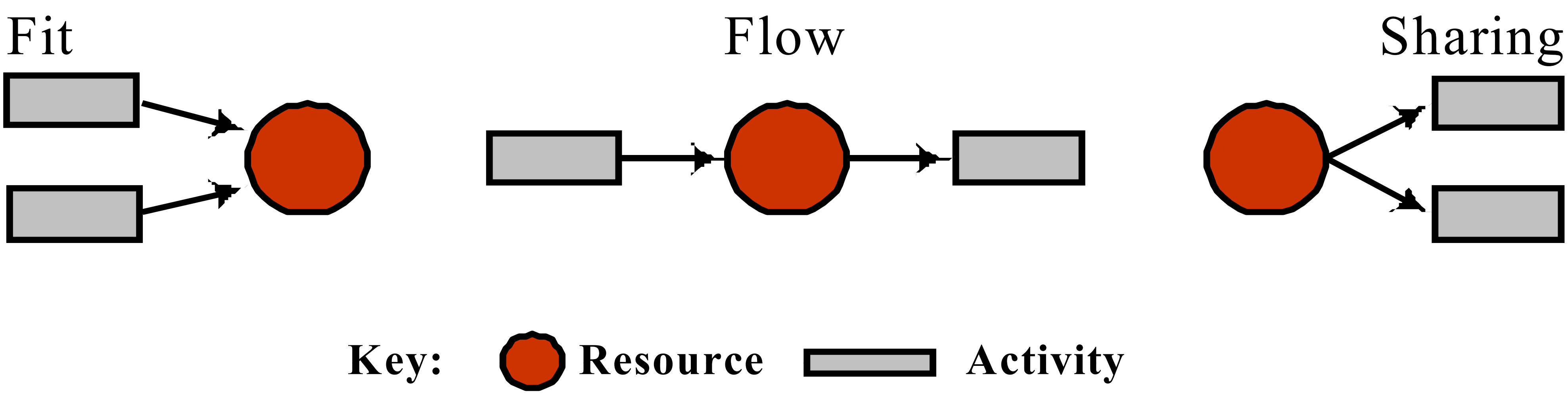
Chapter 14: Coordination

Coordination

- Objectives
  - To develop an overview on atomicity, a major factor in DS
  - To overview coordination middleware systems as examples
- Topics
  - Coordination and dependencies
  - Atomicity and failures
  - One-Phase Commit (1PC)
  - Two-Phase Commit (2PC)
  - Middleware Systems
    - Publish-subscribe paradigm
    - TIB/Rendezvous
    - Jini
    - Zookeeper

Coordination and Dependencies

- The act of coordinating, making different people or things work together for a goal or effect
- Three types



- Co-location
- Daily build
- Interface definitions

- Push-based
- Pull-based
- Market

- First-come first-served
- Manager decides
- Bidding

Atomicity

- Consider a replicated database
  - Running on a distributed system with reliable multicasting
  - Updates are multicast to all replicas and the system guarantees that they are delivered in order

Is this sufficient?

- Nasty scenario:
  - A message is multicast reliably to all replicas and is delivered to the application layer (the database)
  - One replica crashes, while performing the update
  - When it recovers it is in an inconsistent state!
- Atomicity is the key and needed
  - All commit or all abort!
    - Guarantee that an operation is completed at all participants or at none

Example

- Transfer money from bank A to bank B
  - Debit A, credit B, tell client “OK”

- This process wants either both to perform the transfer or neither to do it
  - Never want only one side to act!
  - Better if nothing happens!

- Goal: Atomic Commit Protocol

Two Major Aspects of Atomicity

- Serializability
  - Series of operations requested by users
  - Outside observer sees each of them complete atomically in some complete order
  - Requires support for locking
- Recoverability
  - Each operation executes completely or not at all
  - No partial results exist – “all-or-nothing”
- Prerequisites
  - Serializability applies synchronization
    - Logical and vector clocks (see earlier module)
  - Recoverability applies a distributed protocol (to follow)

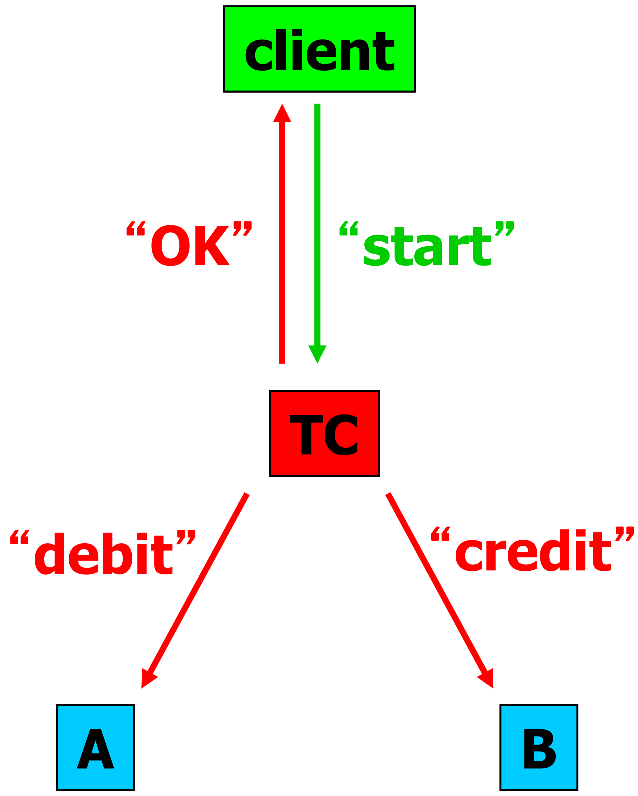
Difficulty of Atomic Commits

- Example
  - A tells B: “I’ll commit, if you commit”
  - A hears no reply from B
  - Now what?
  - Neither party can make final decision!

- Impacts from
  - Communication systems errors
  - Communication protocols’ unreliability
  - Distributed systems’ hardware failures
  - Application misbehaviors

One-Phase Commit (1PC) Protocol

- Create a Transaction Coordinator (TC)
  - A single authoritative entity
- Four entities
  - Client, TC, Bank A, Bank B
- Operation
  - Client sends “start” to TC
  - TC sends “debit” to A
  - TC sends “credit” to B
  - TC reports “OK” to client



Failure Scenarios

- Not enough money in A's bank account
  - A does not commit, B does
- B's bank account no longer exists
  - A commits, B does not
- One network link (of A or B) is broken
  - One commits, the other does not
- One of A or B has crashed
  - One commits, the other does not
- TC crashes between sending to A and B
  - A commits, B does not

Desirable Properties of an Atomic Commit

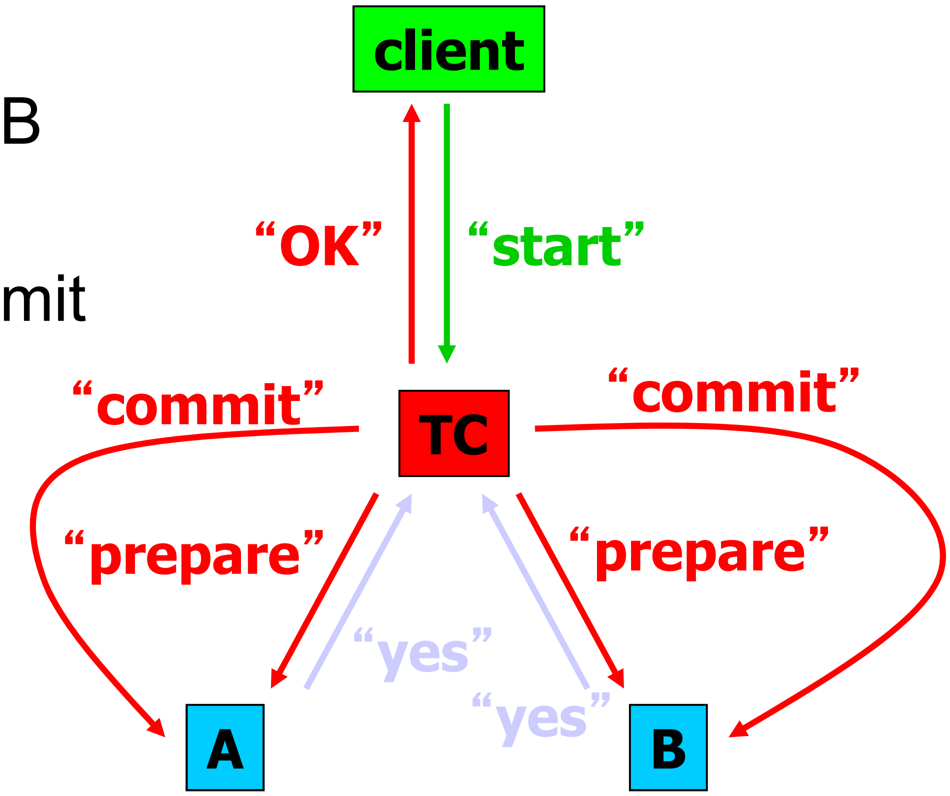
- TC, A, and B have separate notions of committing

- Correctness
  - If one commits, no one aborts
  - If one aborts, no one commits

- Liveness (in a sense related to performance)
  - If no failures and A and B can commit, then commit
  - If failures, come to “some” conclusion as soon as possible

Two-Phase Commit (2PC) Protocol

- Same entities as in 1PC
- Operation
  - Client “starts” and TC sends “prepare” messages to A and B
  - A and B respond, saying whether they’re willing to commit
  - If both say “yes,” TC sends “commit” messages
  - If either says “no,” TC sends “abort” messages
  - A and B “decide to commit”, if they receive a commit message.



