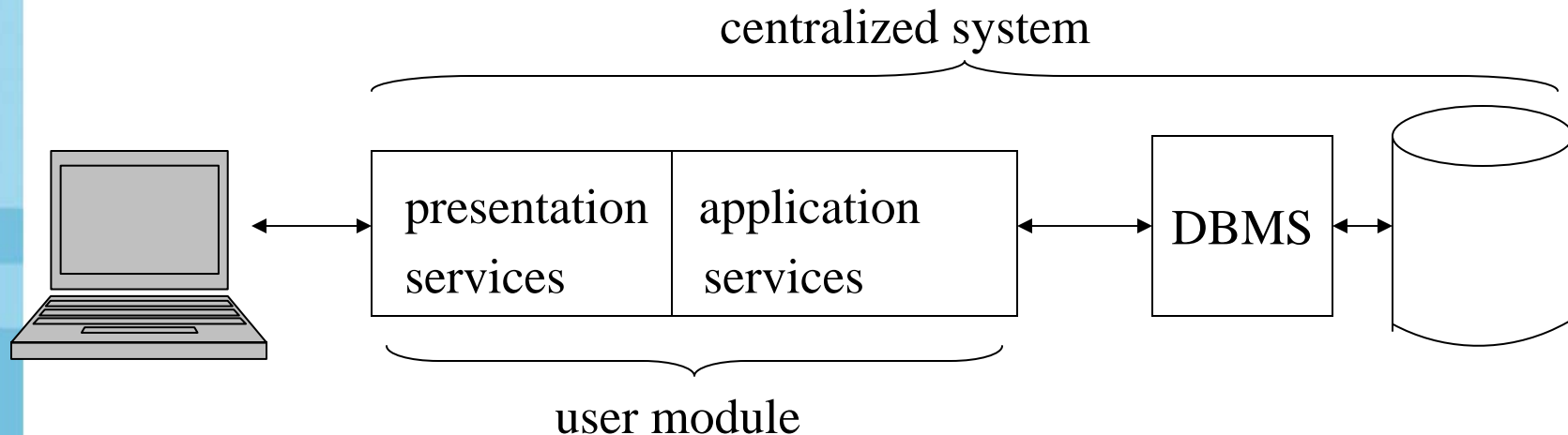
Transaction Processing Systems



Evolution of Transaction Processing Systems

- The basic components of a transaction processing system can be found in single user systems.
- The evolution of these systems provides a convenient framework for introducing their various features.

Single-User System



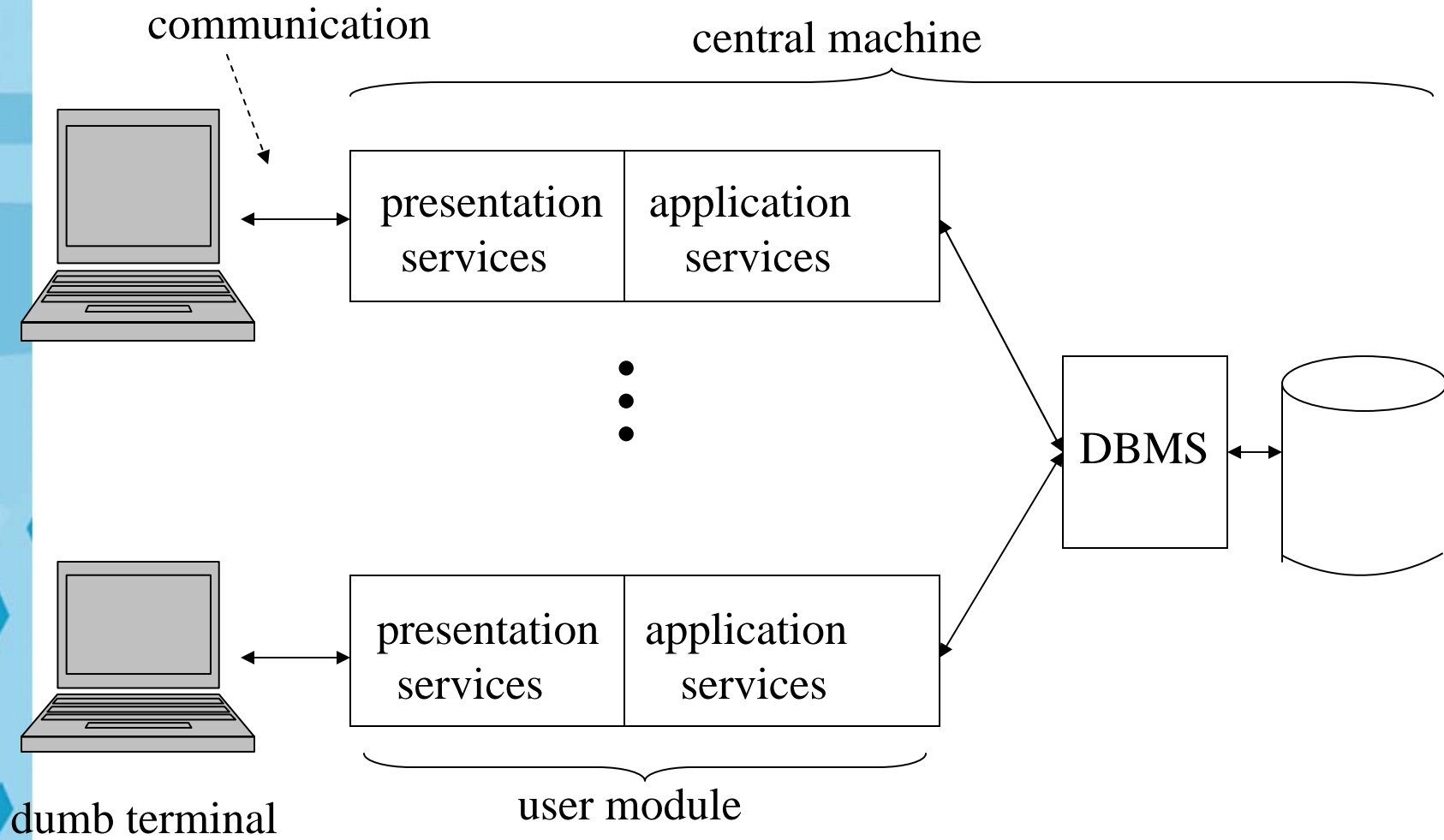
- *Presentation Services* - displays forms, handles flow of information to/from screen
- *Application Services* - implements user request, interacts with DBMS
- ACID properties automatic (isolation is trivial) or not required (this is not really an enterprise)



Centralized Multi-User System

- Dumb terminals connected to mainframe
 - Application and presentation services on mainframe
- ACID properties required
 - Isolation: DBMS sees an interleaved schedule
 - Atomicity and durability: system supports a major enterprise
- Transaction abstraction, implemented by DBMS, provides ACID properties

Centralized Multi-User System

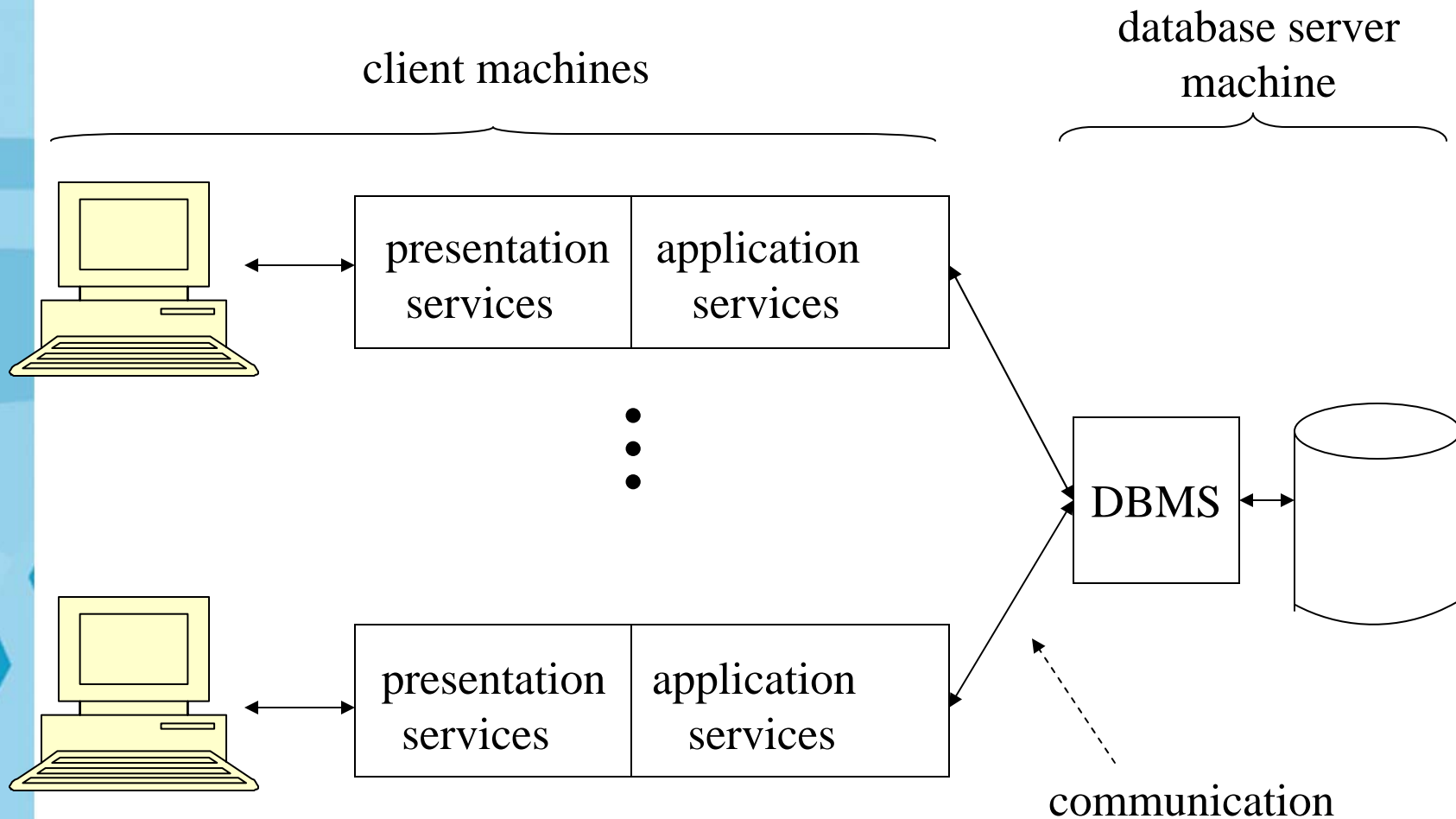




Transaction Processing in a Distributed System

- Decreased cost of hardware and communication make it possible to distribute components of transaction processing system
 - Dumb terminal replaced by computers
- Client/server organization generally used

