Cloud Computing Patterns

Foundations and Introduction

Tutorial at SummerSoC 2013 (1 July – 6 July, 2013, Hersonissos, Crete, Greece)

Christoph Fehling, Prof. Dr. Frank Leymann
Institute of Architecture of Application Systems (IAAS)

Universität Stuttgart
Universitätsstr. 38
70569 Stuttgart
Germany

Phone +49-711-7816 470
Fax +49-711-7816 472
e-mail Leymann| Fehling
@iaas.uni-stuttgart.de

Christoph Fehling · Frank Leymann · Ralph Retter · Walter Schupeck · Peter Arbitter

Cloud Computing Patterns

Fundamentals to Design, Build, and Manage Cloud Applications

This book provides CIOs, software architects, project managers, developers, and cloud strategy initiatives with a set of architectural patterns that offer nuggets of advice on how to achieve common cloud computing-related goals. *The cloud computing patterns* capture knowledge and experience in an abstract format that is independent of concrete vendor products. Readers are provided with a toolbox to structure cloud computing strategies and design cloud application architectures. By using this book cloud-native applications can be implemented and best suited cloud vendors and tooling for individual usage scenarios can be selected. *The cloud computing patterns* offer a unique blend of academic knowledge and practical experience due to the mix of authors. Academic knowledge is brought in by Christoph Fehling and Professor Dr. Frank Leymann who work on cloud research at the University of Stuttgart. Practical experience in building cloud applications, selecting cloud vendors, and designing enterprise architecture as a cloud customer is brought in by Dr. Ralph Retter who works as an IT architect at T Systems, Walter Schupeck, who works as a Technology Manager in the field of Enterprise Architecture at Daimler AG, and Peter Arbitter, the former head of T Systems' cloud architecture and IT portfolio team and now working for Microsoft.

Voices on Cloud Computing Patterns

Cloud computing is especially beneficial for large companies such as Daimler AG. Prerequisite is a thorough analysis of its impact on the existing applications and the IT architectures. During our collaborative research with the University of Stuttgart, we identified a vendor-neutral and structured approach to describe properties of cloud offerings and requirements on cloud environments. The resulting Cloud Computing Patterns have profoundly impacted our corporate IT strategy regarding the adoption of cloud computing. They help our architects, project managers and developers in the refinement of architectural guidelines and communicate requirements to our integration partners and software suppliers.

Dr. Michael Gorriz – CIO Daimler AG

Ever since 2005 T-Systems has provided a flexible and reliable cloud platform with its "Dynamic Services". Today these cloud services cover a huge variety of corporate applications, especially enterprise resource planning, business intelligence, video, voice communication, collaboration, messaging and mobility services. The book was written by senior cloud pioneers sharing their technology foresight combining essential information and practical experiences. This valuable compilation helps both practitioners and clients to really understand which new types of services are readily available, how they really work and importantly how to benefit from the cloud.

Dr. Marcus Hacke – Senior Vice President, T-Systems International GmbH

This book provides a conceptual framework and very timely guidance for people and organizations building applications for the cloud. Patterns are a proven approach to building robust and sustainable applications and systems. The authors adapt and extend it to cloud computing, drawing on their own experience and deep contributions to the field. Each pattern includes an extensive discussion of the state of the art, with implementation considerations and practical examples that the reader can apply to their own projects.
By capturing our collective knowledge about building good cloud applications and by providing a format to integrate new insights, this book provides an important tool not just for individual practitioners and teams, but for the cloud computing community at large.

Kristof Kloeckner – General Manager, Rational Software, IBM Software Group

Computer Science

ISBN 978-3-7091-1567-1

EXTRA MATERIALS extras.springer.com

▶ springer.com

Fehling · Leymann
Retter · Schupeck · Arbitter

Cloud Computing Patterns

Christoph Fehling · Frank Leymann
Ralph Retter · Walter Schupeck
Peter Arbitter

Cloud Computing Patterns

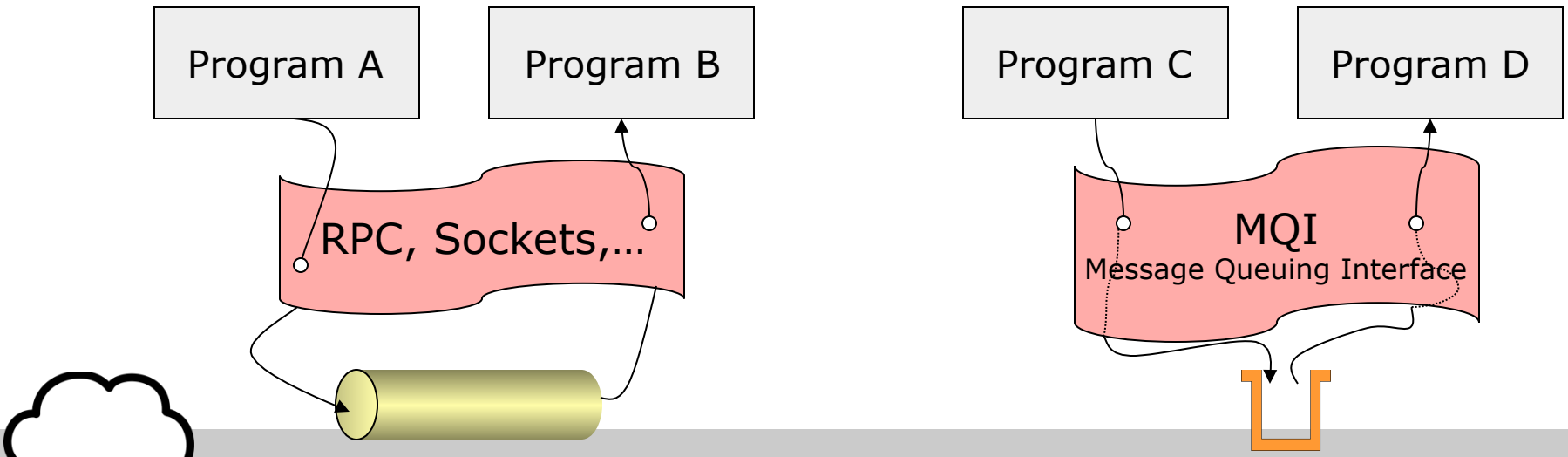
Fundamentals to Design, Build, and Manage Cloud Applications