Correctness and Liveness of 2PC

- Why is the 2PC protocol correct (*i.e.*, safe)?
  - Knowledge centralized at TC about willingness of A and B to commit
  - TC enforces both and must agree for either to commit

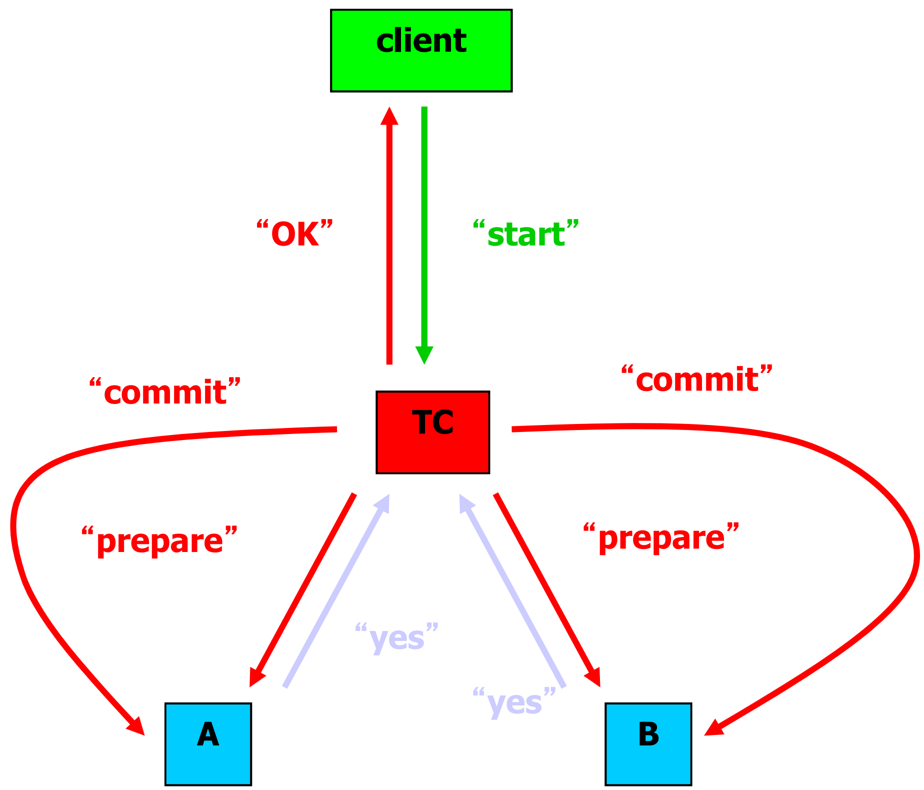
- Does the previous protocol always complete (*i.e.*, does it exhibit liveness)?
  - No!
  - What if nodes crash or messages get lost?

Liveness Problems in 2PC

- Timeout
  - Host is up, but does not receive message it expects
  - Maybe other host crashed, maybe network dropped message, maybe network is down
  - Usually cannot distinguish these cases, so solution must be correct for all!
- Reboot
  - Host crashes, reboots, and must “clean up”
  - *I.e.*, want to wind up in correct state despite reboot

Solution to Liveness Problems

- Solution
  - Introduce timeouts
  - Take appropriate actions
    - But be conservative to preserve correctness!



- Where in the protocol do hosts wait for messages?
  - TC waits for “yes”/”no” from A and B
  - A and B wait for “commit”/”abort” from TC

Transaction Coordinator Times Outs

- Proceed with a decision when TC waits too long for *yes/no*

- TC has not yet sent any “commit” messages
  - So it can safely abort
    - Send “abort” messages

- This preserves safety, but sacrifices liveness
  - Perhaps both A, B prepared to commit, but a “yes” message was lost
  - Could have committed, but TC unaware!
  - Thus, TC is conservative

A (or B) Times Out

- If B voted "no"
  - It can unilaterally abort
  - TC will never send "commit" in this case
- If B voted "yes"
  - B cannot unilaterally abort
    - TC might have received "yes" from both, sends "commit" to A, then crashed before sending "commit" to B
    - Result: A would commit, B would abort: incorrect (unsafe)!
  - B cannot unilaterally commit
    - A might have voted "no"
- If B voted "yes"
  - Either B keeps waiting forever (not a solution) or …

Better plan?

Termination Protocol (B with “yes” Vote)

- B directly contacts A: sends “status” request to A asking, if A knows whether the transaction should commit
  - If A received “commit” or “abort” from TC:
    B decides same way (cannot disagree with TC)
  - If A hasn’t voted anything yet: B and A both abort
    - TC can’t have decided “commit”; it will eventually hear from A or B
  - If A voted “no”: B and A both abort
    - TC can’t have decided “commit”
  - If A voted “yes”: no decision possible, keep waiting
    - TC might have timed out, aborted, and replied to client
  - If no reply from A: no decision possible, wait for TC

Termination Protocol Behavior

- Some timeouts can be resolved with a guaranteed correctness (safety)
- Sometimes, though, A and B must block
  - Especially when TC fails or TC's network connection fails
- Remember
  - TC is entity with centralized knowledge of A's and B's state!

Problem: Crash-and-Reboot

- Cannot back out if commit once decided
  - Suppose TC crashes just after deciding and sending a "commit"
    - What if "commit" message to A or B is lost?
  - Suppose A and/or B crash just after sending "yes"
    - What if "yes" message to TC is lost?

- If A or B reboots, they do not remember saying "yes", which leads to big trouble!
  - Might change mind after reboot
  - Even after everyone reboots, may not be able to decide!

Solution: Persistent State (1)

- Storing state in non-volatile memory (*e.g.*, a disk)
  – If all nodes know their pre-crash state, they can use the previously described termination protocol
  – A and B can also ask TC, which may still remember if it committed

- The order of store and send
  – Write disk
  – Send “yes” message if A/B or “commit” if TC?
  – Or vice-versa?

Solution: Persistent State (2)

- Can a message be send before writing the disk?
  - Might then reboot between sending and writing, and change mind after reboot
  - *E.g.,* B might send “yes”, then reboots, then decides “no”

- Thus, write disk before sending message?
  - For TC, write “commit” to disk before sending
  - For A/B, write “yes” to disk before sending

Revised Recovery Protocol

- TC: after reboot, if no “commit” on disk, abort
  – No “commit” on disk means no “commit” messages had been sent: safe
- A/B: after reboot, if no “yes” on disk, abort
  – No “yes” on disk means that no “yes” messages had been sent, so no one could have committed; safe
- A/B: after reboot, if “yes” on disk, use ordinary termination protocol
  – Might block!
- If everyone rebooted and is reachable, can still decide!
  – Just look at whether TC has stored a “commit” on disk

Summary of 2PC Properties

- "Prepare" and "commit" phases
  - Two-Phase Commit (2PC)
- Properties:
  - Safety: All hosts that decide reach the same decision
  - Safety: No commit unless everyone says "yes"
  - Liveness:
    - If no failures occur and all say "yes," then commit
    - If failures occur, repair, wait long enough, eventually take a decision
- Remember: Consensus not (always) possible!
  - Theorem [Fischer, Lynch, Paterson, 1985]: "No distributed asynchronous protocol can correctly agree (provide both safety and liveness) in presence of crash-failures (*i.e.*, if failures are not repaired)"

Time and Reference Coupling

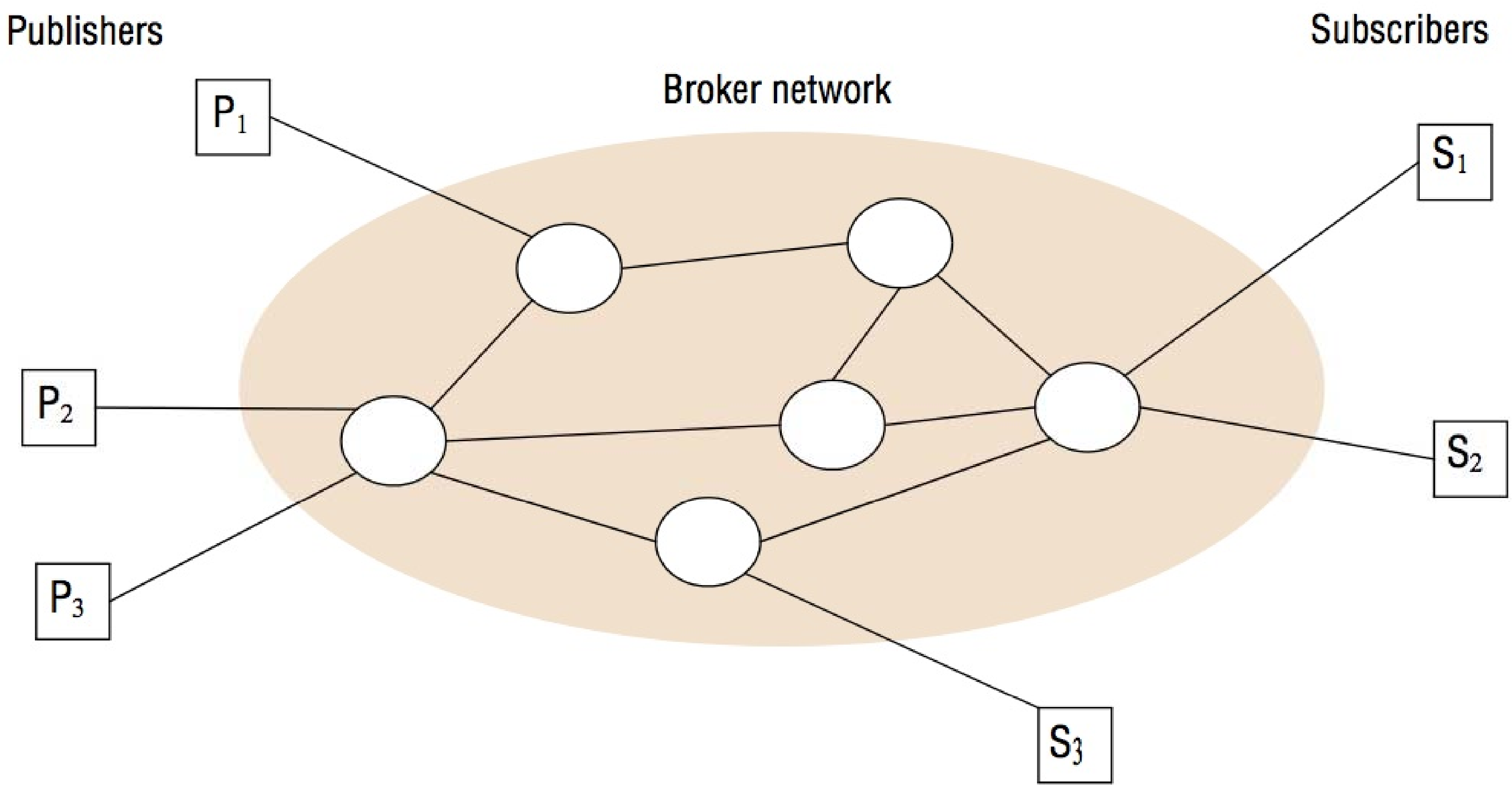
- Views of a distributed system

	Time-coupled	Time-uncoupled
Named/ Reference	Communications directed toward a defined receiver that must exist at the same time *Examples: RPC, RMI*	Communications directed toward a defined receiver that exists at some time *Examples: Message based systems* A
Unnamed	Unknown name of receiver who must exist at the same time *Examples: Multicast* B	Unknown name of receiver who should exist at some time Examples: *Tuple Spaces, Zookeeper* C