Two-Tiered Model of TPS

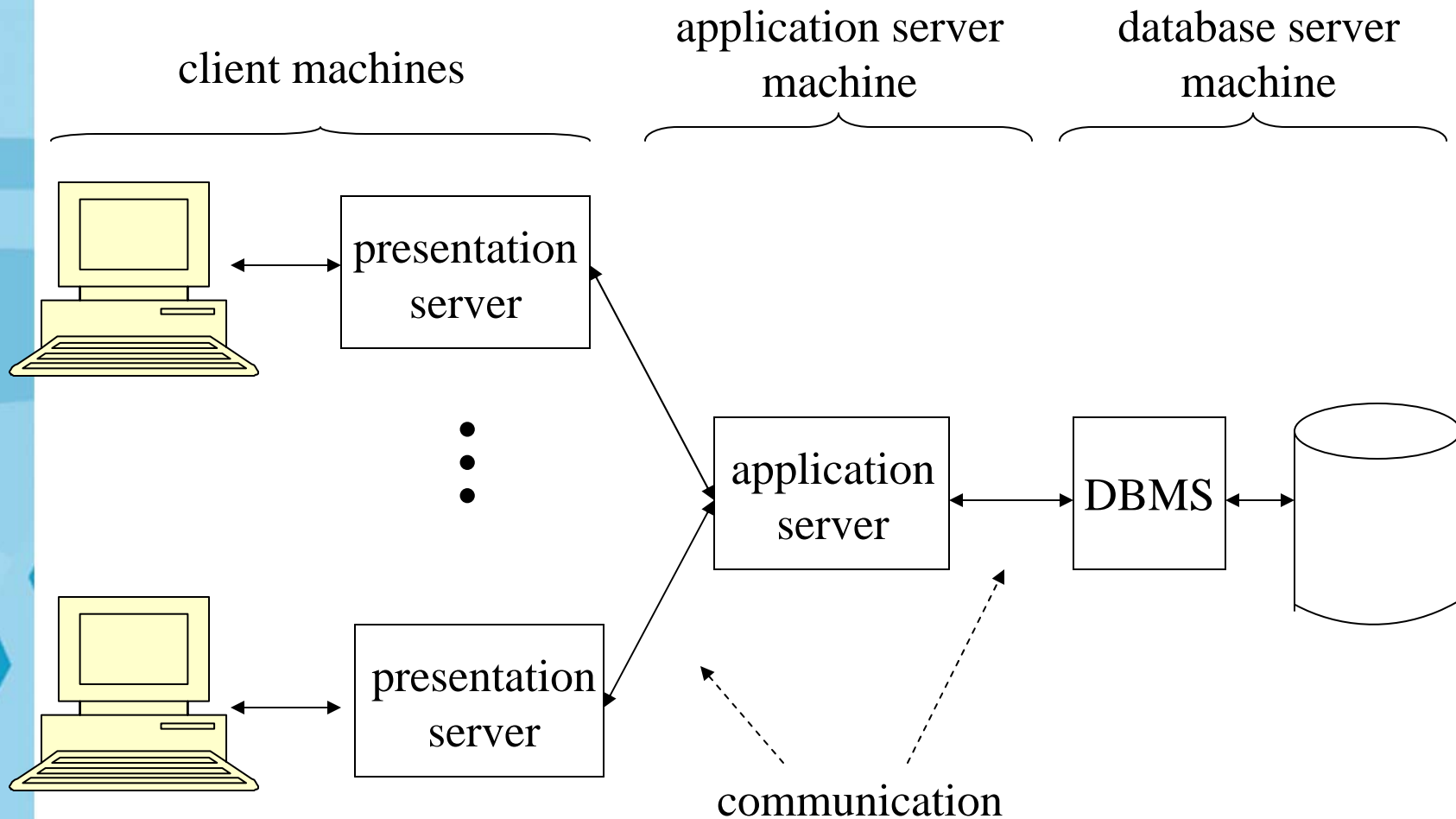




Use of Stored Procedures

- Advantages that result from having the database server export a stored procedure interface instead of an SQL interface
 - Security: procedures can be protected since they are maintained at the enterprise site
 - Network traffic reduced, response time reduced
 - Maintenance easier since newer versions don't have to be distributed to client sites
 - Authorization can be implemented at the procedure (rather than the statement) level
 - Procedures can be prepared in advance
 - Application services sees a higher level of abstraction, doesn't interact directly with database

Three-Tiered Model of TPS





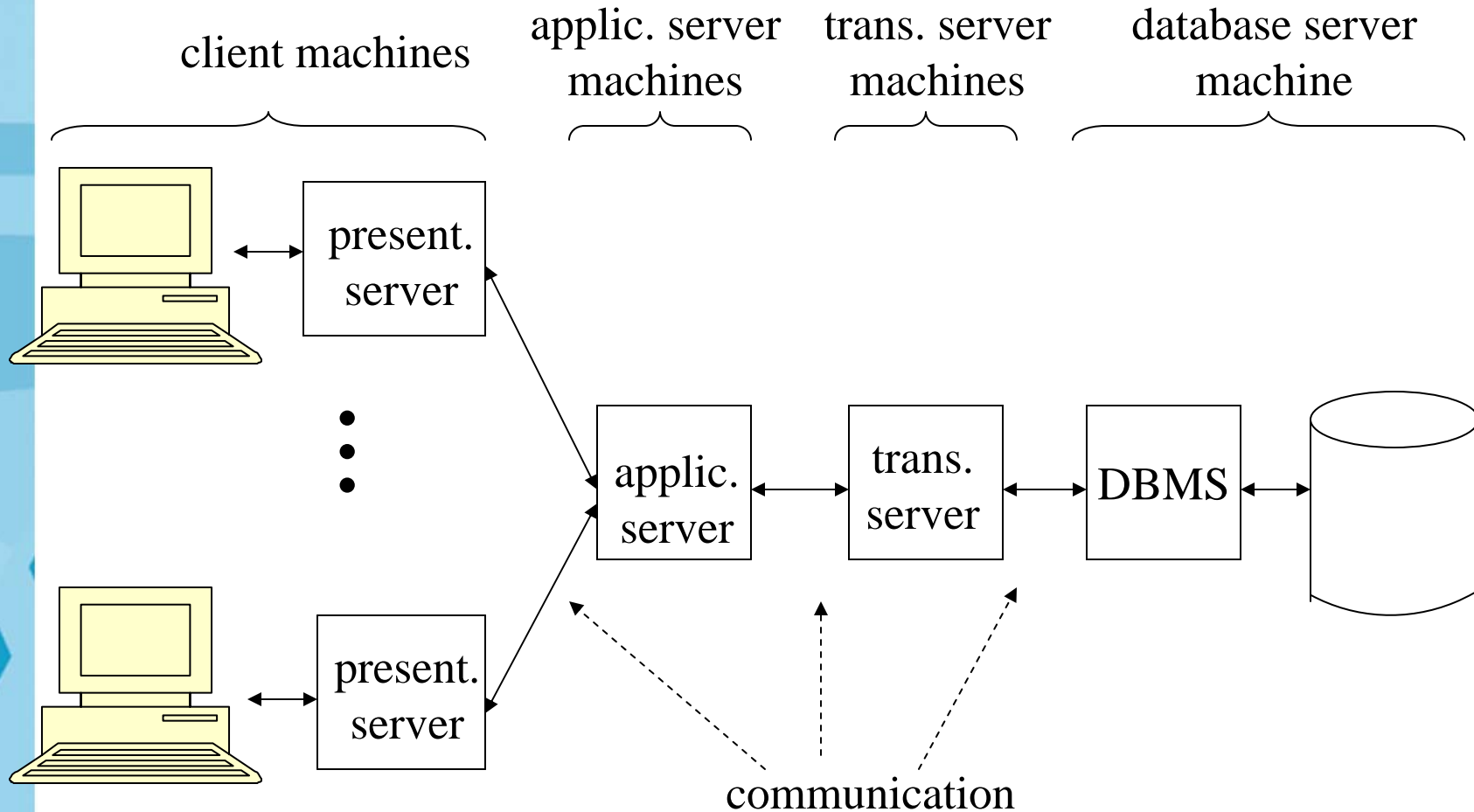
Application Server

- Sets transaction boundaries
- Acts as a *workflow controller*: implements user request as a sequence of tasks
 - e.g., registration = (check prerequisites, add student to course, bill student)
- Acts as a *router*
 - Distributed transactions involve multiple servers
 - Server classes are used for load balancing
- Since workflows might be time consuming and application server serves multiple clients, application server is often *multi-threaded*

Transaction Server

- Stored procedures off-loaded to separate (transaction) servers to reduce load on DBMS
- Transaction server located close to DBMS
 - Application server located close to clients
- Transaction server does bulk of data processing.
 - Transaction server might exist as a *server class*
 - Application server uses any available transaction server to execute a particular stored procedure; might do load balancing
 - Compare to application server which is multi-threaded

Three-Tiered Model of TPS

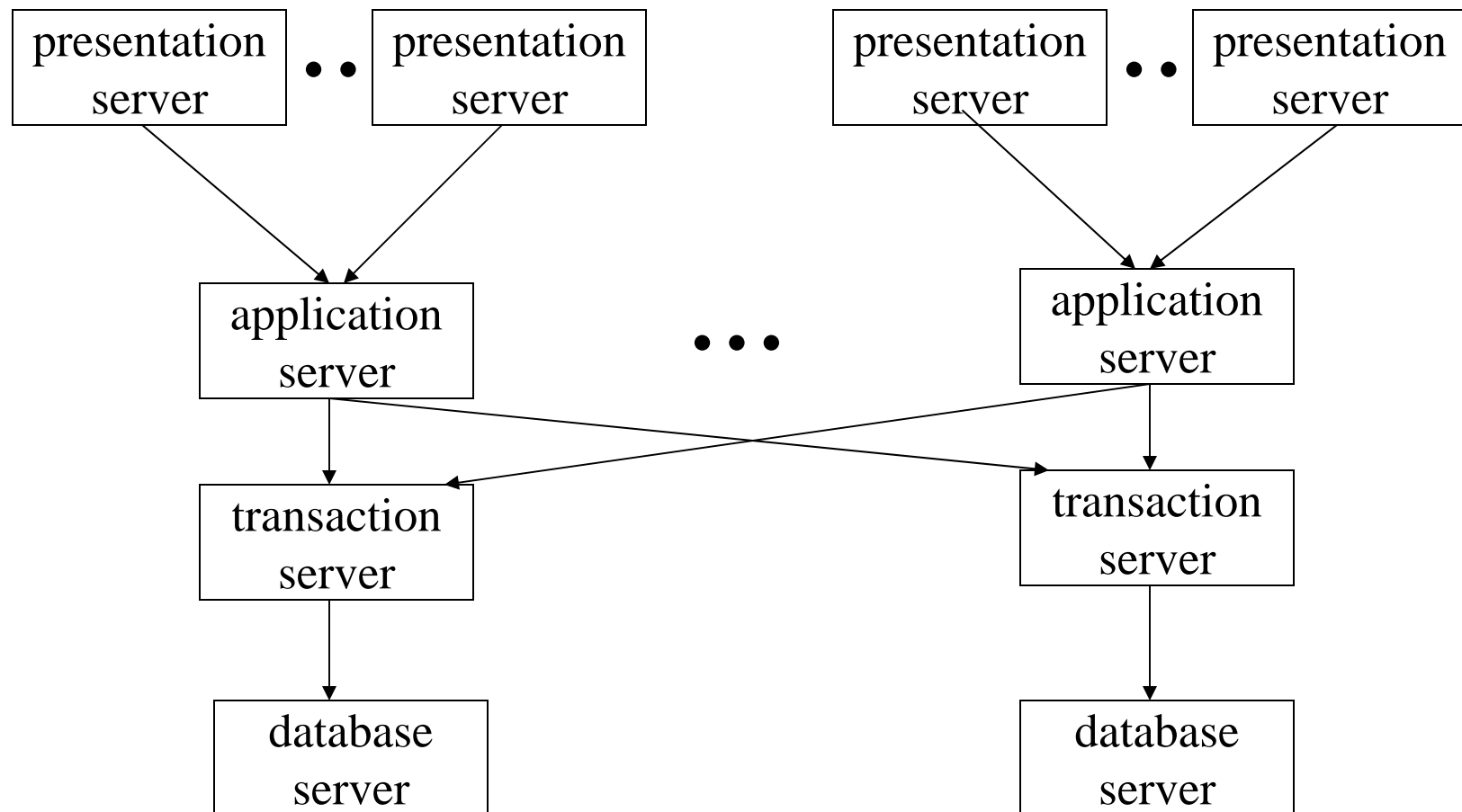




Levels of Abstraction

- Presentation server implements the abstraction of the user interface
- Application server implements the abstraction of a user request
- Stored procedures (or transaction server) implement the abstraction of individual sub-tasks
- Database server implements the abstraction of the relational model