EXTRA MATERIALS extras.springer.com

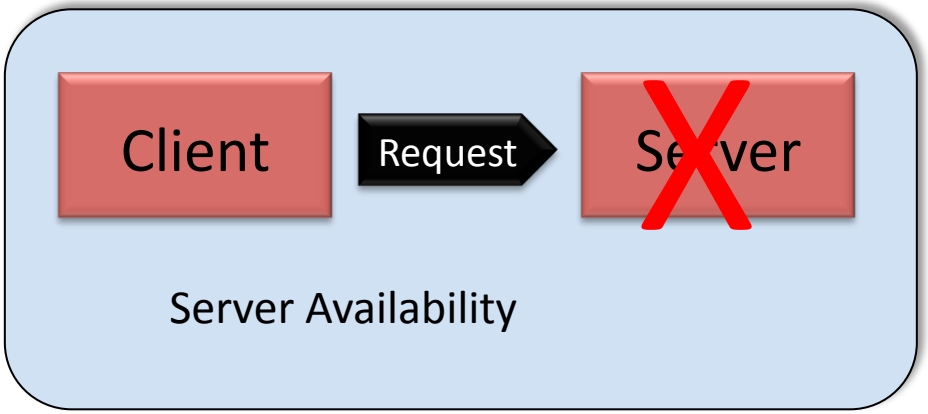
Springer

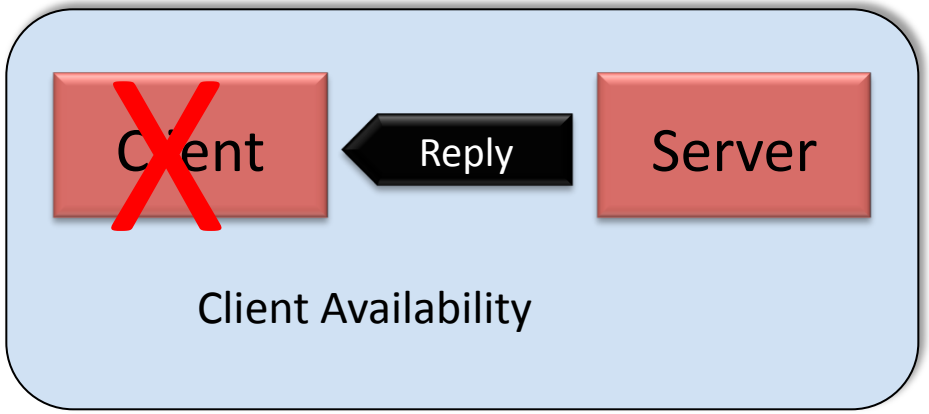
Queue-Based Programs

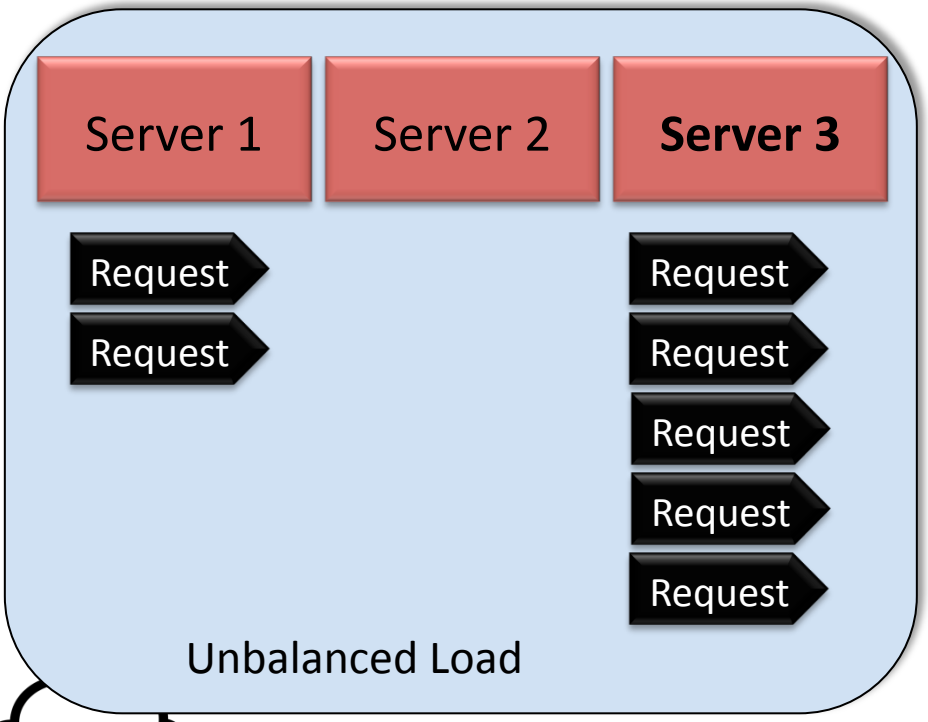
- Connection based programs
  - are operating dependently
  - based on predictable pairings
  - specify program name of partner
- Queue based programs
  - are operating independently
  - based on predictable or unpredictable pairings
  - specify queue name

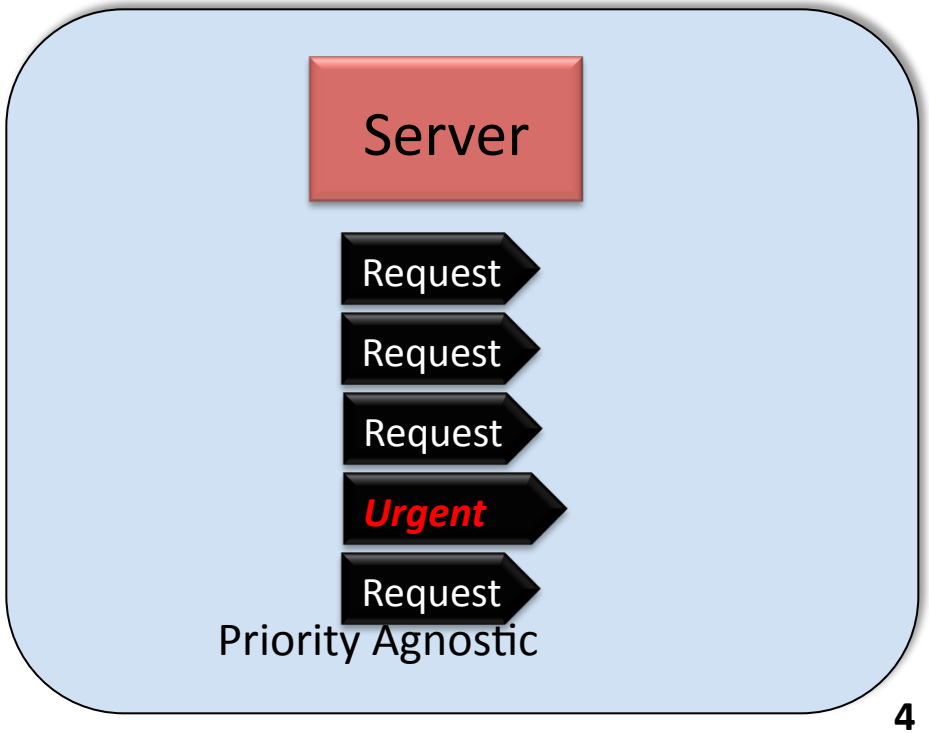


Problems in Direct TP









Loose Coupling

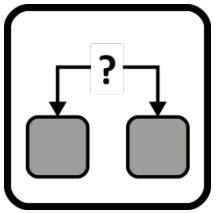
- Core principle:
  Reduce number of assumptions two parties make about each other when they exchange information

- But: Making more assumptions facilitates to increase efficiency → Tight coupling for high-performance environments
  - But less tolerance to changes at a partner‘s side

“Loose Coupling” is a very important new term in practice today!