A Network of Brokers

A

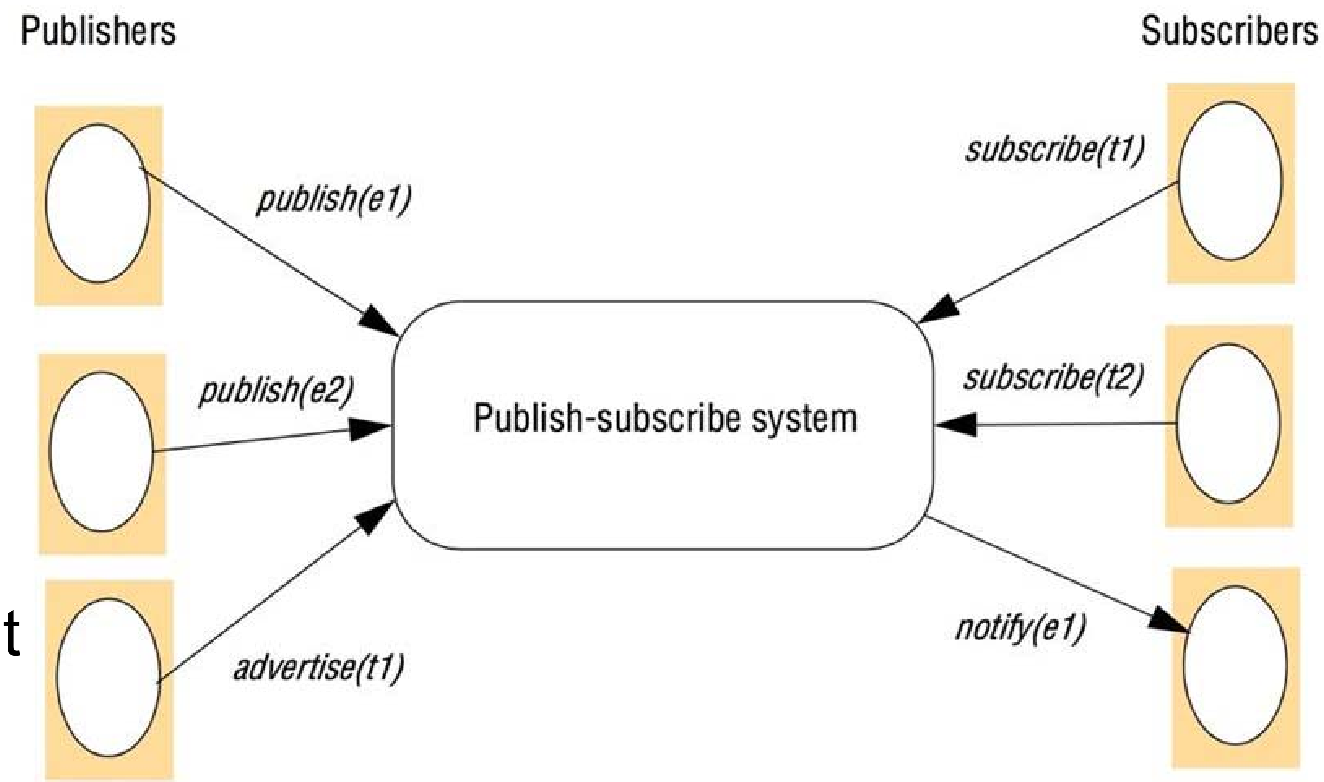
- Message-based systems
  – Publishers and subscribers interconnected by broker network



Publish-subscribe Paradigm

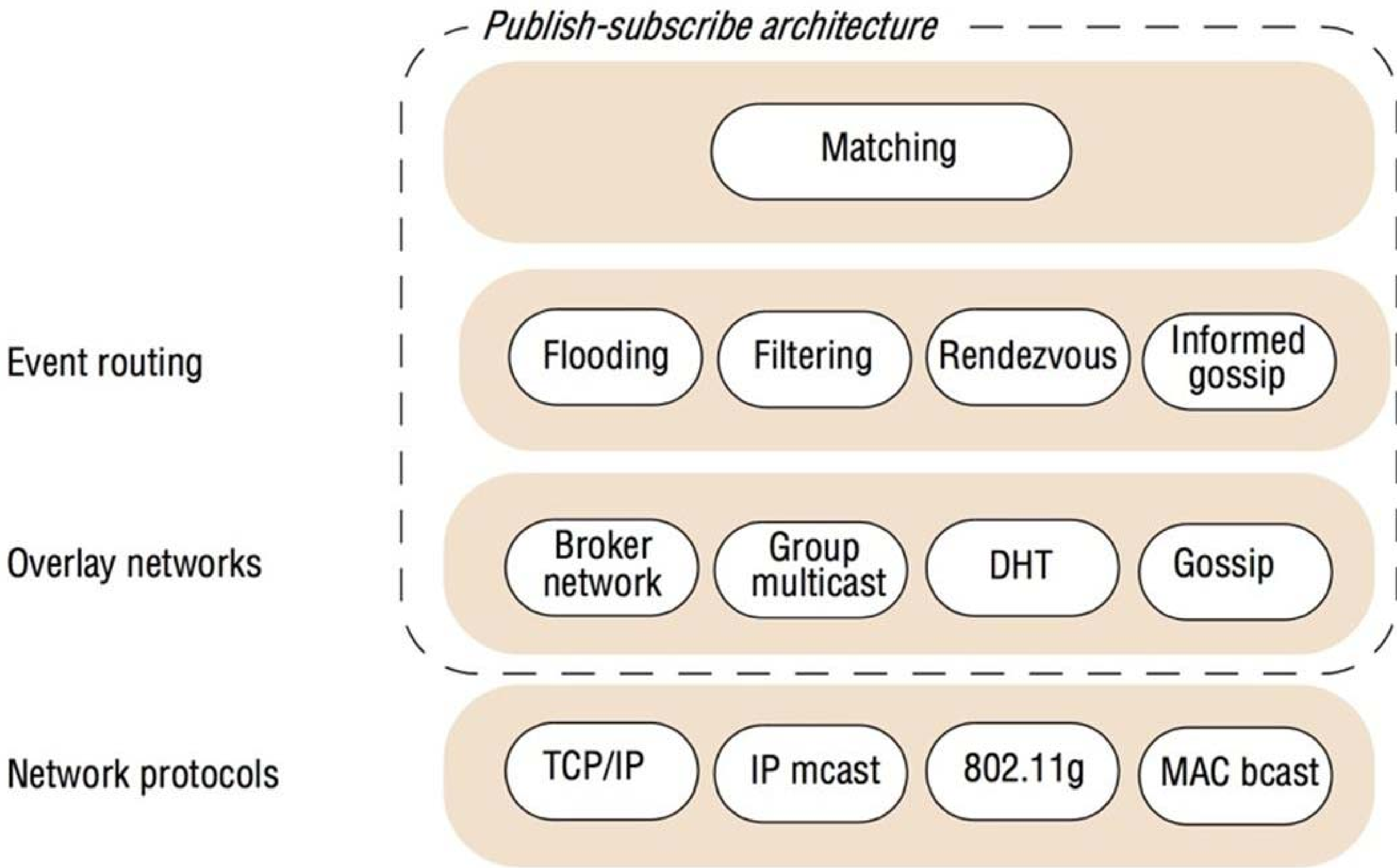
A

- (Event) Type-based subscription
  – Can organize events in hierarchy
- Channel-based subscription
  – Gets all events of that channel
- Content-based subscription
  – Filters events based on content



Architecture of Publish-Subscribe Systems

A



TIB/Rendezvous

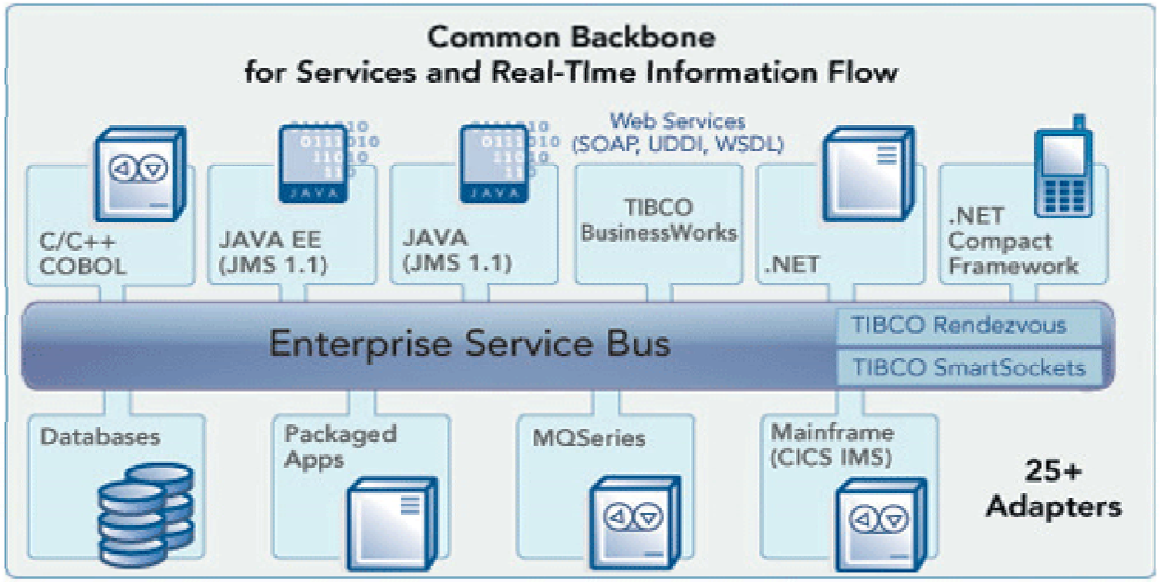
A

- Message Bus for Enterprise Application Integration
- Major design goals
  - Application-dependent communication system
  - Messages are self-describing
  - Processes should be referentially uncoupled
- Addresses are given as
  - Subject names
  - Inbox names
- Communication primitives
    - Send
    - SendRequest
    - SendReply