Interconnection of Servers in Three-Tiered Model





Sessions and Context

- A *session* exists between two entities if they exchange messages while cooperating to perform some task
 - Each maintains some information describing the state of their participation in that task
 - State information is referred to as *context*

Communication in TPSs

- *Two-tiered model:*
 - Presentation/application server communicates with database server
- *Three-tiered model:*
 - Presentation server communicates with application server
 - Application server communicates with transaction/database server
- In each case, multiple messages have to be sent
 - Efficient and reliable communication essential
- Sessions are used to achieve these goals
 - Session set-up/take-down costly => session is long-term



Sessions

- Sessions established at different levels of abstraction:
 - Communication sessions (low level abstraction)
 - Context describes state of communication channel
 - Client/server sessions (high level abstraction)
 - Context used by server describes the client

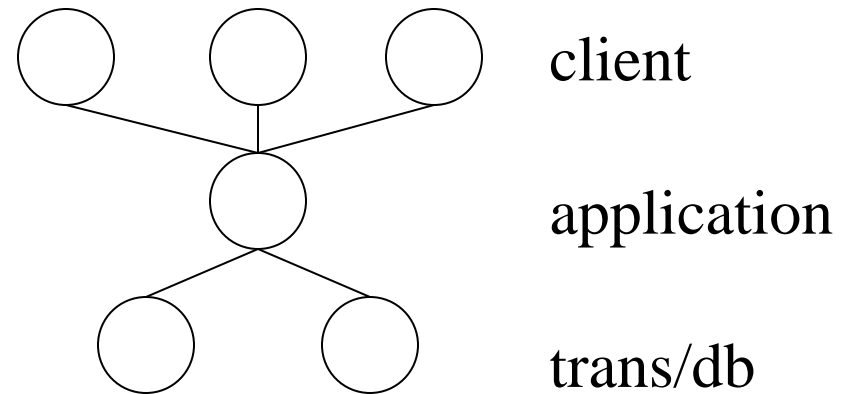
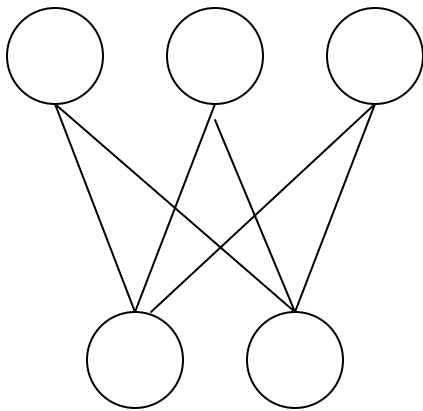


Communication Sessions

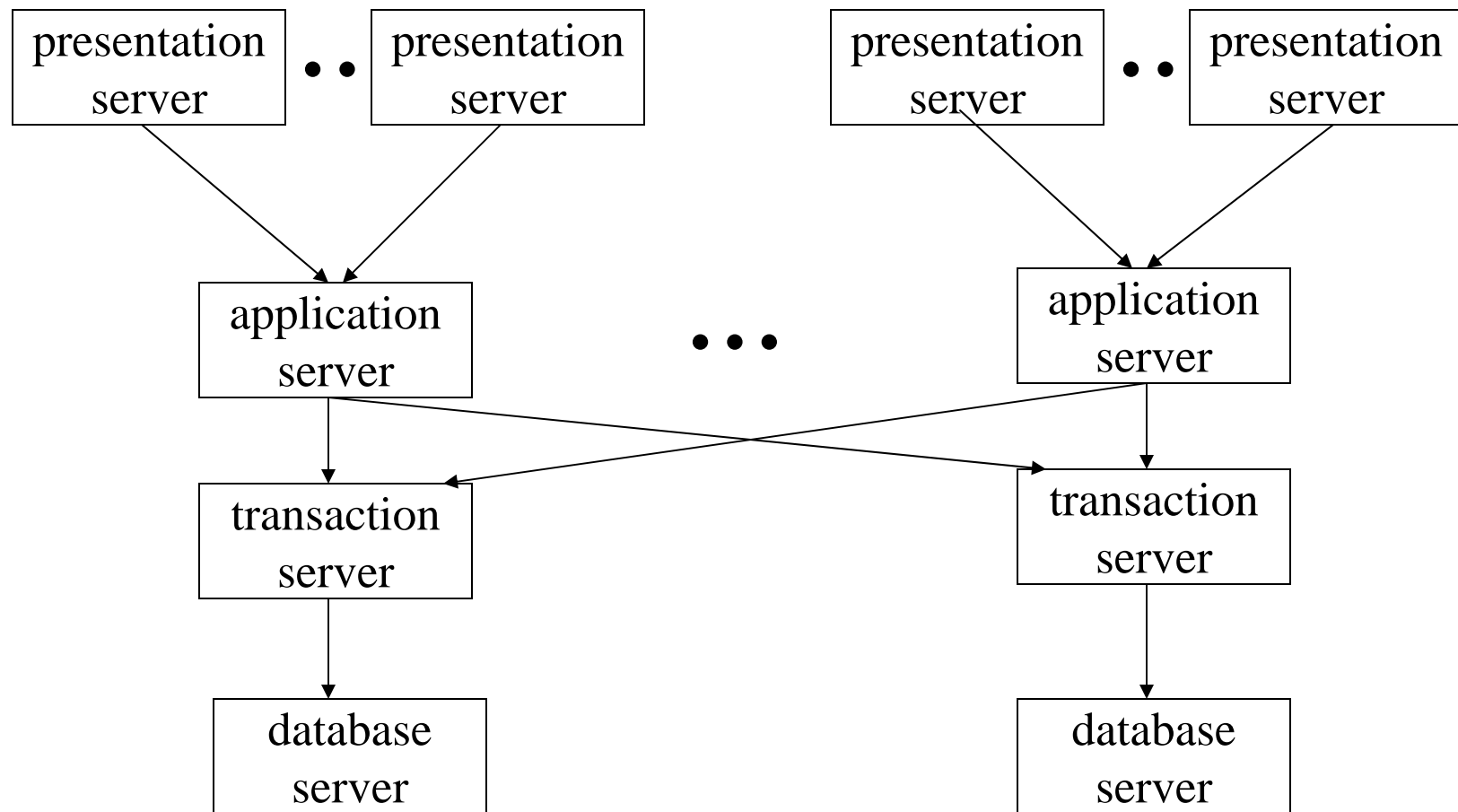
- Context: sequence number, addressing information, encryption keys, ...
- Overhead of session maintenance significant
 - Hence the number of sessions has to be limited
- Two-tiered model:
 - A client has a session with each database server it accesses
- Three-tiered model:
 - Each client has a session with its application server
 - Each application server *multiplexes* its connection to a database server over all transactions it supports

Number of Sessions

- Let n_1 be the number of clients, n_2 the number of application servers, and n_3 the number of transaction/db servers
- Sessions, two-tier (worst case) = $n_1 * n_3$
- Sessions, three-tier (worst case) = $n_1 + n_2 * n_3$
- Since $n_1 \gg n_2$, three-tiered model scales better



Sessions in Three-Tiered Model





Client/Server Sessions

- Server context (describing client) has to be maintained by server in order to handle a sequence of client requests:
 - What has client bought so far?
 - What row was accessed last?
 - What is client authorized to do?



Client/Server Sessions

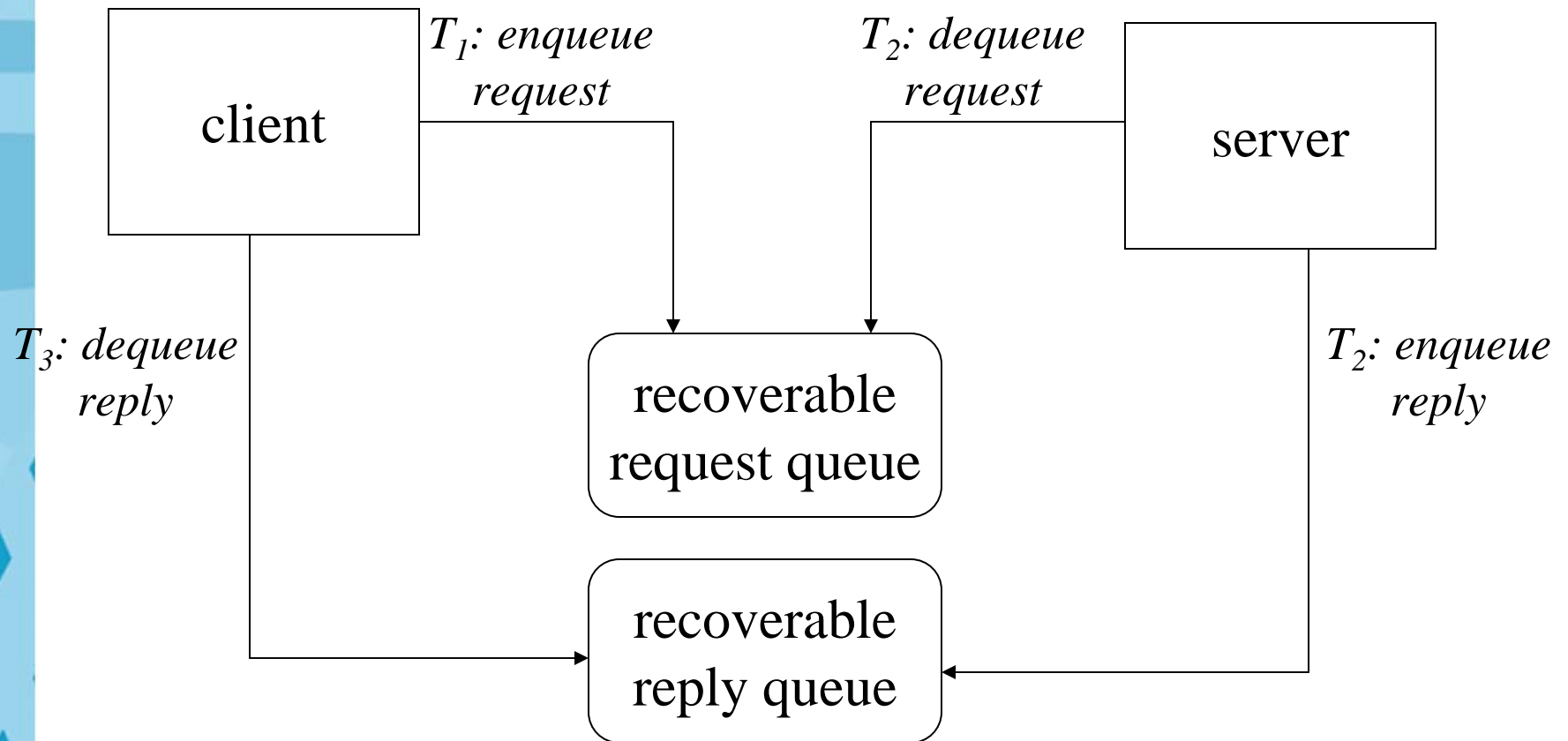
- Where is the server context stored?
 - At the server - but this does not accommodate:
 - Server classes: different server instances (e.g., transaction servers) may participate in a single session
 - Large number of clients maintaining long sessions: storage of context is a problem
 - In a central database - accessible to all servers in a server class
 - At the client - context passed back and forth with each request.
 - Context handle



Queued vs. Direct Transaction Processing

- *Direct:* Client waits until request is serviced. Service provided as quickly as possible and result is returned. Client and server are synchronized.
- *Queued:* Request enqueued and client continues execution. Server dequeues request at a later time and enqueues result. Client dequeues result later. Client and server unsynchronized.