Sometimes, it is used nearly as a synonym for
“being message-based”… ☺

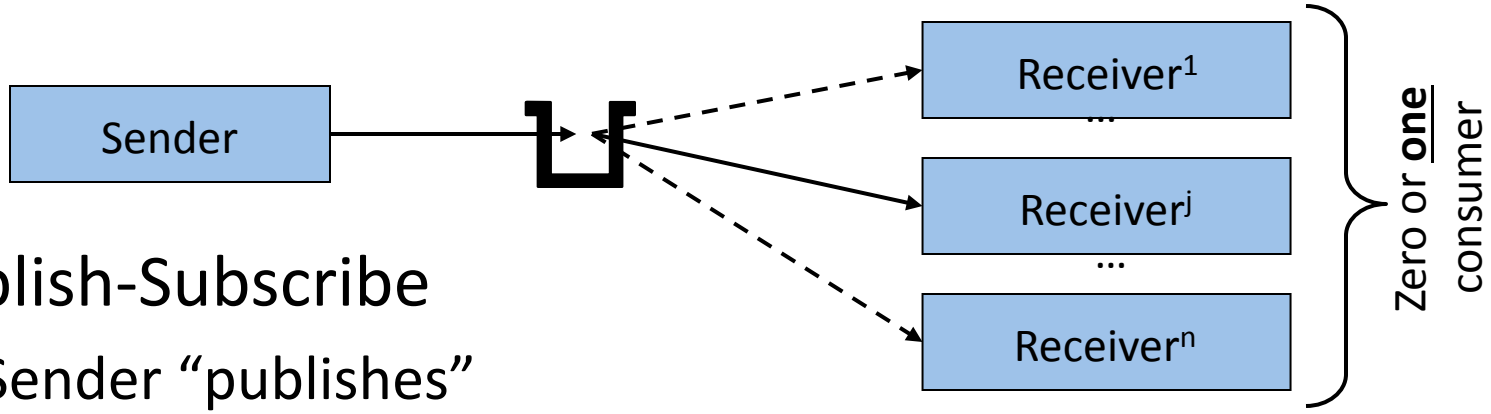
Loose Coupling: Autonomy Aspects



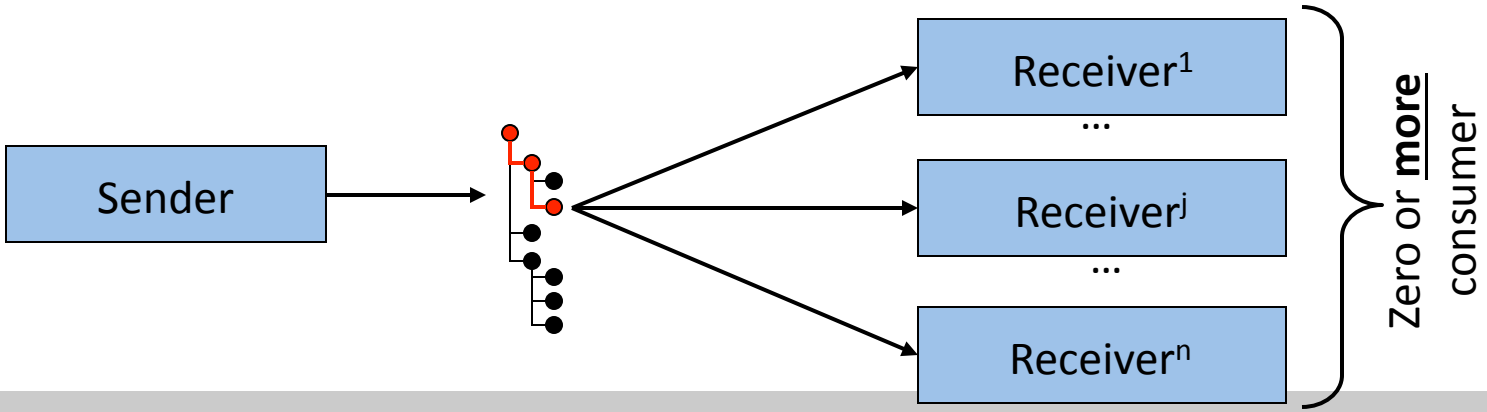
- Reference Autonomy
  - Producers and consumers don't know each other
- Platform Autonomy
  - Producers and consumers may be in different environments, written in different languages,…
- Time Autonomy
  - Producers and consumers access channel at their own pace
    - Communication is asynchronous
    - Data exchanged is persistent
- Format Autonomy
  - Producers and consumers may use different formats of data exchanged
    - Requires transformation "on the wire" (aka **mediation**)

Messaging Styles

- Point-to-Point (that's what has been discussed until now)



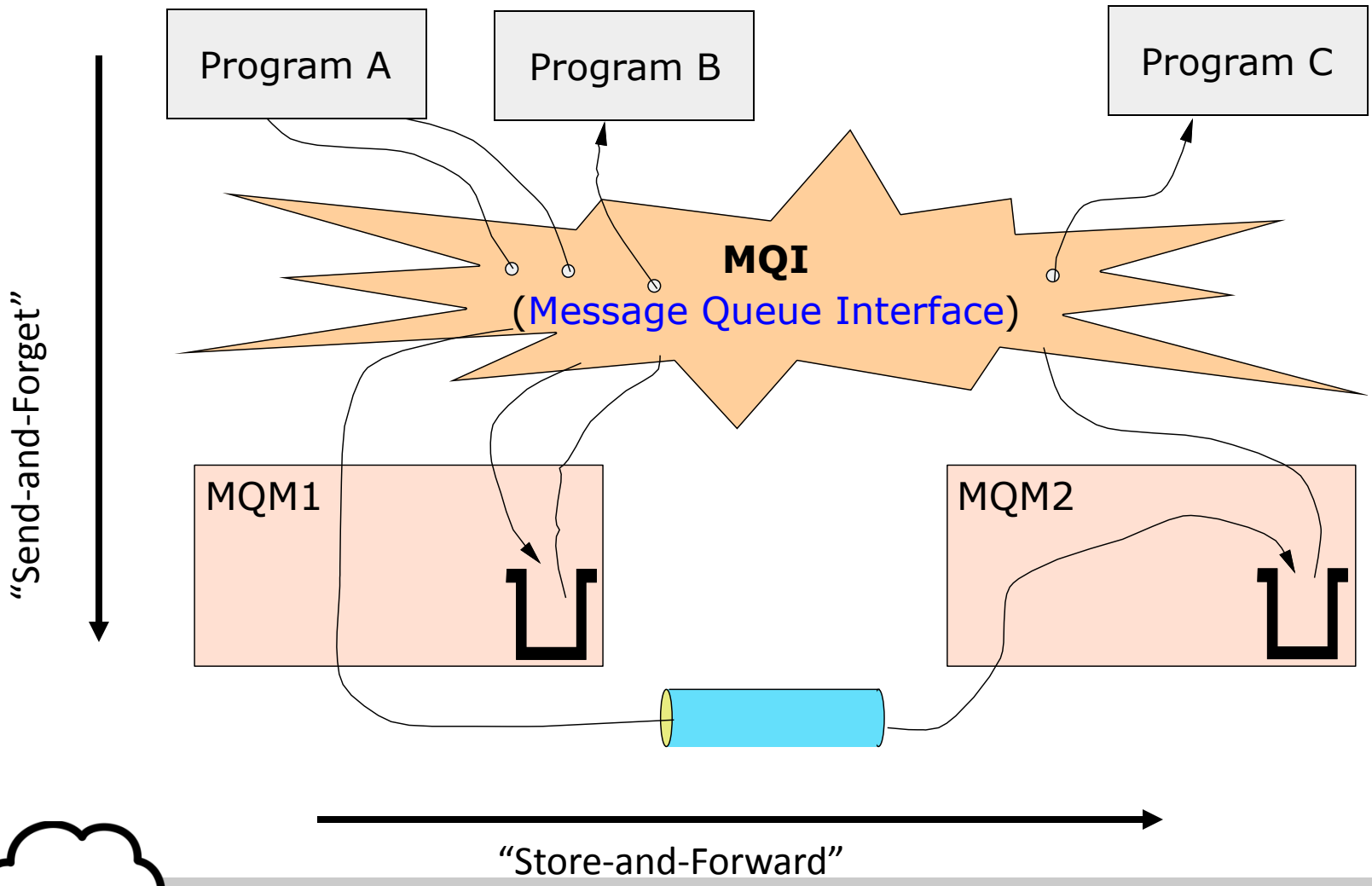
- Publish-Subscribe
  - Sender "publishes" a message on a "topic"
  - Zero or more "subscribers" on that topic get that message delivered