TIB: The Information Bus

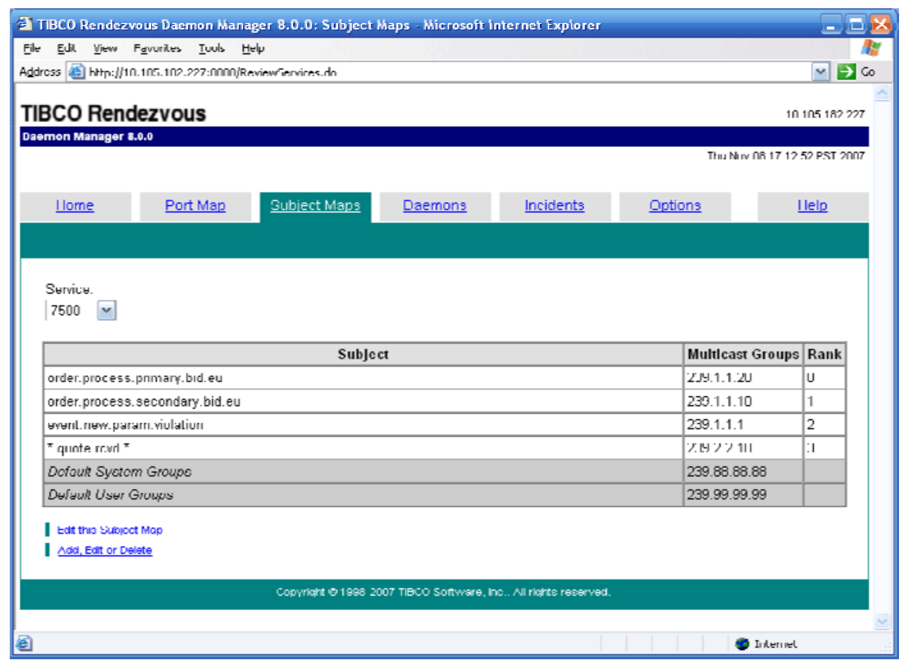
TIBCO – A TIB Instance

A



https://www.tibco.com/products/tibco-rendezvous

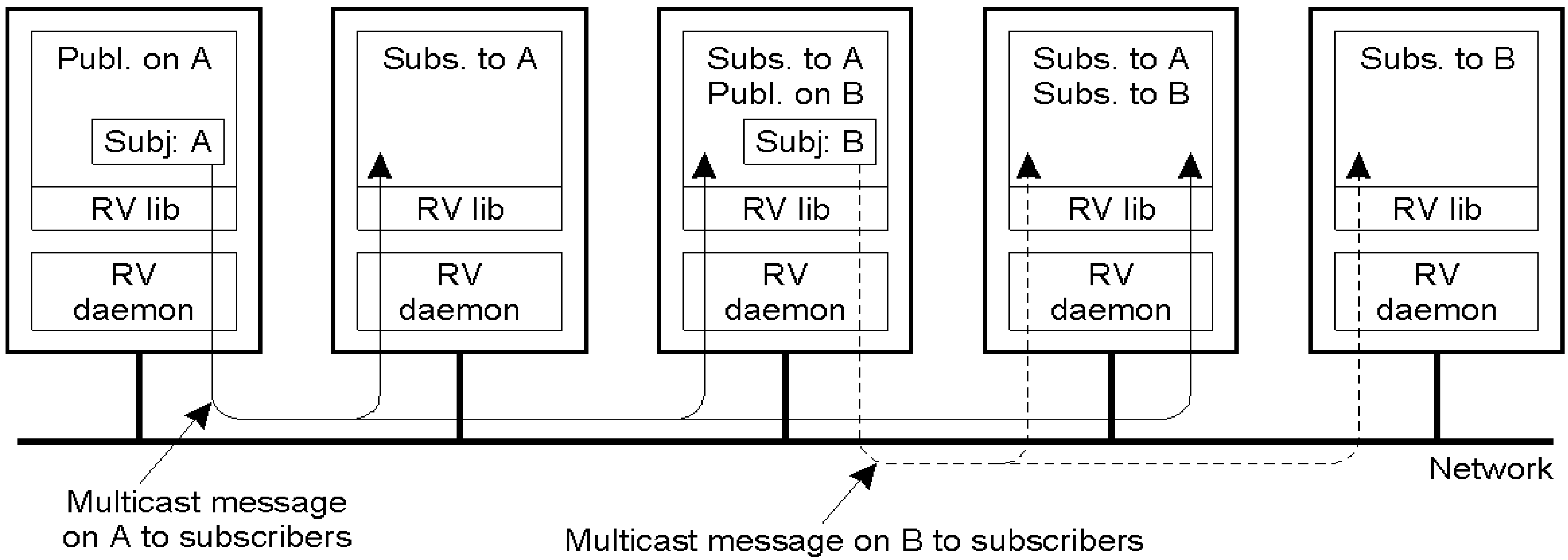
- Message-oriented Middleware
  - High-speed data distributions
  - Fully distributed daemon-based peer-to-peer architecture
  - No single point-of-failure



Coordination Model (1)

A

- Publish/subscribe system as in TIB/Rendezvous

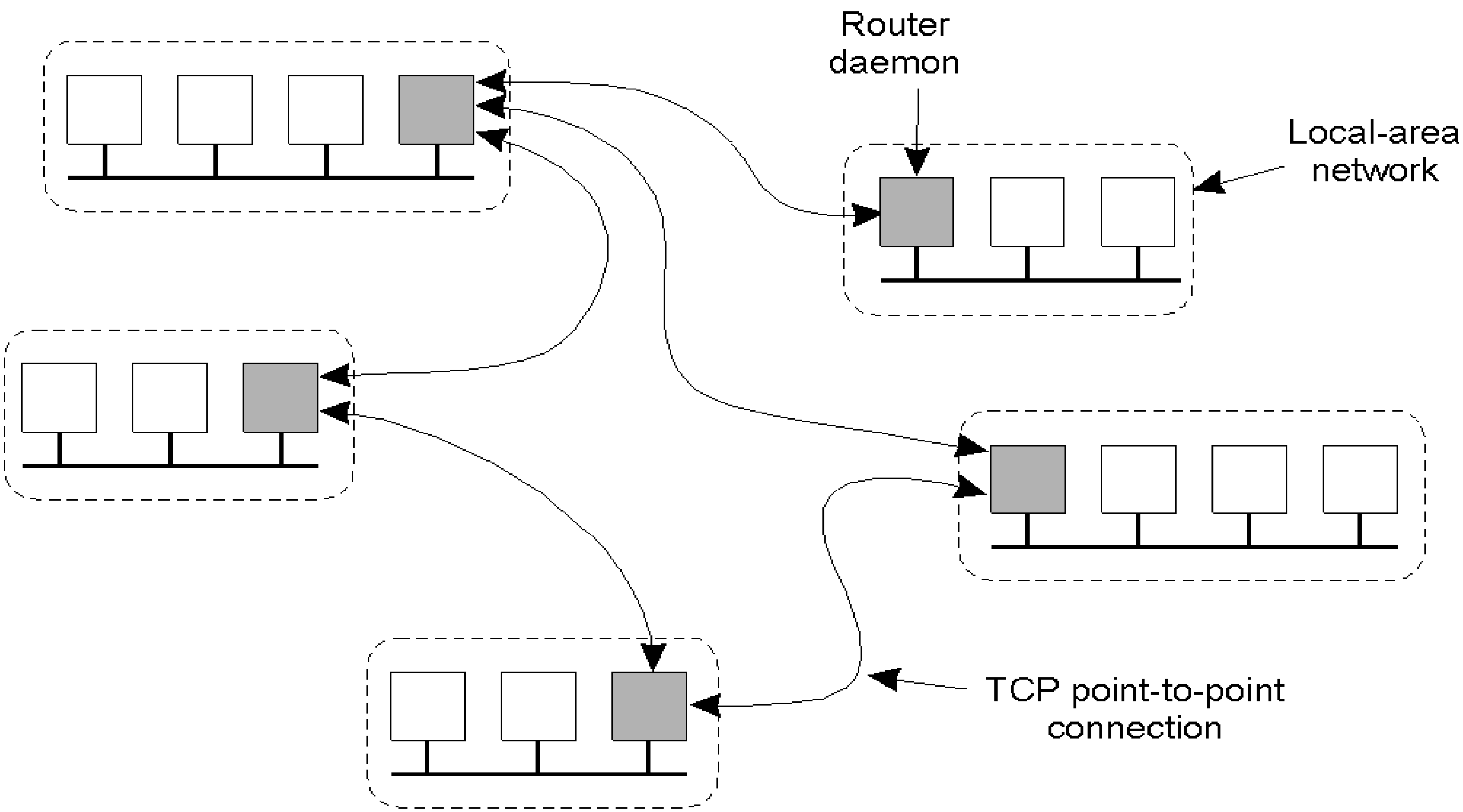


Publ.: Publisher Subs.: Subscriber RV: Rendezvous

Coordination Model (2)