Queued Transaction Processing





Queued Transaction Processing

- Three transactions on two recoverable queues
- **Advantages:**
 - Client can enter requests even if server is unavailable
 - Server can return results even if client is unavailable
 - Request will ultimately be served even if T_2 aborts (since queue is transactional)



Heterogeneous vs. Homogeneous TPSs

- *Homogeneous systems* are composed of HW and SW modules of a single vendor
 - Modules communicate through proprietary (often unpublished) interfaces
 - Hence, other vendor products cannot be included
 - Referred to as *TP-Lite* systems
- *Heterogeneous systems* are composed of HW and SW modules of different vendors
 - Modules communicate through standard, published interfaces
 - Referred to as *TP-Heavy* systems



Heterogeneous Systems

- Evolved from:
 - Need to integrate *legacy* modules produced by different vendors
 - Need to take advantage of products of many vendors
- *Middleware* is the software that integrates the components of a heterogeneous system and provides utility services
 - For example, supports communication (TCP/IP), security (Kerberos), global ACID properties, translation (JDBC)

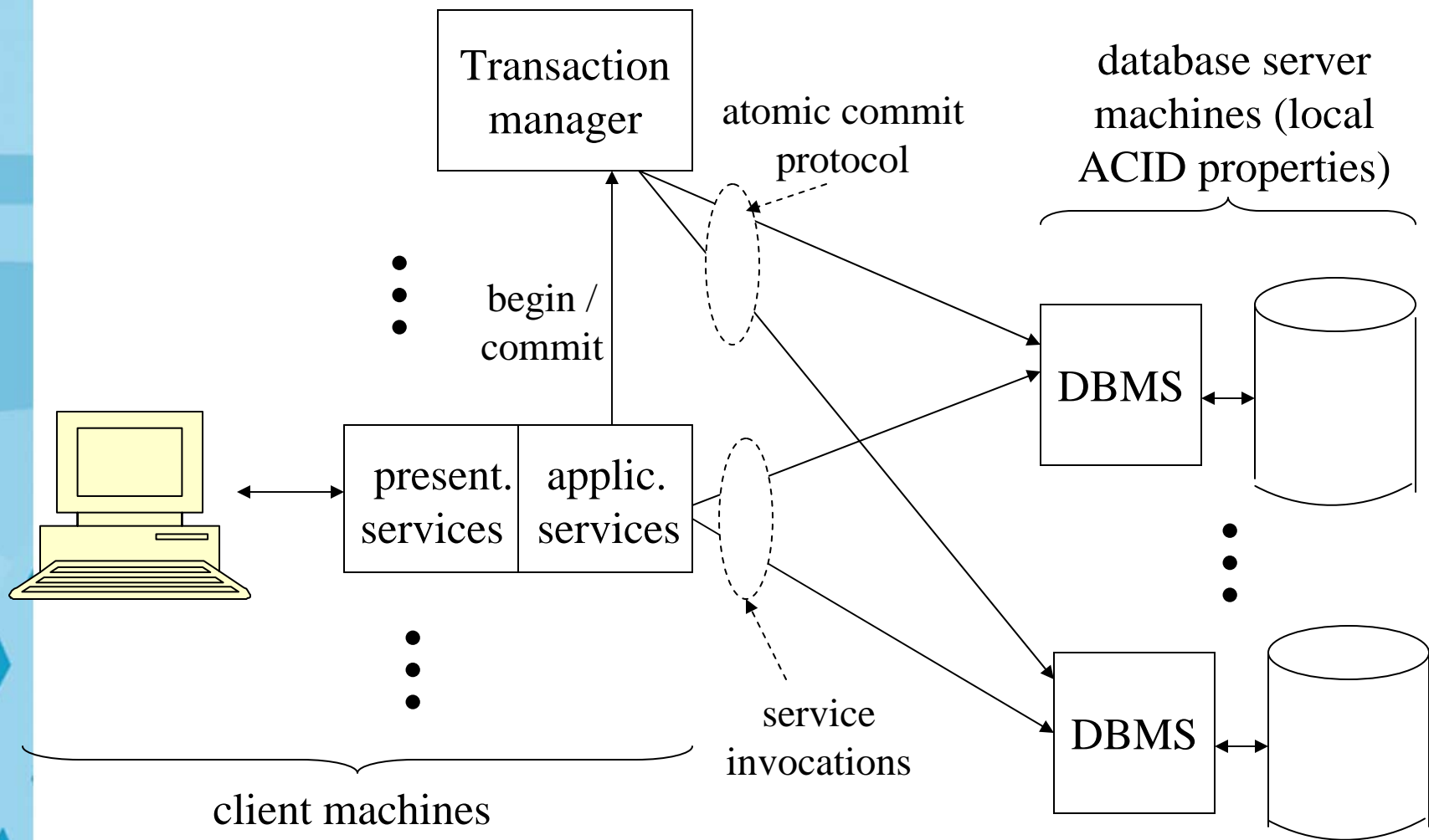


Transaction Manager

- Middleware to support global atomicity of distributed transactions
 - Application invokes manager when transaction is initiated
 - Manager is informed each time a new server joins the transaction
 - Application invokes manager when transaction completes
 - Manager coordinates *atomic commit protocol* among servers to ensure global atomicity

Transaction Manager

(Two-Tiered Model)

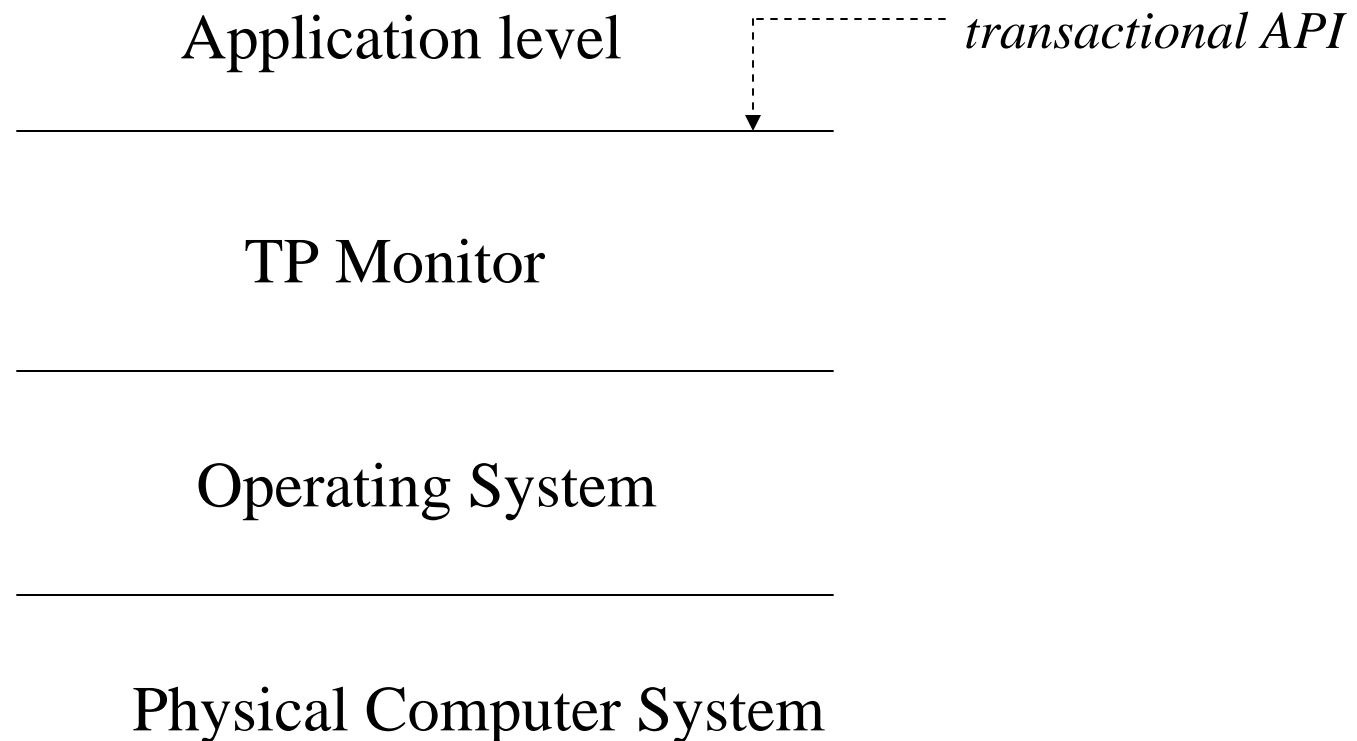




TP Monitor

- A *TP Monitor* is a collection of middleware components that is useful in building heterogeneous transaction processing systems
 - Includes transaction manager
 - Application independent services not usually provided by an operating system
- Layer of software between operating system and application
- Produces the abstraction of a (global) transaction

Layered Structure of a Transaction Processing System



TP Monitor Services

- Communication services
 - Built on message passing facility of OS
 - Transactional peer-to-peer and/or remote procedure call
- ACID properties
 - Local isolation for a (non-db) server might be provided by a lock manager
 - Implements locks that an application can explicitly associate with instances of any resource
 - Local atomicity for a (non-db) server might be provided by a log manager
 - Implements a log that can be explicitly used by an application to store data that can be used to roll back changes to a resource
 - Global isolation and atomicity are provided by transaction manager

TP Monitor Services

- Routing and load balancing
 - TP monitor can use load balancing to route a request to the least loaded member of a server class
- Threading
 - Threads can be thought of as low cost processes
 - Useful in servers (*e.g.*, application server) that might be maintaining sessions for a large number of clients
 - TP monitor provides threads if OS does not



TP Monitor Services

- Recoverable queues
- Security services
 - Encryption, authentication, and authorization
- Miscellaneous servers
 - File server
 - Clock server

Communication Services

- Modules of a distributed transaction must communicate
- Message passing facility of underlying OS is
 - inconvenient to use, lacks type checking.
 - does not support transaction abstraction (atomicity)
 - Distributed transactions spread via messages => message passing facility can support mechanism to keep track of subtransactions
- TP monitor builds an enhanced communication facility on top of message passing facility of OS
 - Transactional remote procedure call
 - Transactional peer-to-peer communication
 - Event communication

Remote Procedure Call (RPC)