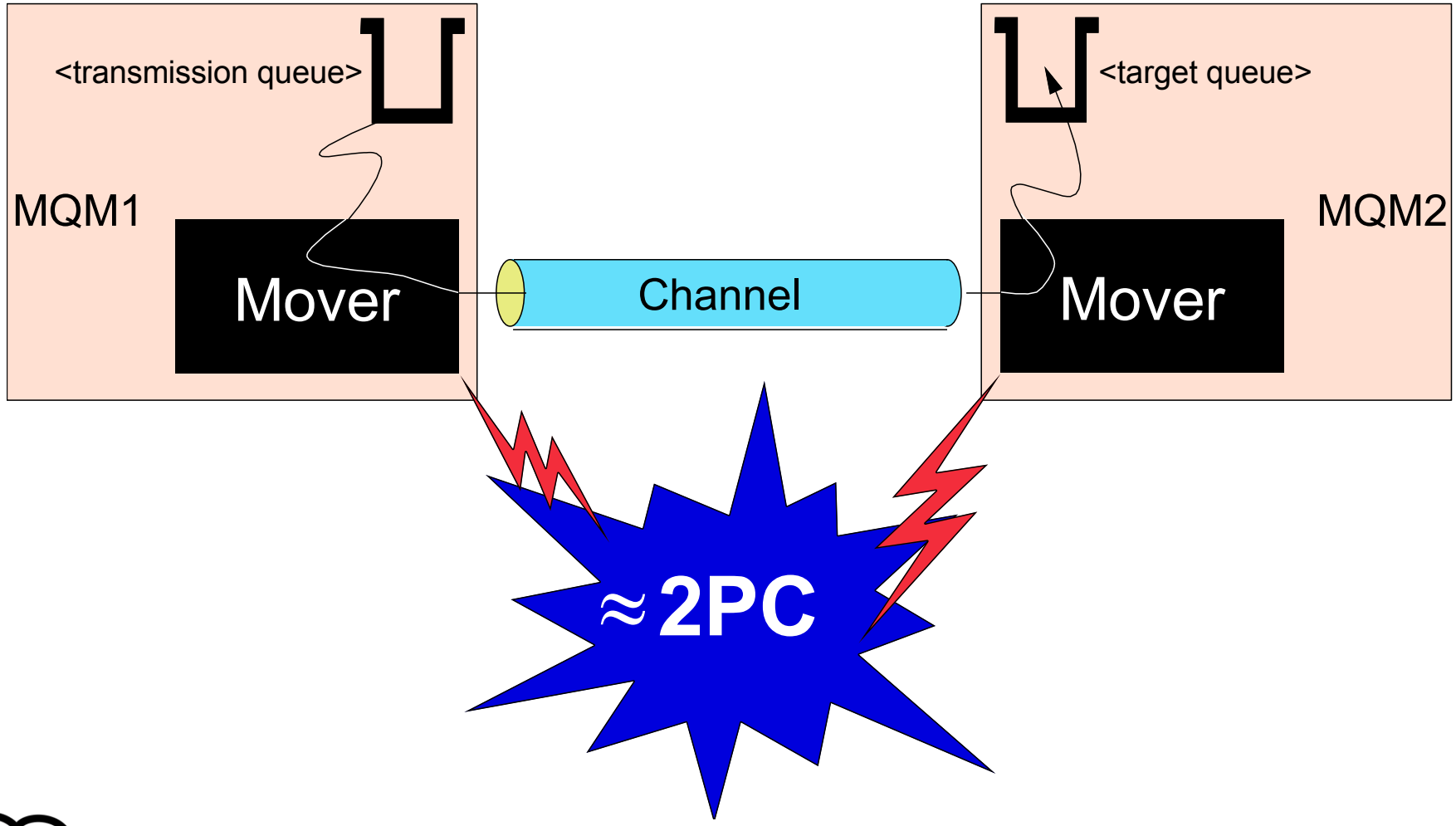
Message Queue Manager (MQM)

- **Message queue manager** (**MQM**) provides environment for queuing applications including the MQI
  - provides reliable storage for queued messages
  - manages concurrent access to data
  - ensures security and authorization
  - provides special queuing functions (like triggering)
- Applications that need queuing facilities must first connect to an MQM
- The MQM an application is directly connected to is called its *local queue manager*
- The application may run as clients using the **Message Queuing Interface** (**MQI**)