A

- Overall architecture of a wide-area TIB/Rendezvous system



Basic Messaging

A

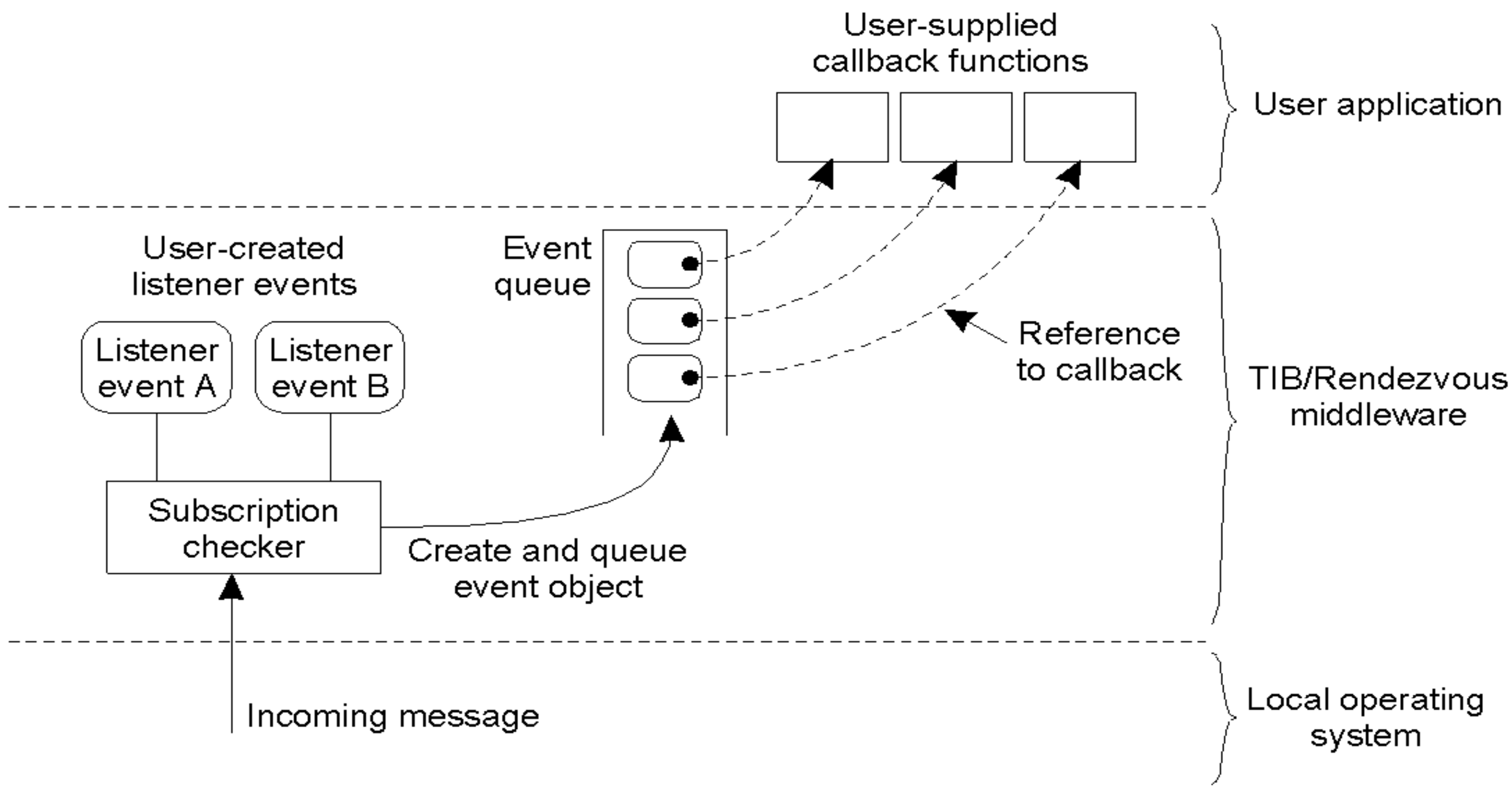
- Example
  - Attributes of a TIB/Rendezvous message field

Attribute	**Type**	**Description**
Name	String	The name of the field, possibly NULL
ID	Integer	A message-unique field identifier
Size	Integer	The total size of the field (in bytes)
Count	Integer	The number of elements in the case of an array
Type	Constant	A constant indicating the type of data
Data	Any type	The actual data stored in a field

Events (1)

A

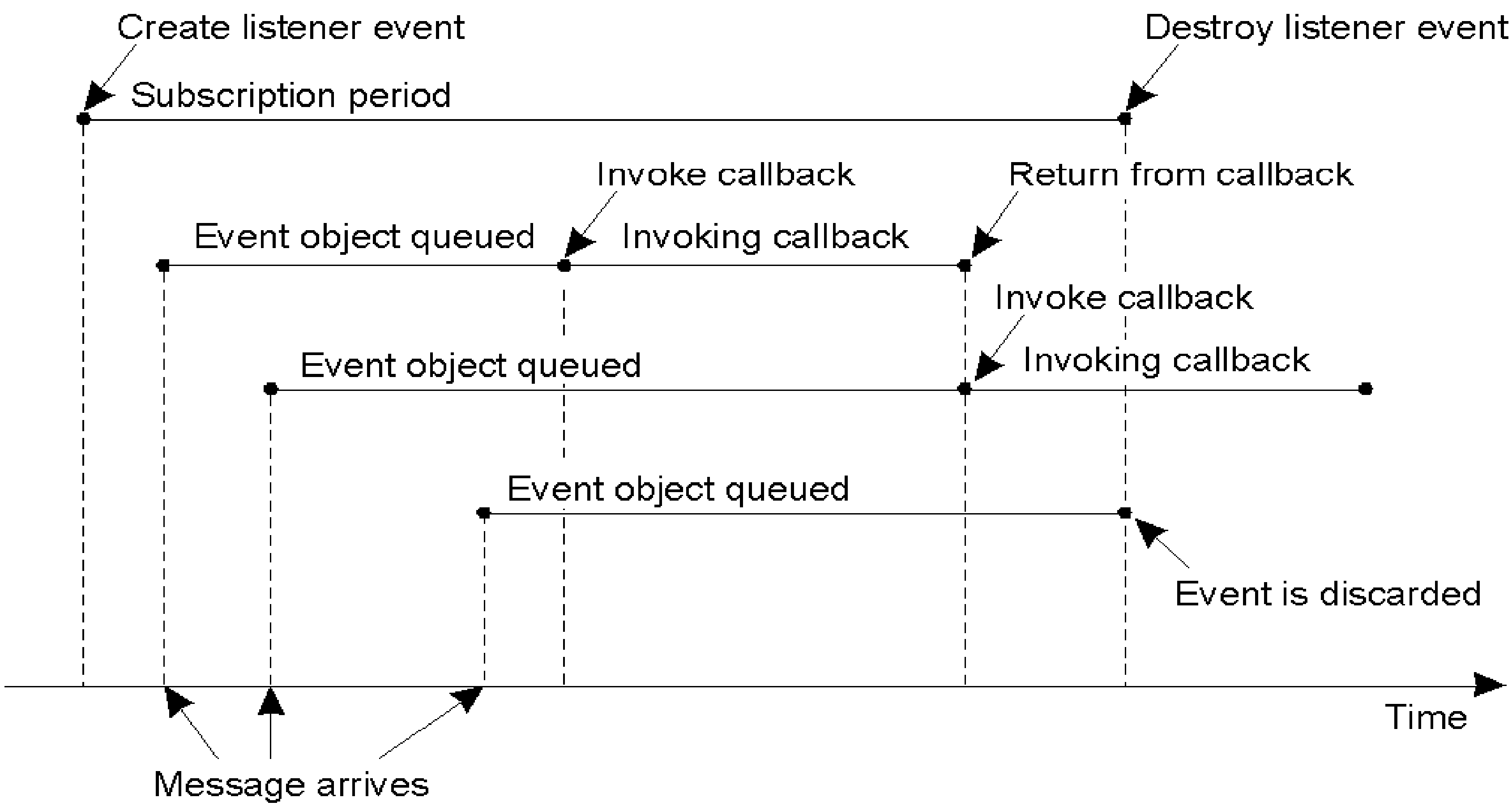
- Processing listener events for subscriptions in TIB/Rendezvous



Events (2)

A

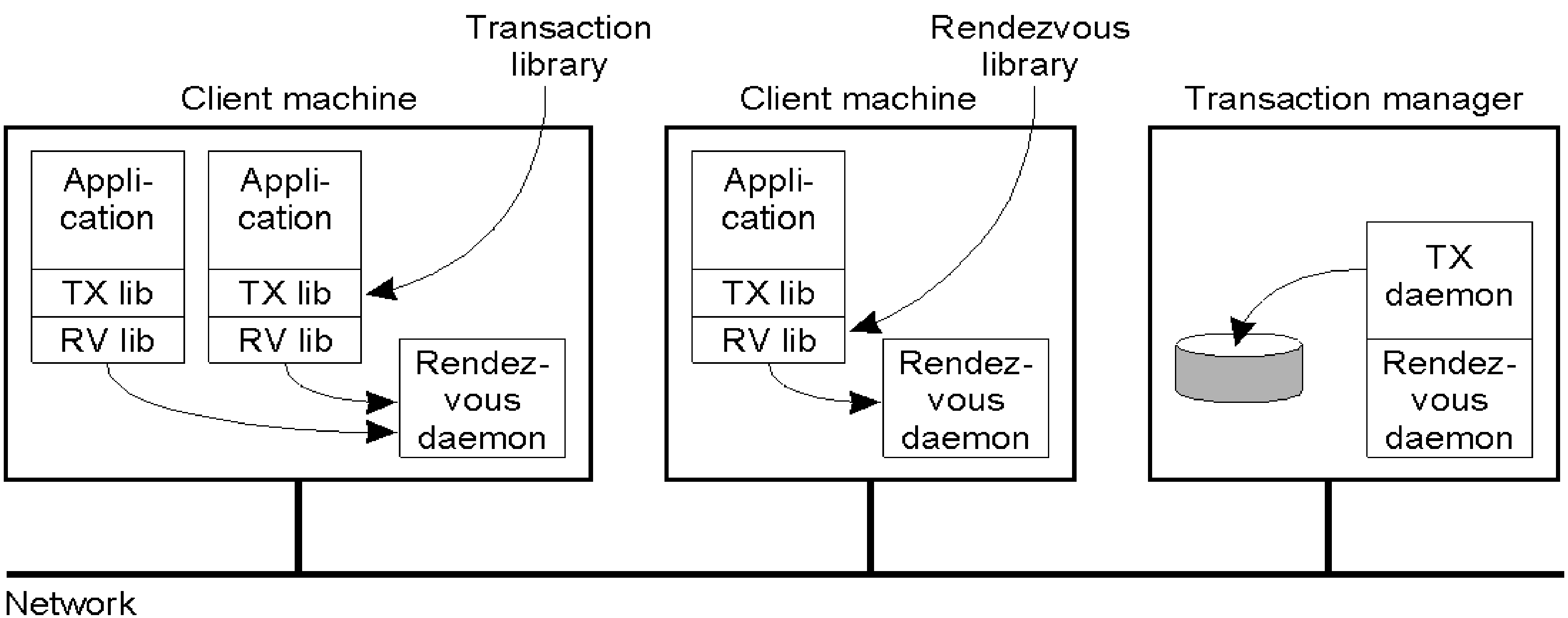
- Processing incoming messages in TIB/Rendezvous