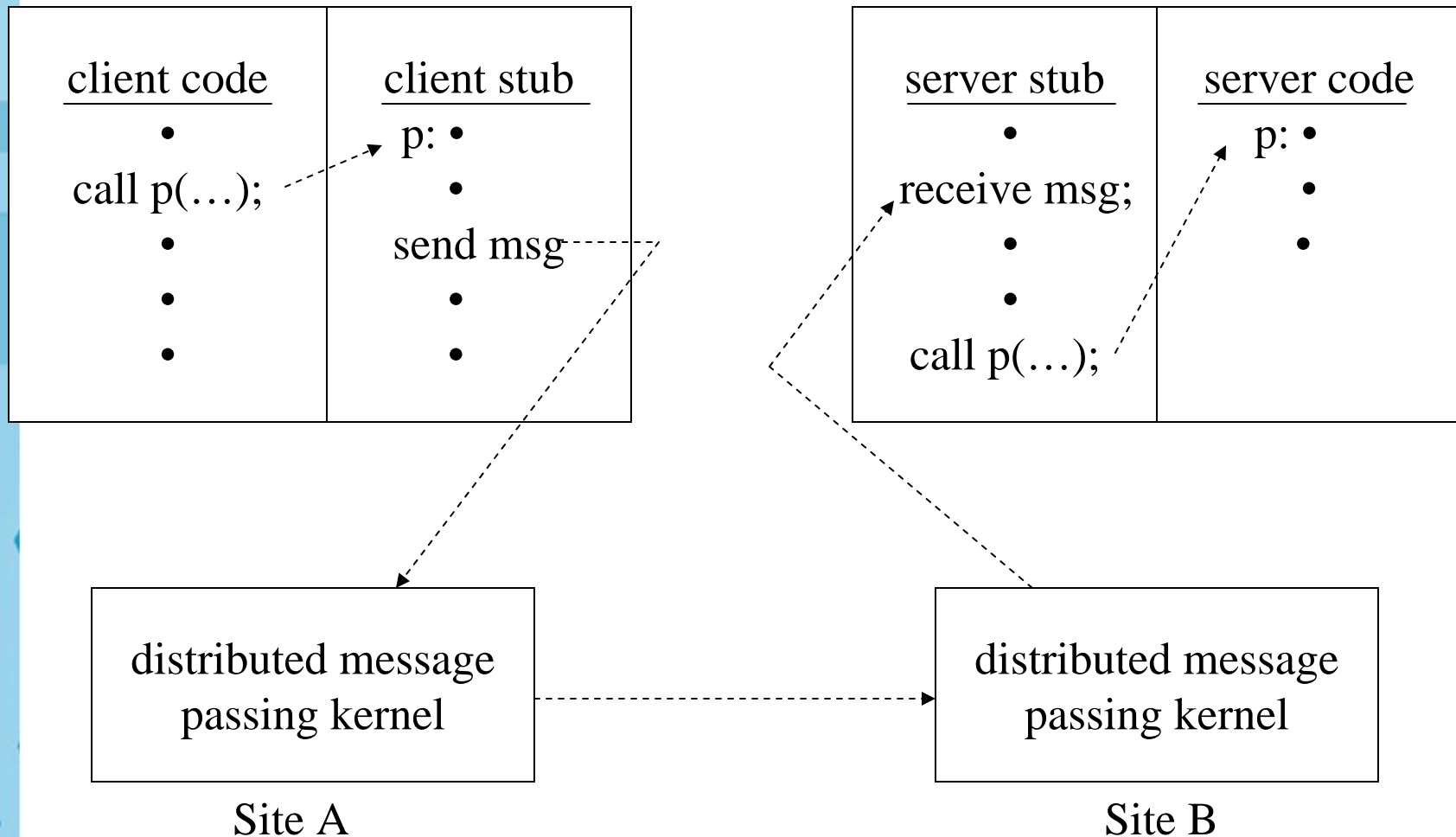
- Procedural interface
 - Convenient to use
 - Provides type checking
 - Naturally supports client/server model
- RPC extends procedural communication to distributed computations
- Deallocation of local variables limits ability to store context (*stateless*)
 - Context can be stored globally (*e.g.*, in database) or ...
 - passed between caller and callee (*context handle*)



Remote Procedure Call

- **Asymmetric** : caller invokes, callee responds
- **Synchronous**: caller waits for callee
- **Location transparent**: caller cannot tell whether
 - Callee is local or remote
 - Callee has moved from one site to another

RPC Implementation: Stubs



Stub Functions

- **Client stub:**
 - Locates server - sends globally unique name (character string) provided by application to *directory services*
 - Sets up connection to server
 - *Marshalls* arguments and procedure name
 - Sends invocation message (uses message passing)
- **Server stub:**
 - Unmarshalls arguments
 - Invokes procedure locally
 - Returns results



Connection Set-Up: IDL

- Server publishes its interface as a file written in **Interface Definition Language** (IDL). Specifies procedure names, arguments, etc
- IDL compiler produces header file and server-specific stubs
- Header file compiled with application code (type checking possible)
- Client stub linked to application



Connection Set-Up: Directory Services

- Interface does not specify the location of server that supports it.
 - Server might move
 - Interface might be supported by server class
- **Directory (Name) Services** provides run-time rendezvous.
 - Server registers its globally unique name, net address, interfaces it supports, protocols it uses, etc.
 - Client stub sends server name or interface identity
 - Directory service responds with address, protocol



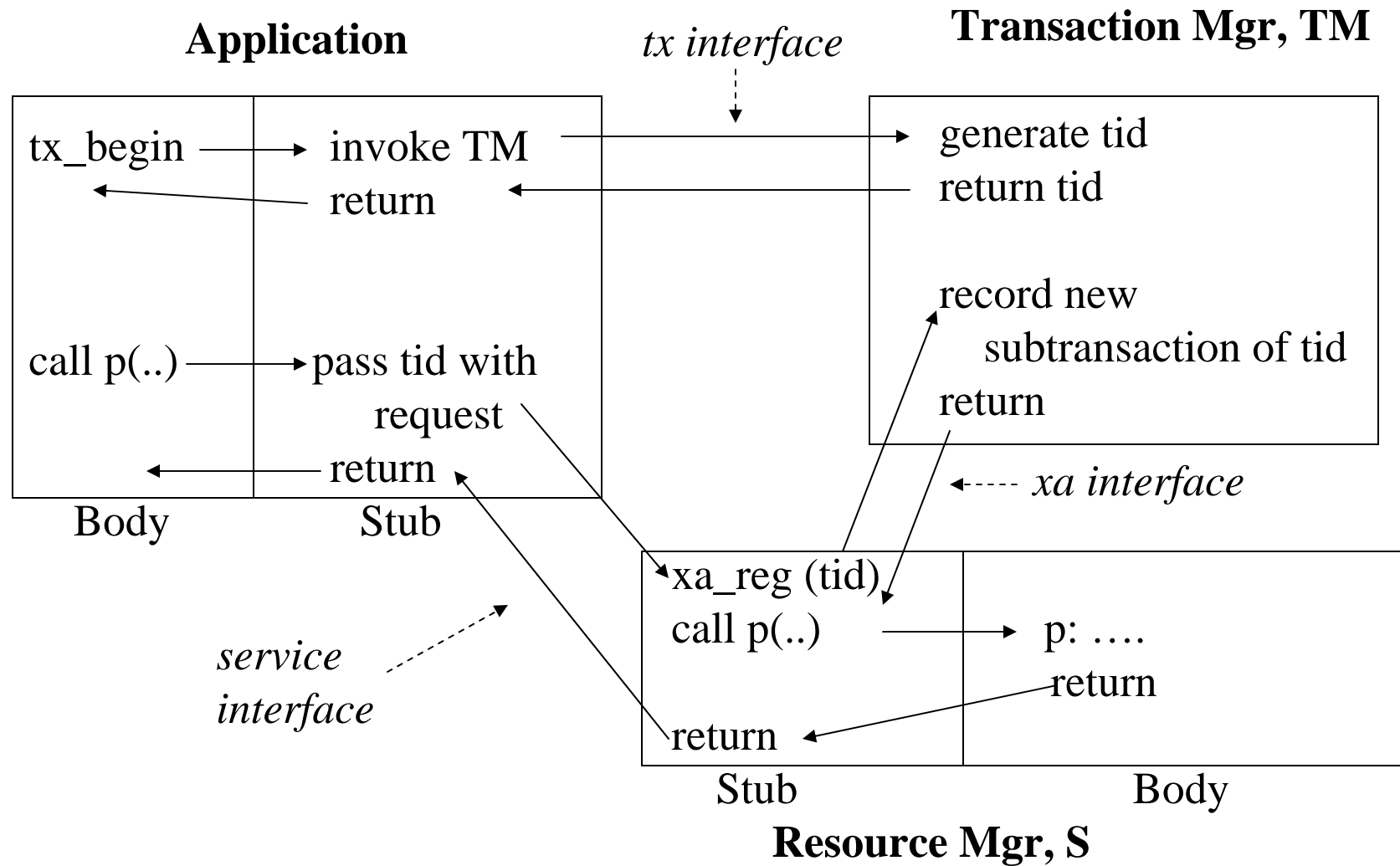
RPC: Failures

- Component failures (communication lines, computers) are expected
- Service can often be provided despite failures
- **Example:** no response to invocation message
 - Possible reasons: communication slow, message lost, server crashed, server slow
 - Possible actions: resend message, contact different server, abort client, continue to wait
 - **Problem:** different actions appropriate to different failures

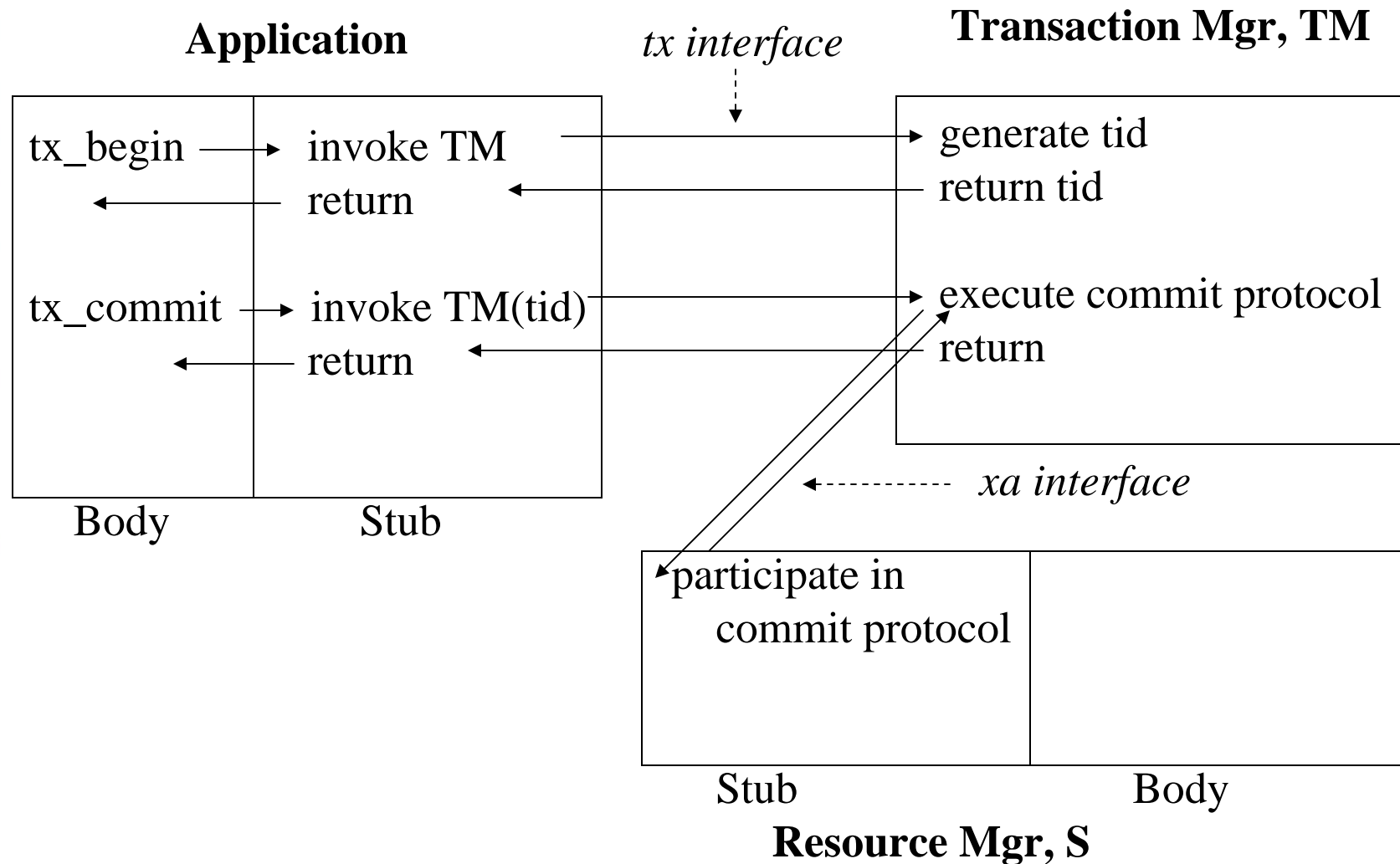
Transactional RPC and Global Atomicity

- Client executes *tx_begin* to initiate transaction
 - Client stub calls transaction manager (TM)
 - TM returns transaction id (*tid*)
- **Transactional RPC (TRPC):**
 - Client stub appends *tid* to each call to server
 - Server stub informs TM when it is called for first time
 - Server has joined the client's transaction.
- Client commits by executing *tx_commit*
 - Client stub calls TM
 - TM coordinates *atomic commit protocol* among servers to ensure global atomicity