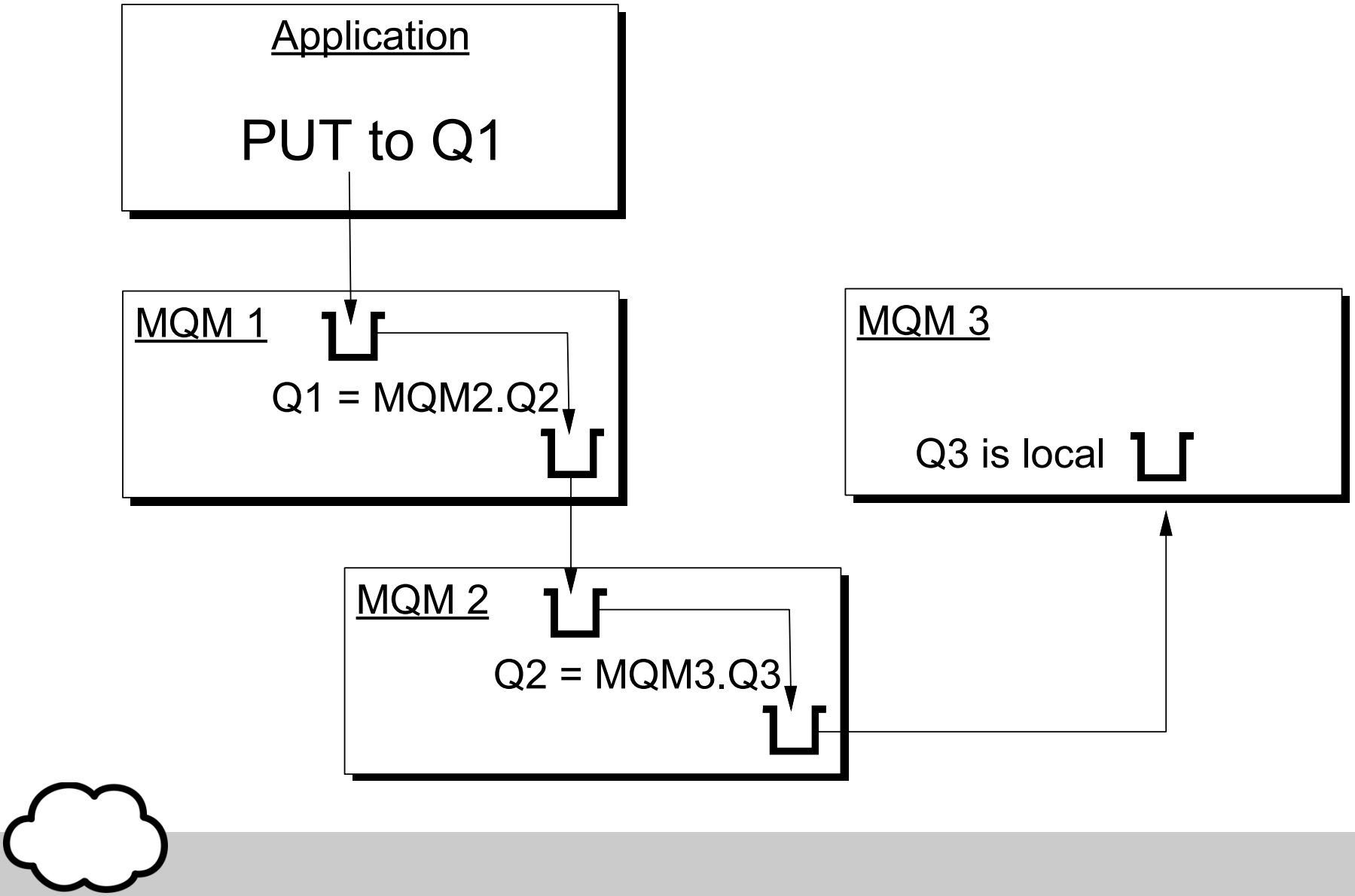
Local/Remote Queues And The MQI



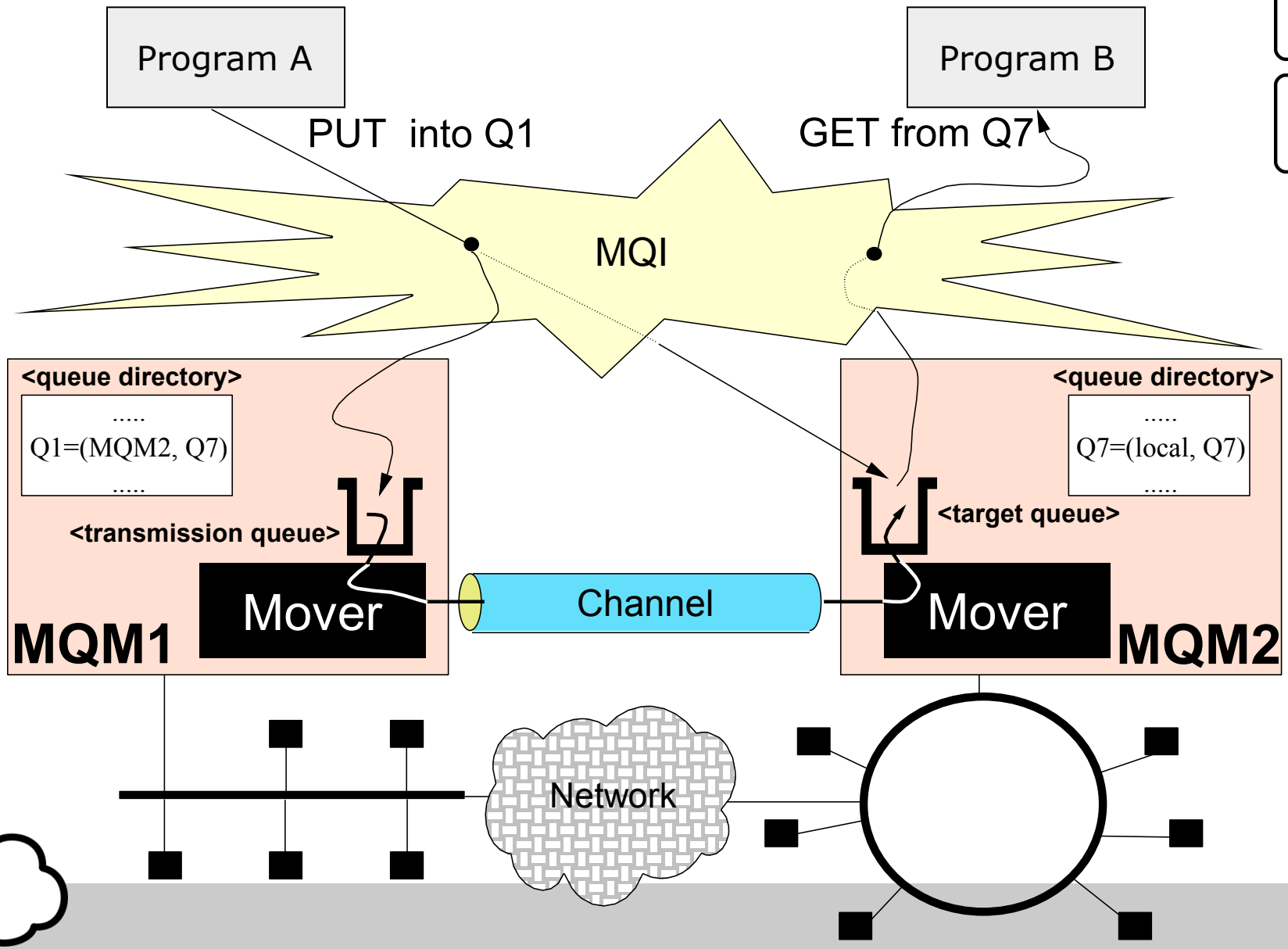
The Mover


<transmission queue>
<target queue>
MQM1
MQM2
Mover
Channel
Mover
≈2PC

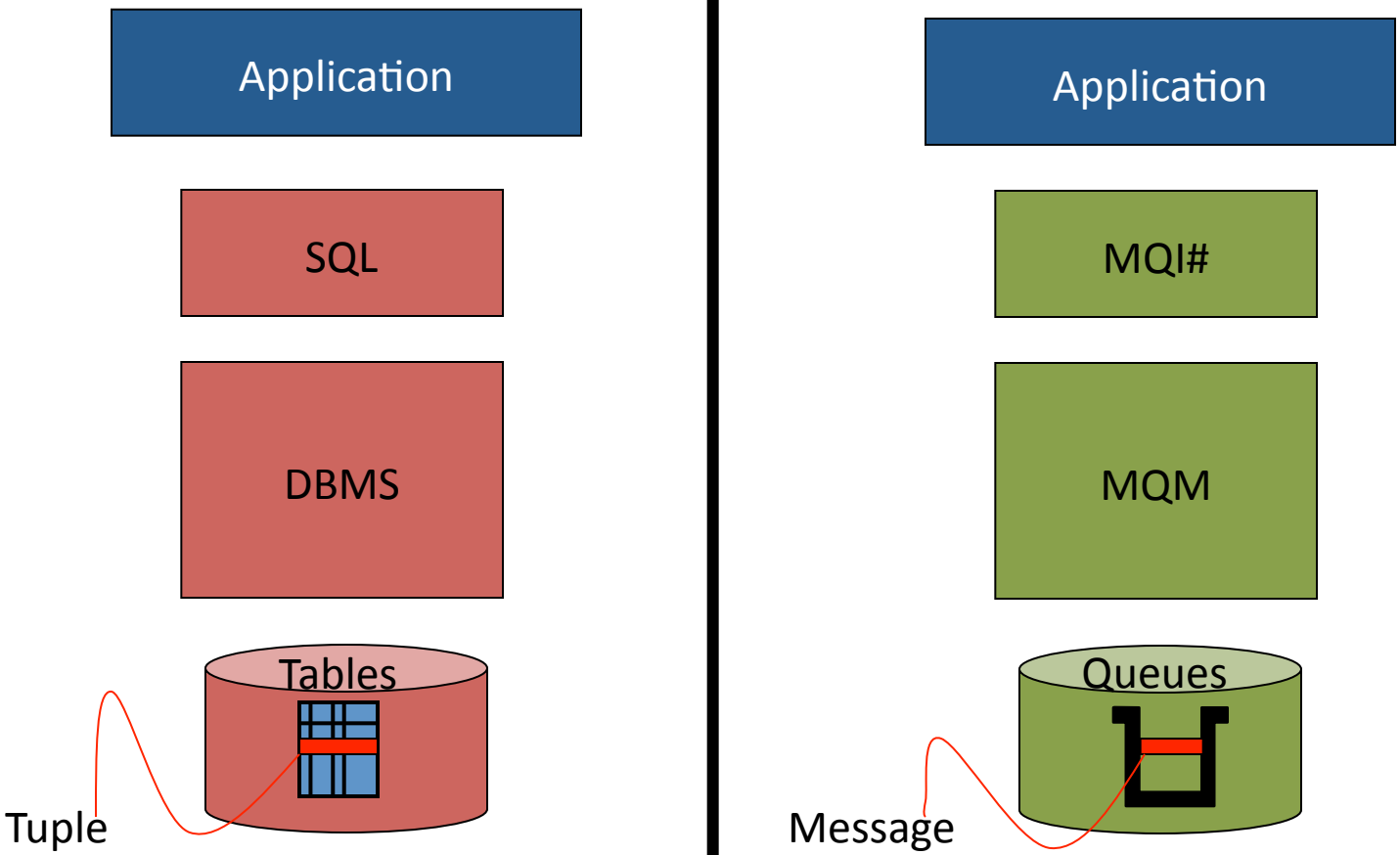
Multi-Hop Forwarding



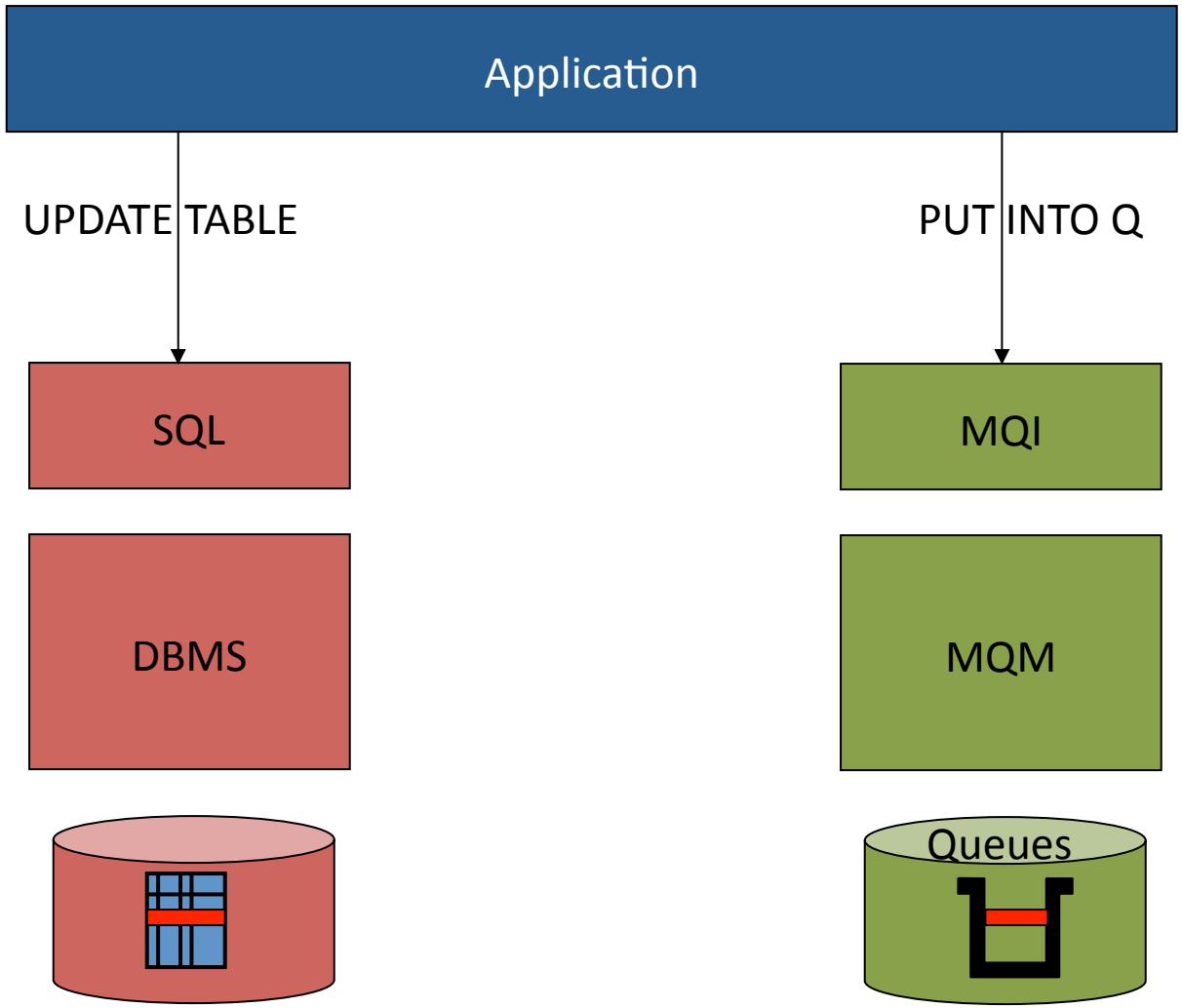
Message Delivery: Summary



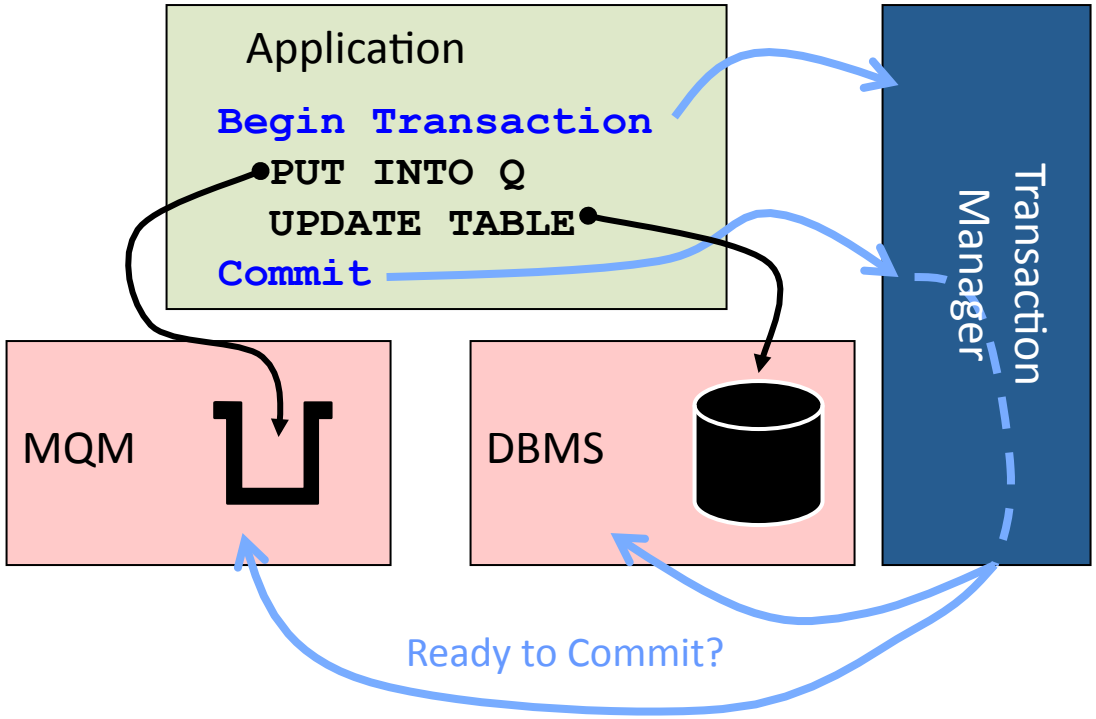
…In Case You Know Databases ☺



Transactions Across DBMS And MQM

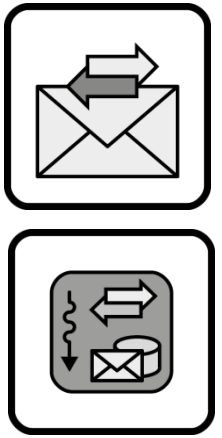


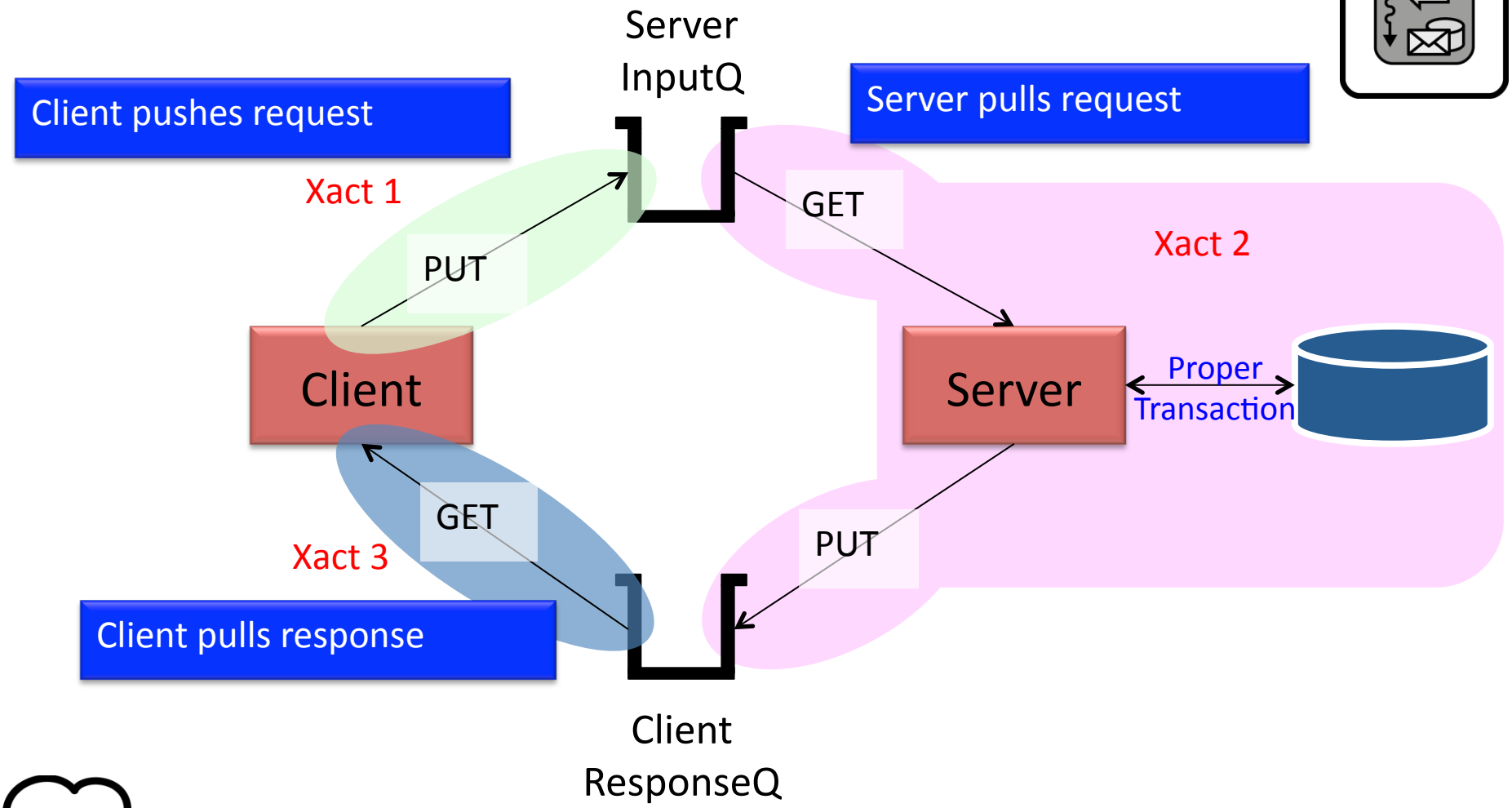
MQM As Resource Manager



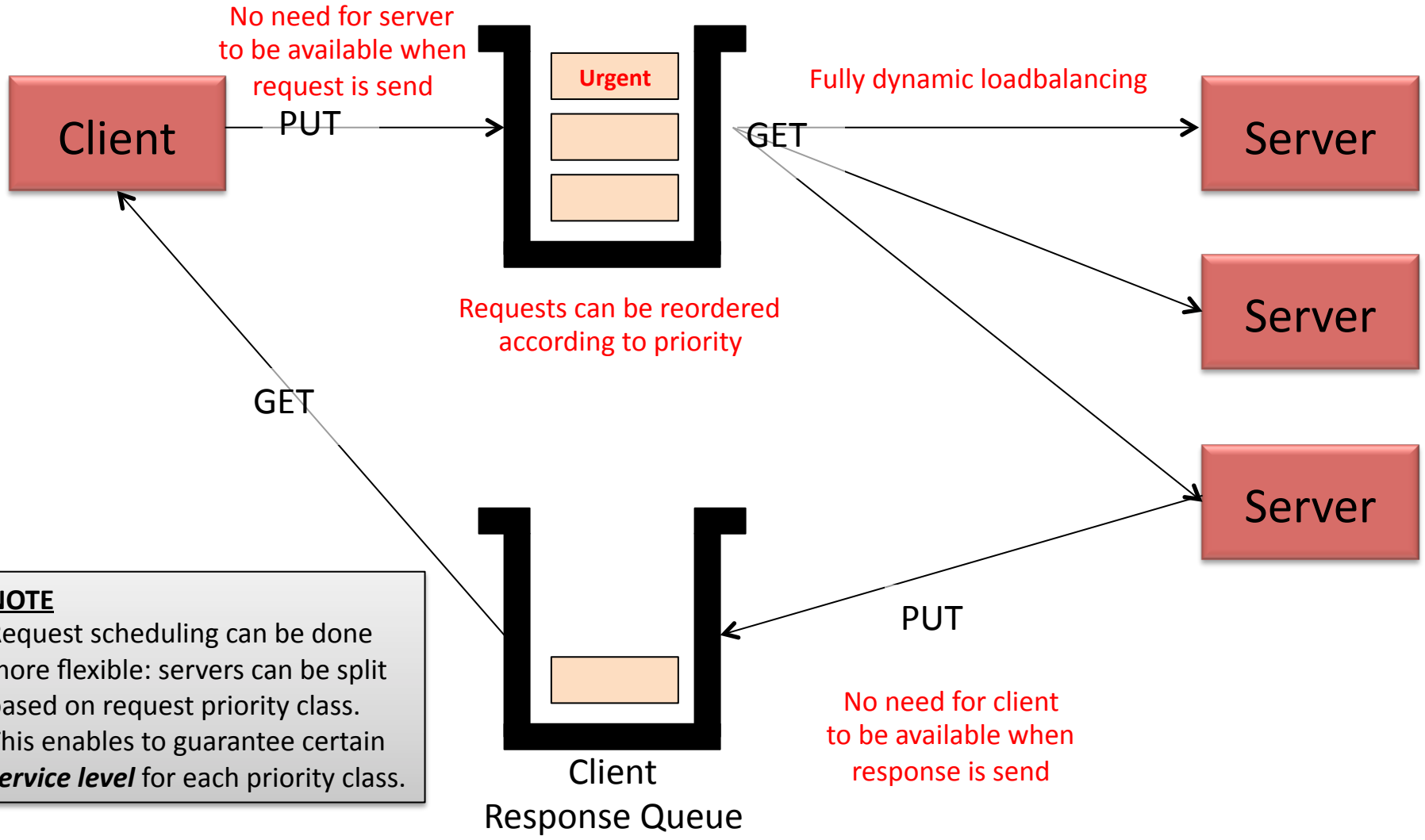