Synchronization

A

- Organization of transactional messaging as a separate layer in TIB/Rendezvous

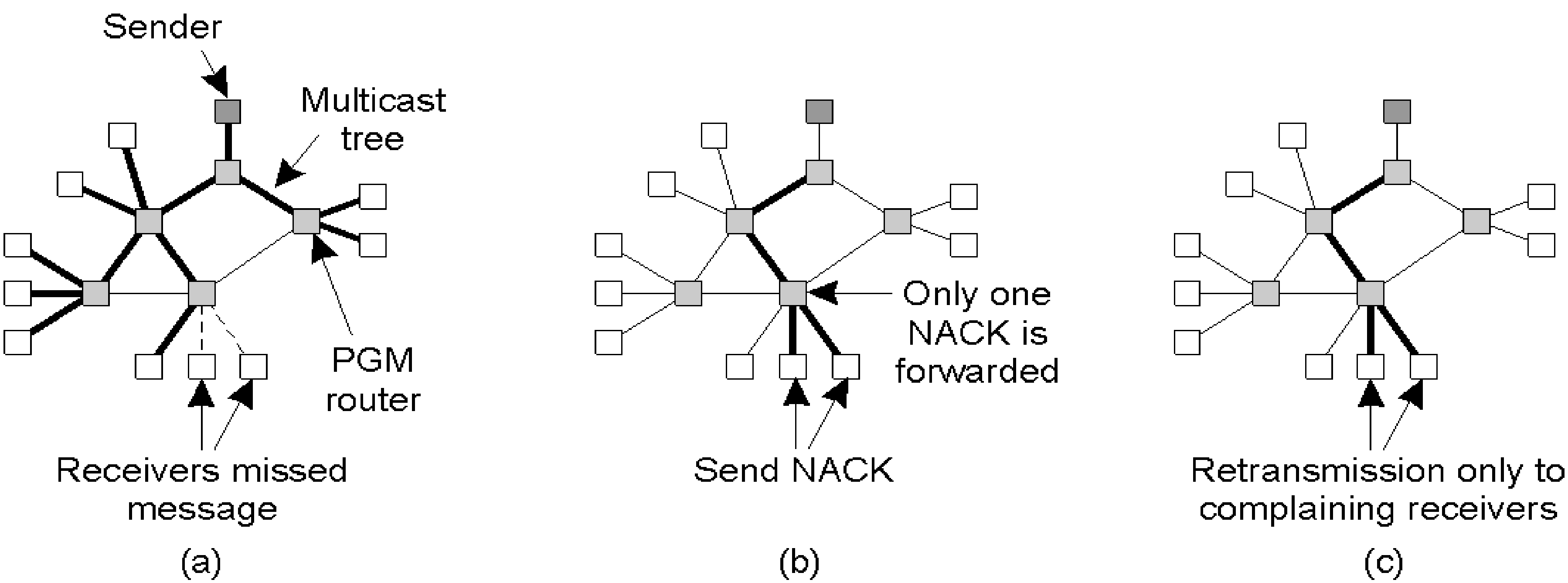


Reliable Communications

B

General Multicast

a) A message is sent along a multicast tree
b) A router will pass only a single NACK for each message
c) A message is retransmitted only to receivers that have asked for it

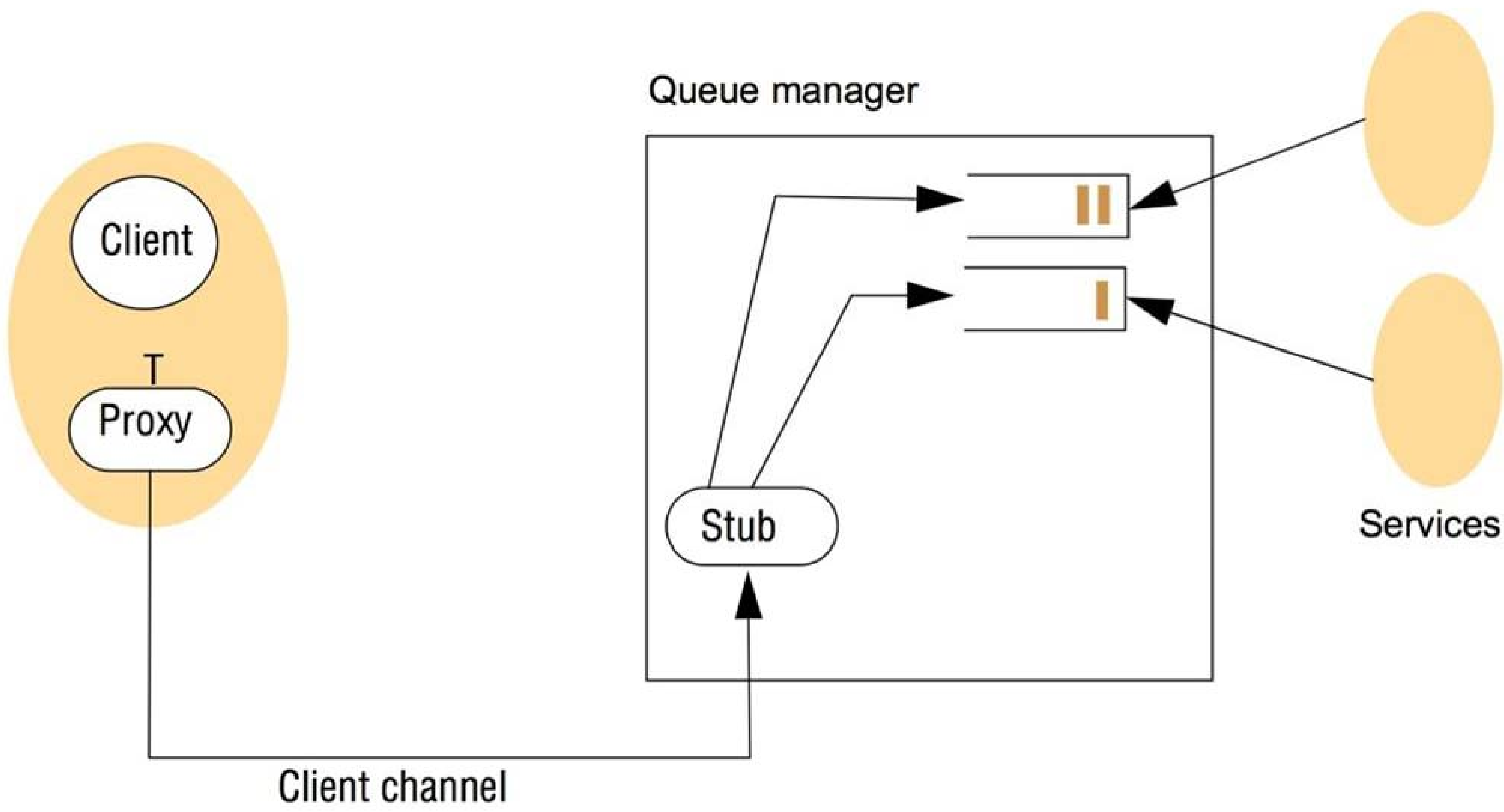


ifi

A Networked Topology: WebSphere MQ

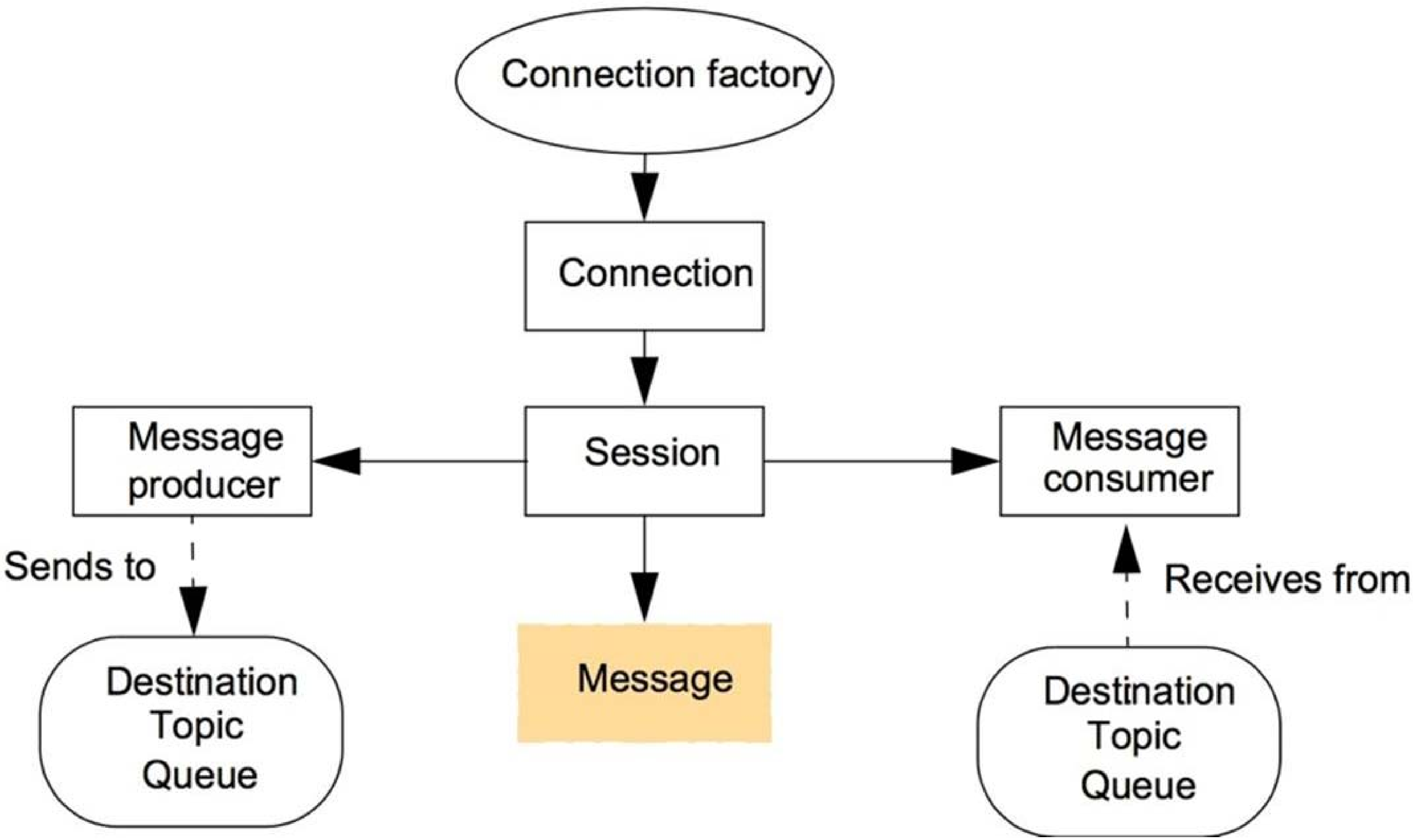
C

- MQ: Message Queue



Java Messaging Service (JMS)