Transaction Manager and TRPC



Transaction Manager and Atomic Commit Protocol



TRPC and Failure

- Stubs must guarantee *exactly once semantics* in handling failures. Either:
 - called procedure executed exactly once and transaction can continue, or
 - called procedure does not execute and transaction aborts
- Suppose called procedure makes a (nested) call to another server and then crashes.
 - *Orphan* is left in system when transaction aborted
 - Orphan elimination algorithm required



Peer-To-Peer Communication

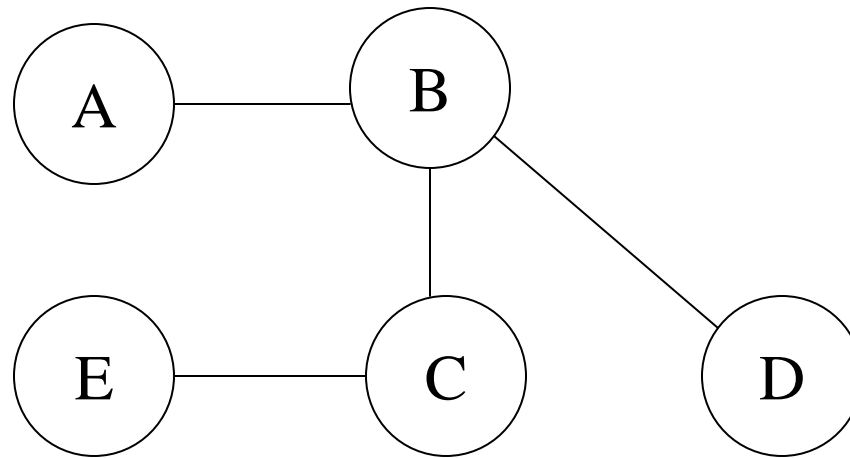
- **Symmetric:** Once a connection is established communicants are equal (peers) - both can execute send/receive. If requestor is part of a transaction, requestee joins the transaction
- **Asynchronous:** Sender continues executing after send; arbitrary patterns (streams) of messages possible; communication more flexible, complex, and error prone than RPC
- **Not location transparent**



Peer-To-Peer Communication

- Connection can be *half duplex* (only one peer is in send mode at a time) or *full duplex* (both can send simultaneously)
- Communication is *stateful*: each peer can maintain context over duration of exchange.
 - Each message received can be interpreted with respect to that context

Peer-To-Peer Communication




- A module can have multiple connections concurrently
 - Each connection associated with transaction that created it
 - A new instance of called program is created when a connection is established
 - The connections associated with a particular transaction form an acyclic graph
 - All nodes of graph coequal (no root as in RPC)



Peer-To-Peer and Commit

- **Problems:**

- Coequal status: Since there is no root (as in RPC) who initiates the commit?
- Asynchronous: How can initiator of commit be sure that all nodes have completed their computations? (This is not a problem with RPC.)



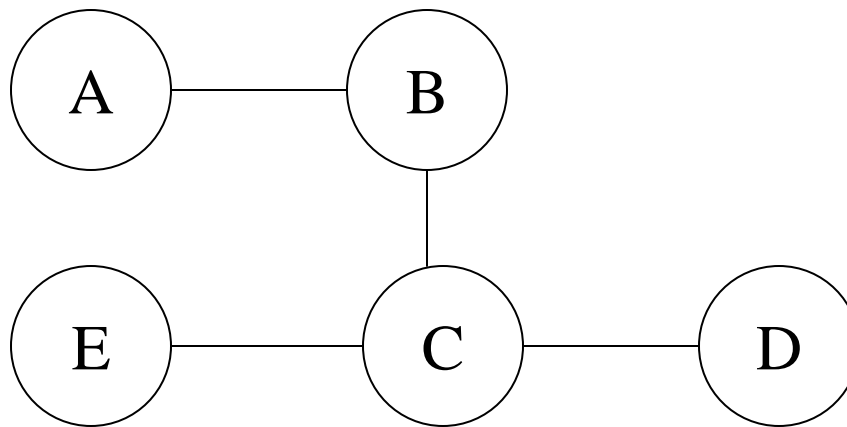
Solution: Syncpoints in Half Duplex Systems

- One node, *A*, initiates commit. It does this when
 - it completes its computation and
 - all its connections are in send mode.
- *A* declares a *syncpoint* and waits for protocol to complete.
- TP monitor sends syncpoint message to all neighbors, *B*.

Syncpoint Protocol (con't)

- When *B* completes its computation and all its connections (other than connection to *A*) are in send mode, *B* declares syncpoint and waits.
- TP monitor sends syncpoint message to all *B*'s neighbors (other than *A*).
- When syncpoint message reaches all nodes, all computation is complete and all have agreed to commit.

Syncpoint Error



- **Problem:** Two nodes, all of whose connections are in send mode, initiate commit concurrently
- **Result:** Protocol does not terminate. First node to receive two syncpoint messages cannot declare a syncpoint



Handling Exceptional Situations

- With TRPC and peer-to-peer communication, a server services requests and is idle when no request is pending.
- **Problem:** Sometimes a server needs an interrupt mechanism to inform it of exceptional events that require immediate response even if it is busy.

Example

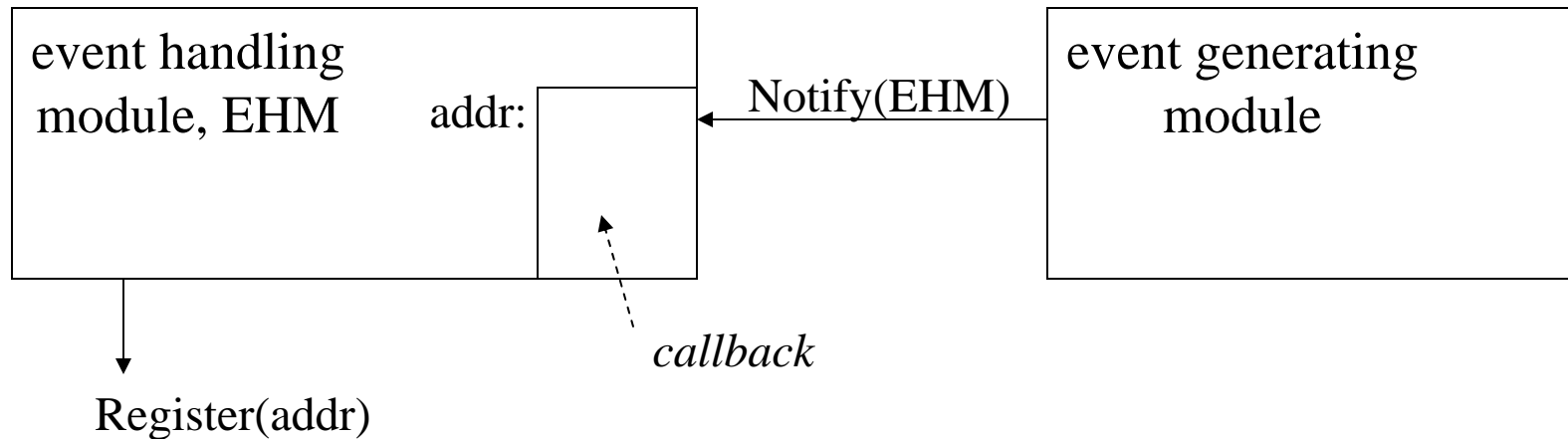
- *M1* controls flow of chemicals into furnace
- *M2* monitors temperature
- *M1* must be informed when temperature reaches limit
- **Solution 1:** *M1* polls *M2* periodically to check temperature
 - Wasteful if limit rarely reached, very wasteful if *M1* must respond fast (polling must be frequent)
- **Solution 2:** *M2* interrupts *M1* when limit reached



Event Communication

- Event handling module *registers* address of its event handling routine with TP monitor using event API
 - Handler operates as an interrupt routine
- Event generating module recognizes event and *notifies* event handling module using event API
- TP monitor interrupts event handling module and causes it to execute handling routine

Event Communication

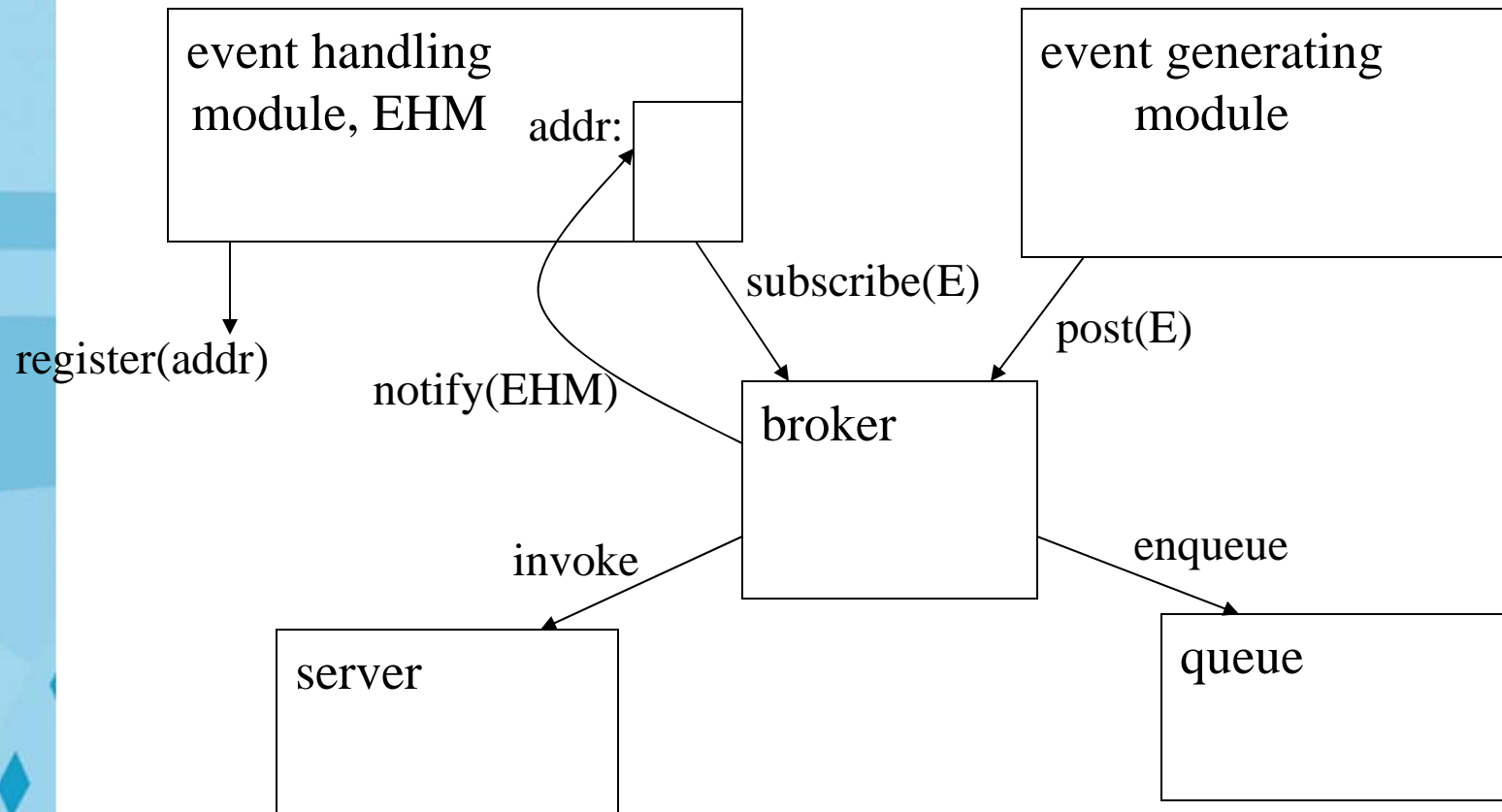


- If event generating module recognizes event in the context of a transaction, T , execution of handler is part of T
- **Problem:** event generating module must know the identity of event handling module