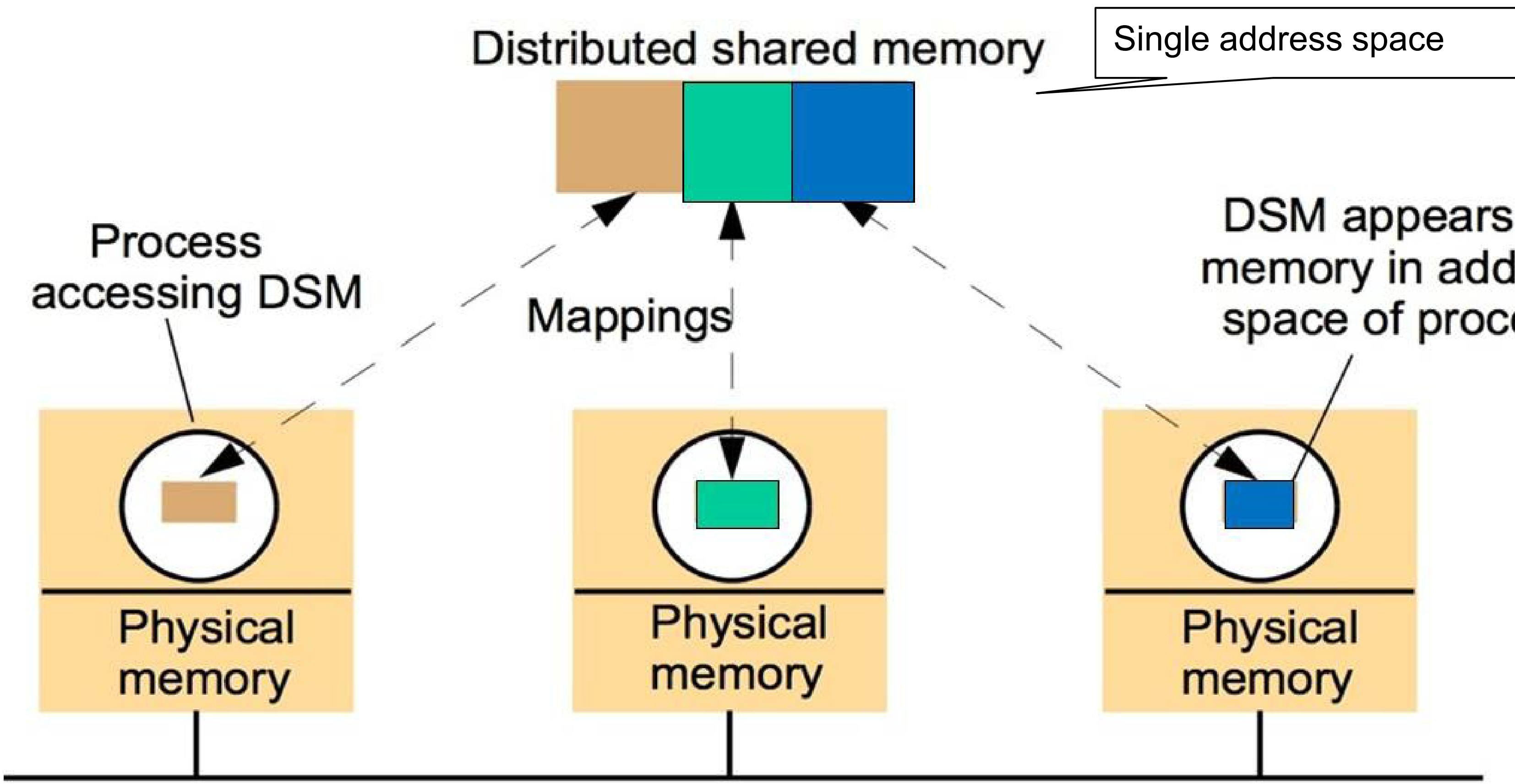
C



Distributed Shared Memory Abstraction

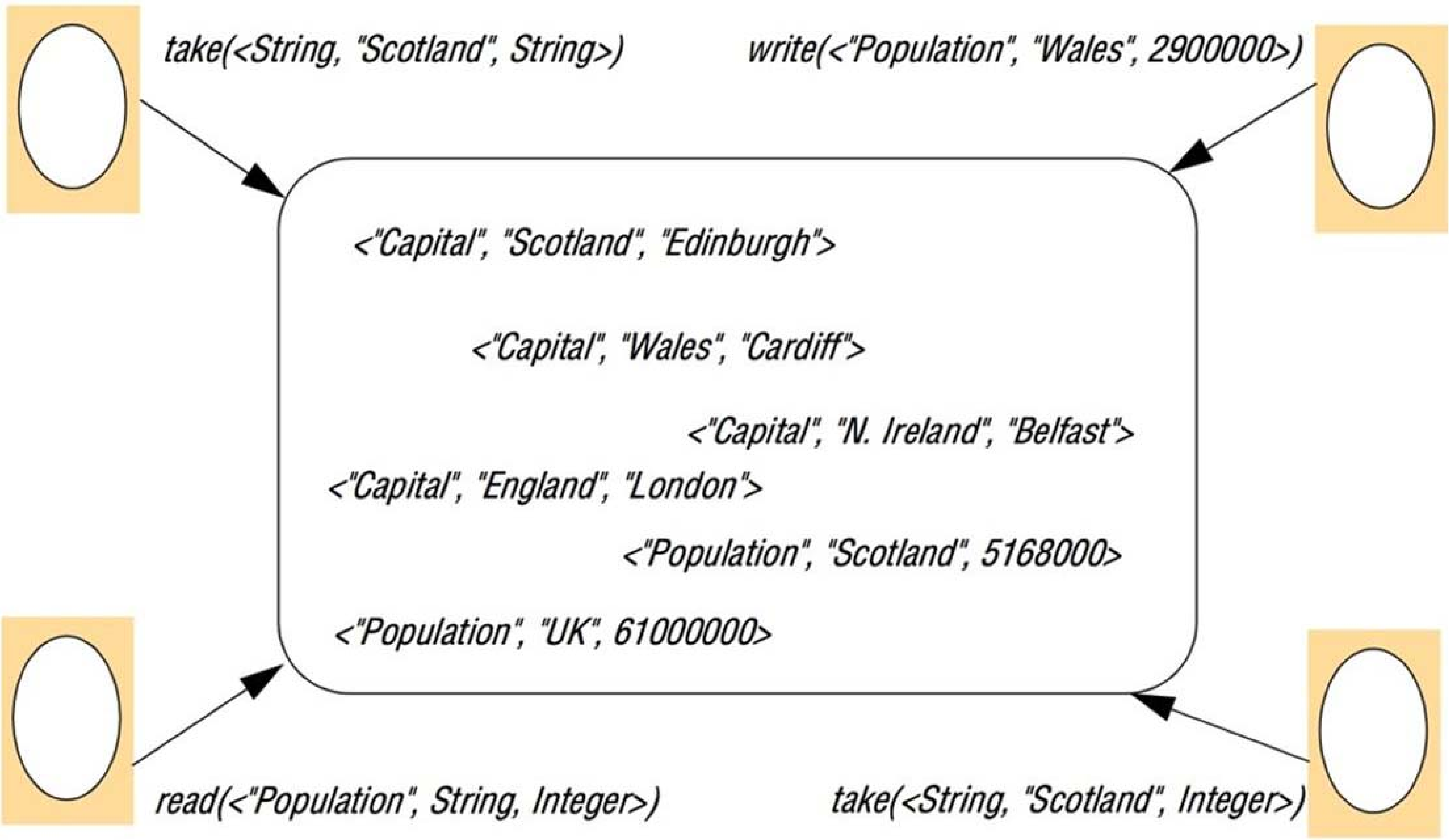
C

- Indirect communication, based on bytes



Tuple Space Abstraction

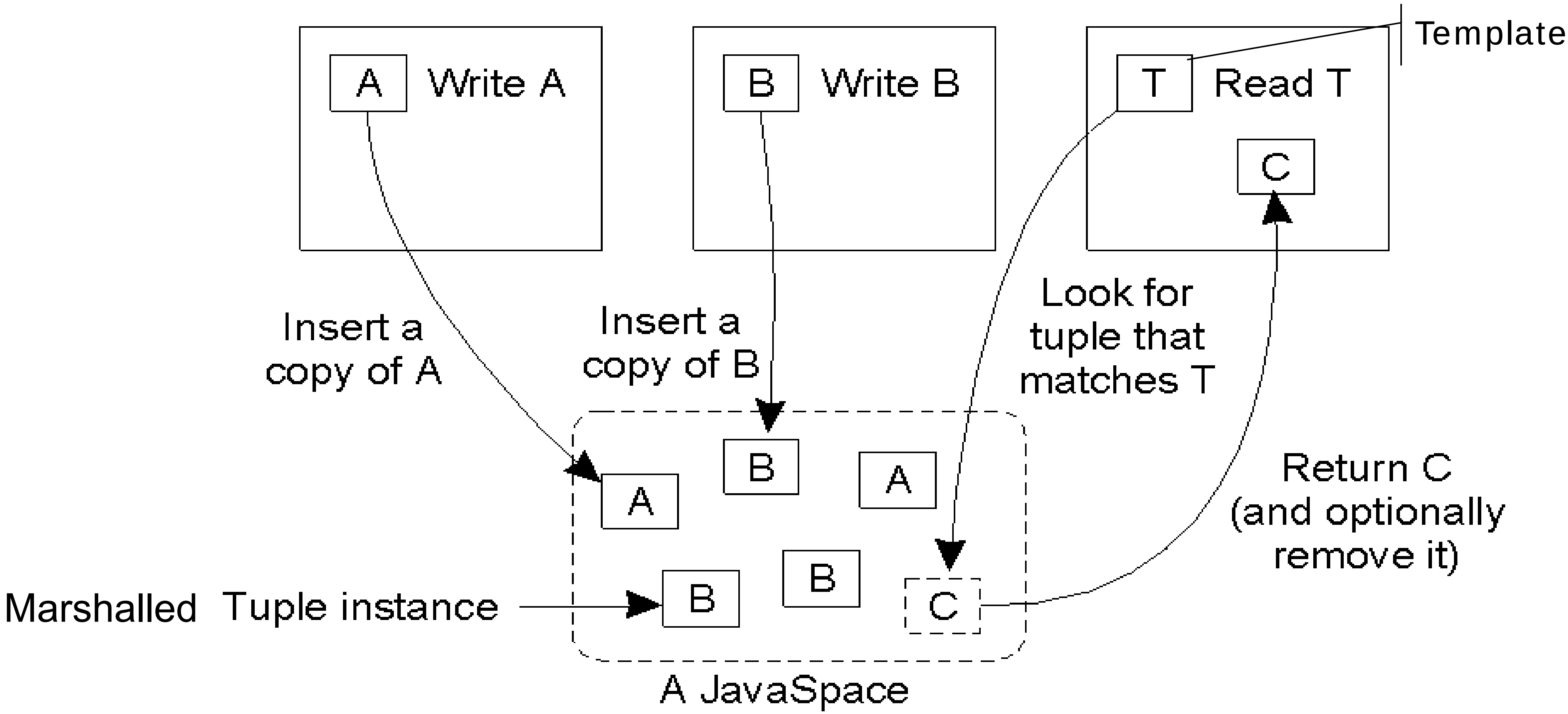
C



Jini Overview

C

- General organization of a JavaSpace in Jini

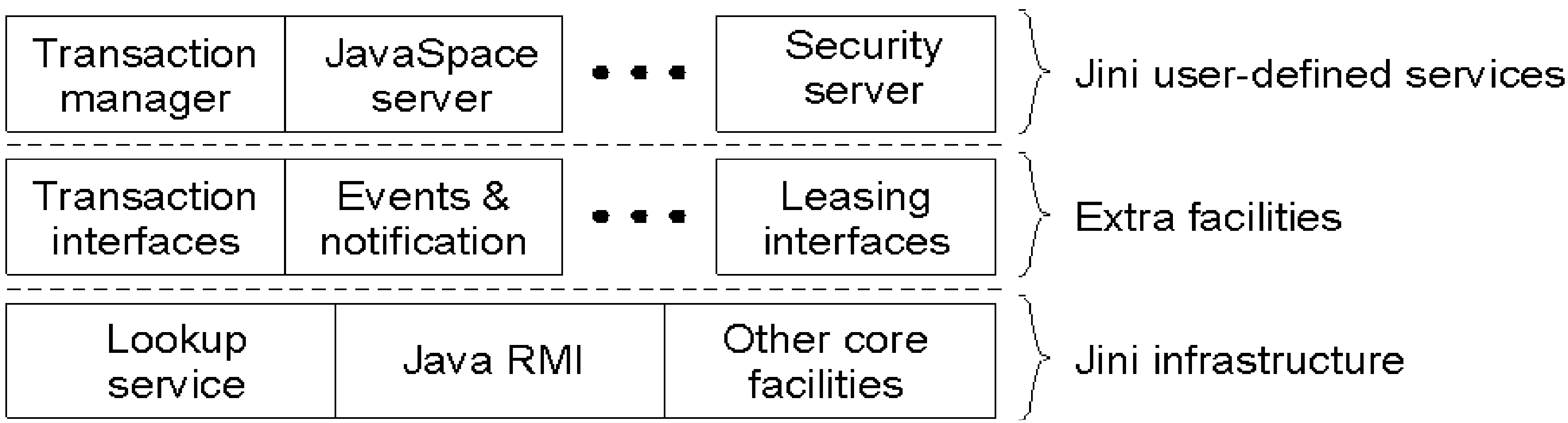


CDK, Fig. 6.20

Jini Architecture

C

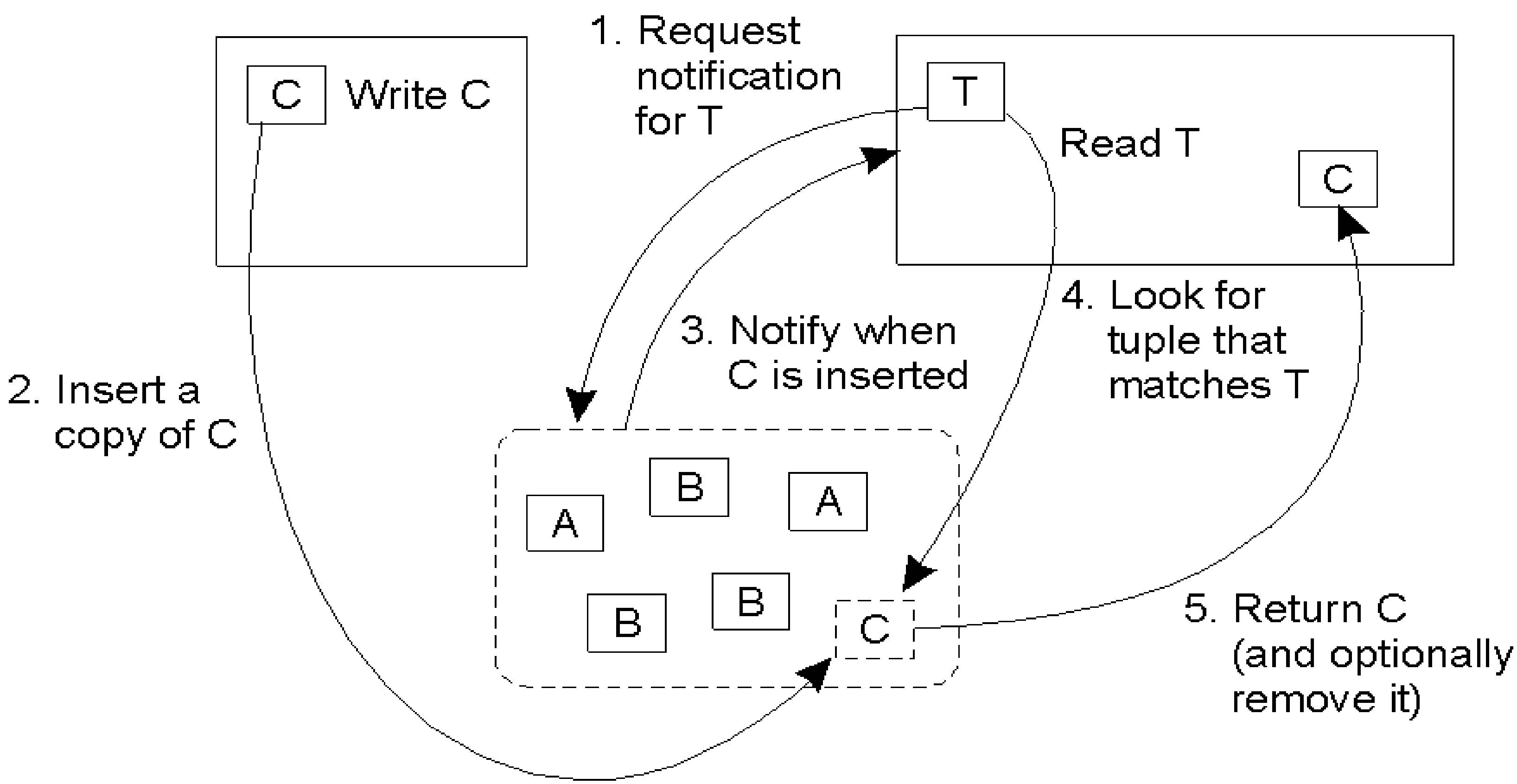
- Layered architecture of Jini