Event Broker

- **Solution:** Use broker as an intermediary:
 - Events are named (E)
 - Event handling module *registers* handler with the TP monitor and *subscribes* to E by calling the broker with E as a parameter
 - Event generating module *posts* E to the broker when it occurs.
 - Broker *notifies* event handler
 - If recognition of E in generating module is part of a transaction, execution of handler is also

Event Broker



- Broker can cause alternate actions in addition to notifying handler



Storage Architecture

- Much of this presentation deals with various architectures for increasing performance
- One performance bottleneck we have not yet discussed is disk I/O
- It is hard to get throughput of thousands of transactions per second when each disk access requires a time measured in thousandths of a second



Disk Cache

- The DBMS maintains a disk cache in main memory
 - Recently accessed disk pages are kept in the cache and if at some later time a transaction accesses that page, it can access the cached version
 - Many designers try to obtain over 90% hit rate in the cache
 - Many cache sizes are in the tens of gigabytes



RAID Systems

- A RAID (Redundant Array of Independent Disks) consists of a set of disks configured to look like a single disk with increased throughput and reliability
 - Increased throughput is obtained by striping or partitioning each file over a number of disks, thus decreasing the time to access that file
 - Increased reliability is obtained by storing the data redundantly, for example using parity bits



RAID Systems (continued)

- Here we point out that a number of RAID levels have been defined, depending on the type of striping and redundancy used
- The levels usually recommended for transaction processing systems are
 - Level 5: Block-level striping of both data and parity
 - Level 10: A striped array of mirrored disks



NAS and SAN

- In a NAS (Network Attached Storage) a file server, sometimes called an appliance, is directly connected to the same network as the application server and other servers
 - The files on the appliance can be shared by all the servers



NAS and SAN (continued)

- In a SAN (Storage Area Network) a server connects to its storage devices over a separate high speed network
 - The network operates at about the same speed as the bus on which a disk might be connected to the server
 - The server accesses the storage devices using the same commands and at the same speed as if it were connected on a bus




NAS and SAN (continued)

- Both NAS and SAN can scale to more storage devices than if those devices were connected directly to the server
- SANs are usually considered preferable for high performance transaction processing systems because the DBMS can access the storage devices directly instead of having to go through a server.



Transaction Processing on the Internet

- The growth of the Internet has stimulated the development of many Internet services involving transaction processing
 - Often with throughput requirements of thousands of transactions per second




C2B and B2B Services

- C2B (Customer-to-Business) services
 - Usually involve people interacting with the system through their browsers
- B2B (Business-to-Business) services
 - Usually fully automated
 - Programs on one business's Web site communicates with programs on another business's Web site



Front-End and Back-End Services

- Front-end services refers to the interface a service offers to customers and businesses
 - How it is described to users
 - How it is invoked
- Back-end services refers to how that service is actually implemented



Front-End and Back-End Services (continued)

- Next we discuss architectures for C2B systems in which the front-end services are provided by a browser and the back-end services can be implemented as discussed earlier in the presentation
- Then we discuss how back-end services can be implemented by commercial Web application servers
- These same back-end implementations can be used for B2B services using a different front-end
 - B2B front-end services implemented as Web Services



C2B Transaction Processing on the Internet

- Common information interchange method
 - Browser requests information from server
 - Server sends to browser HTML page and possibly one or more Java programs called applets
 - User interacts with page and Java programs and sends information back to server
 - Java servlet on server reads information from user, processes it, perhaps accesses a database, and sends new page back to browser



C2B Transaction Processing on the Internet (continued)

- Servlets have a lifetime that extends beyond one user
 - When it is started, it creates a number of threads
 - These threads are assigned dynamically to each request
- Servlets provide API for maintaining context
 - Context information is stored in file on Web server that is accessed through a *session number*
 - *Cookies*: servlet places session number in a cookie file in browser; subsequent servlets can access cookie
 - *Hidden fields in HTML*: servlet places session number in hidden field in HTML document it sends to browser; hidden field is not displayed but is returned with HTML document
 - *Appended field to HTTP return address*: session number is appended to HTTP return address and can be accessed by next servlet



Architectures for Transaction Processing on the Internet

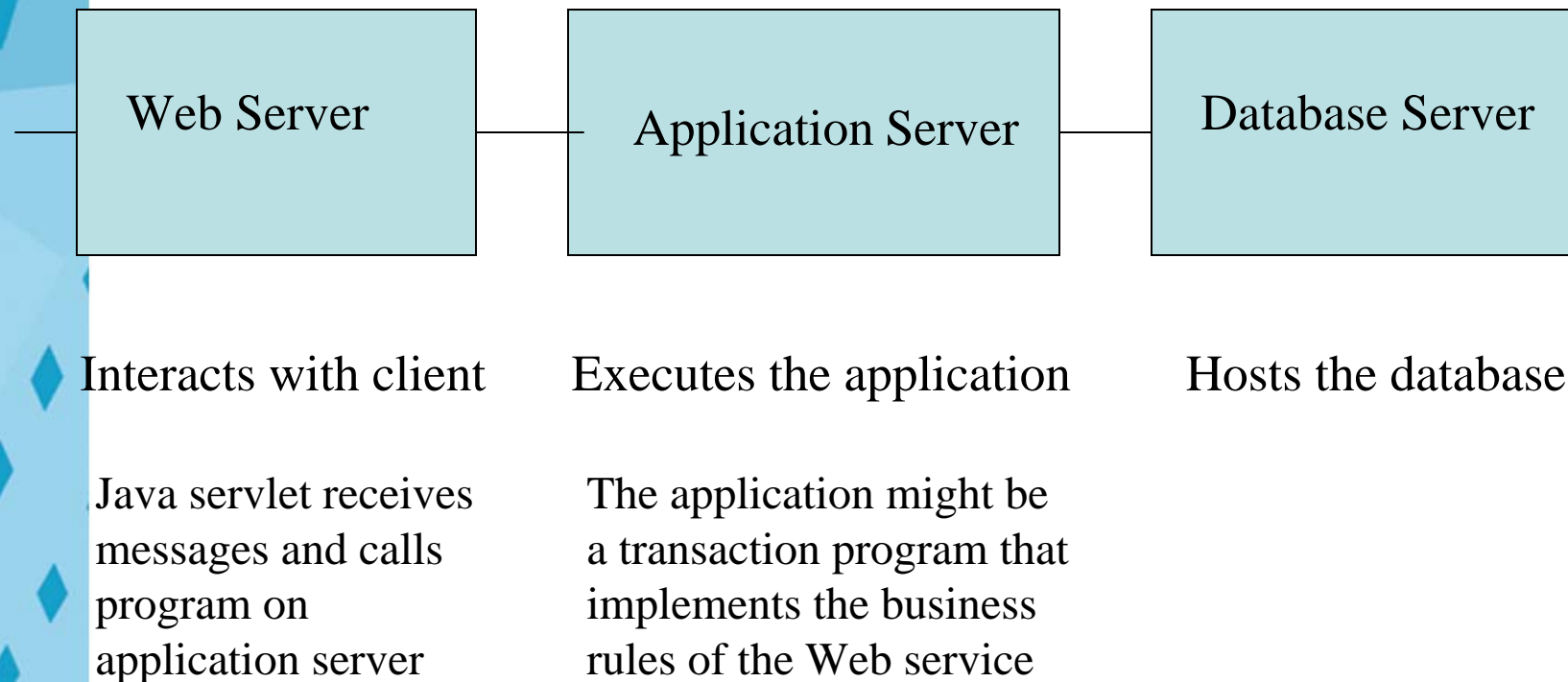
- Browser plays the role of presentation server and application server
 - Java applet on browser implements the transaction and accesses database using JDBC
- Browser plays the role of presentation server, and servlet program on server plays the role of application server
 - Servlet program implements the transaction and accesses database using JDBC



Architectures for Transaction Processing on the Internet

- Many high throughput applications require a three- or four-tiered architecture
 - After getting inputs from browser, the servlet program initiates the transaction on the application server, which is not connected to the Internet
 - Application server might be separated from the Web server by a firewall
 - From the TP system's viewpoint, the browser and servlet program together are acting as the presentation server

Architecture of a Web Transaction Processing System





Web Server

- HTTP Interface to Web
 - Java **servlet** on Web server interacts with *client's* browser using HTTP messages and then initiates programs on the application server



Web Application Server

- A Web application server is a set of tools and modules for building and executing transaction processing systems for the Web
 - Including the application server tier of the system
- Name is confusing because *application server* is the name usually given to the middle tier in an transaction processing system



Web Application Servers (continued)

- Most Web application servers support the J2EE (Java 2 Enterprise Edition) standards
 - Or Microsoft .NET
- We discuss J2EE
 - J2EE One language, many platforms
 - A standard implemented by many vendors
 - .NET One platform, many languages
 - A set of products of Microsoft



J2EE

- J2EE defines a set of services and classes particularly oriented toward transaction-oriented Web services
 - Java servlets
 - Enterprise Java beans



Enterprise Java Beans

- Java classes that implement the business methods of an enterprise
- Execute within an infrastructure of services provided by the Web application server
 - Supports transactions, persistence, concurrency, authorization, etc.
 - Implements declarative transaction semantics
 - The bean programmer can just declare that a particular method is to be a transaction and does not have to specify the *begin* and *commit* commands
 - Bean programmer can focus on business methods of the enterprise rather on details of system implementation



Entity Beans

- An **entity bean** represents a persistent business object whose state is stored in the database
 - Each entity bean corresponds to a database table
 - Each instance of that bean corresponds to a row in that table.



Example of an Entity Bean

- An entity bean called `Account`, which corresponds to a database table `Account`
 - Each instance of that bean corresponds to a row in that table
- `Account` has fields that include `AccountId` and `Balance`
 - `AccountId` is the primary key
 - Every entity bean has a *FindByPrimaryKey* method that can be used to find the bean based on its primary key
- `Account` has other methods that might include *Deposit* and *Withdraw*



Persistence of Entity Beans

- Any changes to the bean are *persistent* in that those changes are propagated to the corresponding database items
- This persistence can be managed either manually by the bean itself using standard JDBC statements or automatically by the system (as described later)
- The system can also automatically manage the authorization and transactional properties of the bean (as described later)



Session Bean

- A **session bean** represents a client performing interactions within a session using the business methods of the enterprise
 - A session is a sequence of interactions by a user to accomplish some objective. For example, a session might include selecting items from a catalog and then purchasing them.
- The session bean retains its state during all the interactions of the session
 - Stateless session beans also exist



Example of a Session Bean

- ShoppingCart provides the services of adding items to a “shopping cart” and then purchasing the selected items
 - Methods include *AddItemToShoppingCart* and *Checkout*
 - ShoppingCart maintains state during all the interactions of a session
 - It remembers what items are in the shopping cart



Session Beans and Entity Beans

- Session beans can call methods in entity beans
 - The *Checkout* method of the ShoppingCart session bean calls appropriate methods in the Customer, Order, and Shipping entity beans to record the order in the database



Session Bean Transactions

- Session beans can be transactional
 - The transaction can be managed manually by the bean itself using standard JDBC or JTA (Java Transaction API) calls or automatically by the system (as described below)



Message-Driven Beans

- All of the communication so far is *synchronous*
 - A session bean calls an entity bean and waits for a reply
- Sometimes the sender of a message does not need to wait for a reply
 - Communication can be *asynchronous*
 - Thus increasing throughput
 - Message-driven beans are provided for this purpose
- A **message-driven bean** is like a session bean in that it implements the business methods of the enterprise
 - It is called when an asynchronous JMS message is placed on the message queue to which it is associated
 - Its *onMessage* method is called by the system to process the message



Example of a Message-Driven Bean

- When shopping cart *Checkout* method completes, it sends an asynchronous message to the shipping department to ship the purchased goods
- The shipping department maintains a message queue, *ShippingMessageQ*, and a message driven bean, *ShippingMessageQListener*, associated with that queue
- When a message is placed on the queue, the system selects an instance of the bean to process it and calls that bean's *onMessage* method



Structure of an Enterprise Bean

- The bean class
 - Contains the implementations of the business methods of the enterprise
- A remote interface (also optionally a local interface)
 - Used by clients to access the bean class remotely, using RMI (or locally with the local interface)
 - Acts as a proxy for the bean class
 - Includes declarations of all the business methods
- A home interface (also optionally a local home interface)
 - Contains methods that control bean's life cycle
 - Create, remove
 - Also finder methods(e.g. *FindByPrimaryKey*) methods



Structure of an Enterprise Bean (continued)

- A deployment descriptor
 - Declarative metadata for the bean
 - Describes persistence, transactional, and authorization properties



Example of Deployment Descriptor

- The deployment descriptor for a banking application might say that
 - The *Withdraw* method of an Account entity bean
 - Is to be executed as a transaction
 - Can be executed either by the account owner or by a teller
 - The *Balance* field of the Account Bean
 - Has its persistence managed by the system
 - Any changes are automatically propagated to the DB
- Deployment descriptors are written in XML



Portion of a Deployment Descriptor Describing Authorization

```
<method-permission>  
  <role-name> teller </role-name>  
  <method>  
    <ejb-name> Account </ejb-name>  
    <method-name> Withdraw </method-name>  
  </method>  
</method-permission>
```



EJB Container

- Enterprise beans together with their deployment descriptors are encapsulated within an **EJB container** supplied by the Web application server
- The EJB container provides system-level support for the beans based on information in their deployment descriptors



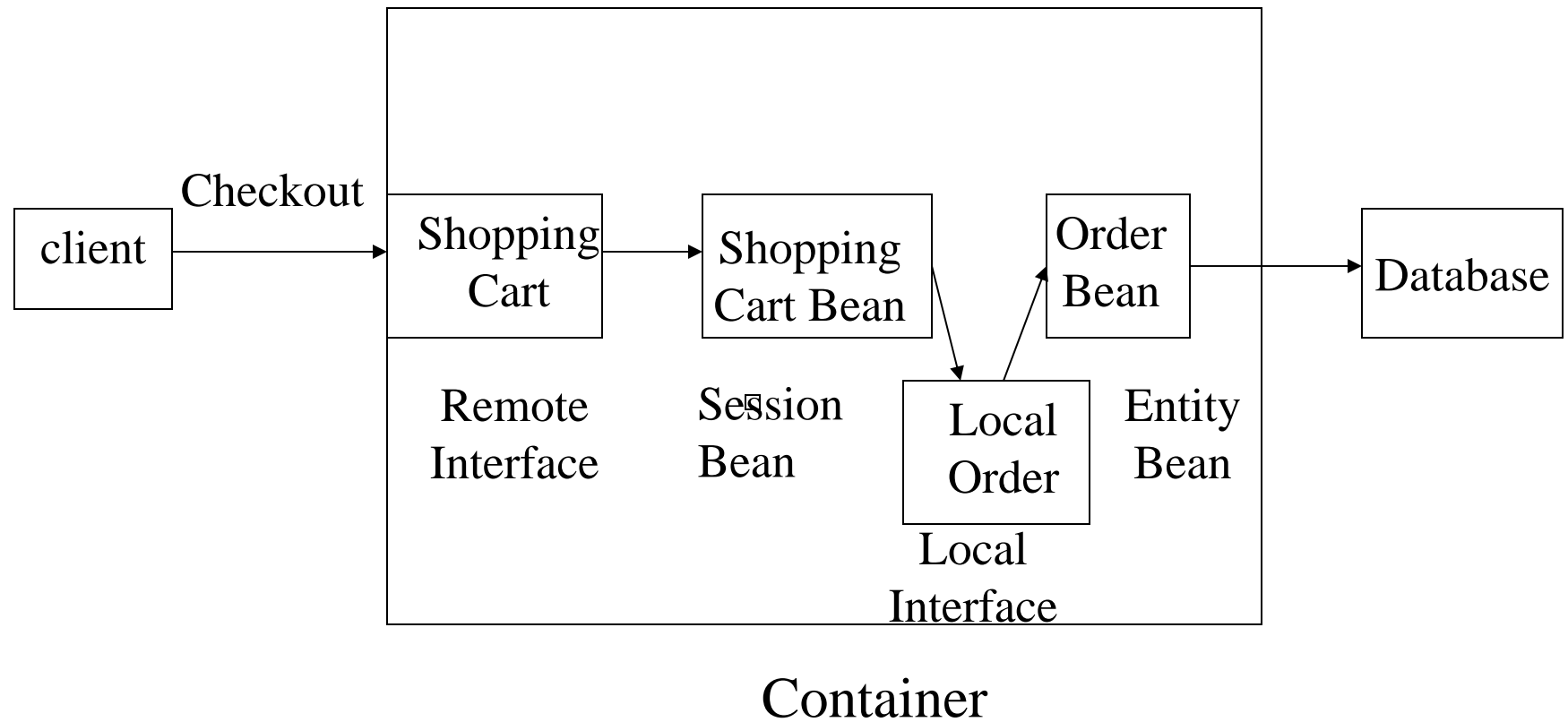
EJB Container (continued)

- The EJB container provides this support by intervening before and after each bean method is called and performing whatever actions are necessary
 - When a method is called, the call goes to the similarly named interface method
 - The interface method performs whatever actions are necessary before and after calling the bean method




EJB Container (continued)

- For example, if the deployment descriptor says the method is to run as a transaction
 - The interface method starts the transaction before calling the method
 - Commits the transaction when the method completes
- The EJB container supplies the code for the interface methods.




Remote and Local Interfaces Within a Container



Persistence of Entity Beans

- Persistence of entity beans can be managed
 - Manually by the bean itself (bean-managed persistence) using standard JDBC statements
 - Automatically by the container (container-managed persistence, cmp)



Example of Deployment Descriptor for Container Managed Persistence

```
<persistence-type> container </persistence-type>  
  <cmp-field>  
    <field-name> balance </field-name>  
  </cmp-field>
```

Get and Set Methods

- The entity bean must contain declarations for *get* and *set* methods. For example

public abstract float getBalance()

public abstract void setBalance (float balance)

- The container generates code for these methods
- A client of the bean, for example a session bean, can use
 - a finder method, for example, *FindByPrimaryKey()*, to find an entity bean and then
 - a *get* or *set* method to read or update specific fields of that bean.



EJB QL Language

- The container will generate the code for the *FindByPrimaryKey()* method
- The container will also generate code for other finder methods described in the deployment descriptor
 - These methods are described in the deployment descriptor using the **EJB QL (EJB Query Language)**
 - EJB QL is used to find one or more entity beans based on criteria other than their primary key

Example of EJB QL

```
<query>
  <query-method>
    <method-name>FindByName </method-name>
    <method-params>
      <method-param>string</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    SELECT OBJECT (A) FROM Account A WHERE A.Name = ?1
  </ejb-ql>
</query>
```



Create and Remove Methods

- A client of an entity bean can use the create and remove methods in its home interface to create and remove instances of that entity (rows in the database table).



Container-Managed Relationships

- In addition to fields that represent data, entity beans can have fields that represent relationships
 - One-to-one, One-to-many, Many-to-many
 - As with other database applications, these relationships can be used to find entity beans based on their relationships to other beans.
- Example: there might be a many-to-many relationship, *Signers*, relating Account bean and BankCustomer bean
 - *Signers* specifies which customers can sign checks on which accounts



Container-Managed Relationships (continued)

- Relationships can be declared in the deployment descriptor to be container-managed
 - For a many-to-many relationship such as *Signers*, the container will automatically create a new table to manage the relationship
 - For a one-to-one relationship, the container will automatically generate a foreign key

Portion of Deployment Descriptor

```
<ejb-relation>
  <ejb-relation-name>Signers</ejb-relation-name>
  <ejb-relationship-role>
    <ejb-relationship-role-name>account-has-signers
                                </ejb-relationship-role-name>

    <multiplicity>many</multiplicity>
    <relationship-role-source>
      <ejb-name>Account</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>Signers</cmr-field-name>
      <cmr-field-type>java.util.collection</cmr-field-type>
    </cmr-field>
  </ejb-relationship-role>
  ..... description of the other bean in the relation
</ejb-relation>
```

Get and Set Methods

- The entity bean must contain declarations for *get* and *set* methods for these relationship fields (as for other fields)

- For example

```
public abstract collection getSigners( )
```

```
public abstract void setSigners (collection BankCustomers)
```

- The EJB container will generate code for these methods



Transactions

- Transactions can be managed
 - Manually by the bean itself (bean-managed transactions) using standard JDBC or JTA calls
 - Bean programmer must provide statements to start and commit transactions
 - Automatically by the container (container-managed transactions)
 - Deployment descriptor contains declarative description of transaction attributes of each method



Transactions (continued)

- In container-managed transactions, the deployment descriptor must specify the transaction attributes of each method
- Attributes supported are
 - *Required*
 - *RequiresNew*
 - *Mandatory*
 - *NotSupported*
 - *Supports*
 - *Never*
- The semantics of these attributes will not be discussed now



Restrictions on Attributes

- For message-driven beans, only the *Required* and *NotSupported* attributes are allowed
 - If the bean has the *Required* attribute and aborts, the message is put back on the queue and the transaction will be called again
- For stateless session beans only *Requires*, *RequiresNew*, *Supports*, *Never*, and *NotSupported* are allowed
- For entity beans with container-managed persistence, only *Requires*, *RequiresNew*, and *Mandatory* are allowed.

Example of Deployment Descriptor

```
<container-transaction>  
  <method>  
    <ejb-name> ShoppingCart </ejbname>  
    <method-name> Checkout </method-name>  
  </method>  
  <trans-attribute> Required </trans-attribute>  
</container-transaction>
```



Two-Phase Commit

- The container also contains a transaction manager, which will manage a two-phase commit procedure, for both container-managed and bean-managed transactions.



Concurrency of Entity Beans

- A number of concurrently executing clients might request access to the same entity bean and hence the same row of a database table
- If that bean has been declared transactional, the concurrency is controlled by the container
 - If not, each client gets its own copy of the entity bean and the concurrency is controlled by the DBMS
 - For session beans and message-driven beans with bean-managed concurrency the bean programmer can specify the isolation level within the code for the bean
- The default J2EE implementation of container-managed concurrency is that each client gets its own copy of the entity bean and the underlying DBMS manages the concurrency



Another Implementation of Container-Managed Concurrency

- Some vendors of Web application servers offer other alternatives, such as optimistic concurrency control
 - DBMS executes at READ COMMITTED
 - All writes are kept in the entity bean until the transaction requests to commit
 - The intentions list
 - The validation check verifies that no entity the transaction read has been updated *since the read took place*.
 - Not the same validation check performed by the optimistic control discussed earlier
 - In that control, the validation check verifies that no entity the transaction read has been updated *anywhere in its read phase*
 - Both implementation provide serializability



Reusability of Enterprise Beans

- Part of the vision underlying enterprise beans is that they would be reusable components
 - Sam's Software Company sells a set of beans for shopping cart applications, including a ShoppingCartBean session bean
 - Joe's Hardware Store buys the beans
 - Instead of using the standard ShoppingCartBean, Joe's system uses a child of that bean, JoesShoppingCartBean that had been changed slightly to reflect Joe's business rules
 - Joe also changes the deployment descriptor a bit



Reusability of Enterprise Beans continued

- The implementation of Joe's system is considerably simplified
- Joe's programmers need be concerned mainly with Joe's business rules not with implementation details
- Joe's shopping cart application will run on any system using any Web application server that supports J2EE
 - Provided it does not use any proprietary extensions to J2EE