Communication Events

C

- Using events in combination with a JavaSpace and Leases

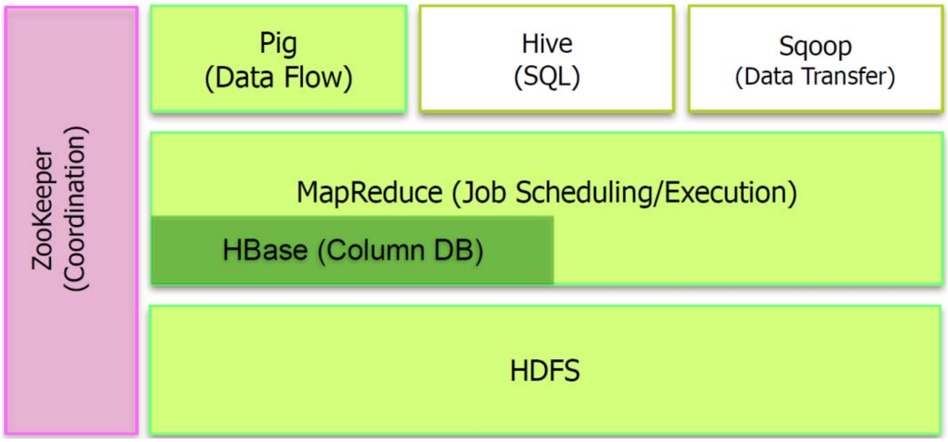


Performance and Reliability Goals

C

- Servers with replication
- Caching on client side
- Zookeeper
  - Distributed coordination service
    - Publish-and-subscribe mechanism at hand (“watch”)
    - Hierarchical, scalable, atomicity, reliability
    - Sequential consistency
  - Uses Paxos to solve consensus

Example: Zookeeper in Hadoop Framework



ifi