- Application can manipulate resources in multiple resource managers
- All manipulations can be grouped into a single transaction
- Transaction manager will run Two-Phase-Commit protocol to ensure that all manipulations are committed or undone
- Note the "*Poisoned Message Problem*": If a message cannot be processed and results in ROLLBACK it will be put into the queue and processing begins again, i.e. ROLLBACK + processing + ROLLBACK + … (ad infinity)
  - Multiple solutions: Synchronization points, maximum number of retries maintained my MQM

Queued TP




Server
InputQ
Client pushes request
Server pulls request
Xact 1
GET
Xact 2
PUT
Client
Server
Proper
Transaction
GET
Xact 3
PUT
Client pulls response
Client
ResponseQ

Problems Solved With Queued TP



NOTE
Request scheduling can be done more flexible: servers can be split based on request priority class. This enables to guarantee certain ***service level*** for each priority class.

Definition & Solution Principles

Performed workload is usually required to be linear proportional to resources (linear scalability):

$$\text{performed workload} = \gamma \times \text{resources},$$

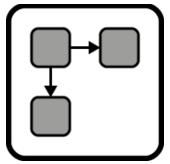
where the factor is ideally close to one: $\gamma \lesssim 1$

In principle, this can be achieved in three different ways:

- Work harder
  Processing speed
- Work smarter
  Better algorithms
- Get help
  Introduce parallelism

Scalability is the ability to
endure increasing workloads
without decreasing an agreed service level when
underlying resources are also increased

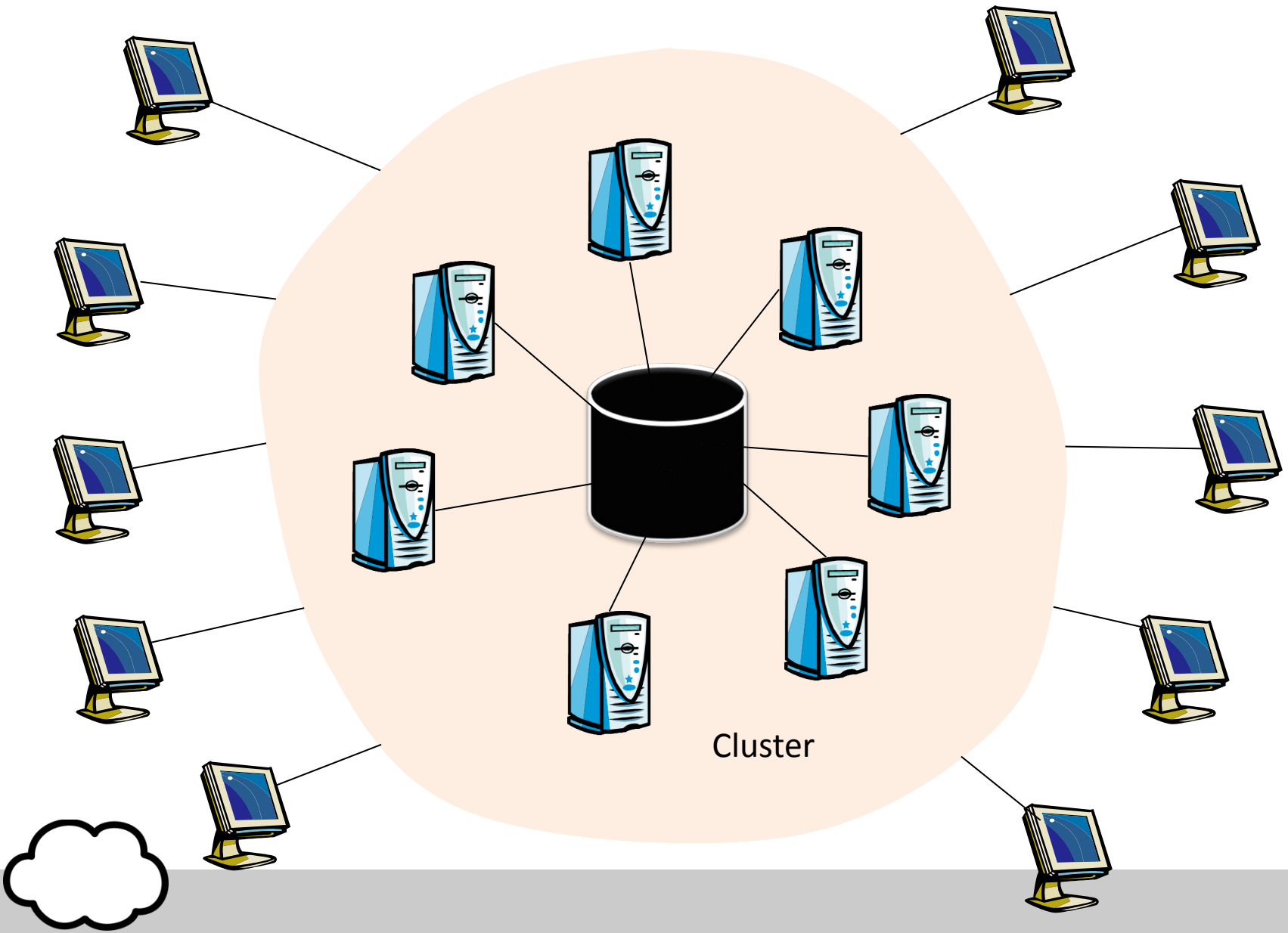
Clusters: Definition



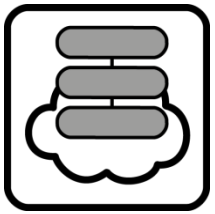
A **cluster** is a distributed system that

- consists of a collection of interconnected whole computers
- is used as a single, unified computing resource

Clusters In Client/Server Applications (1/2)



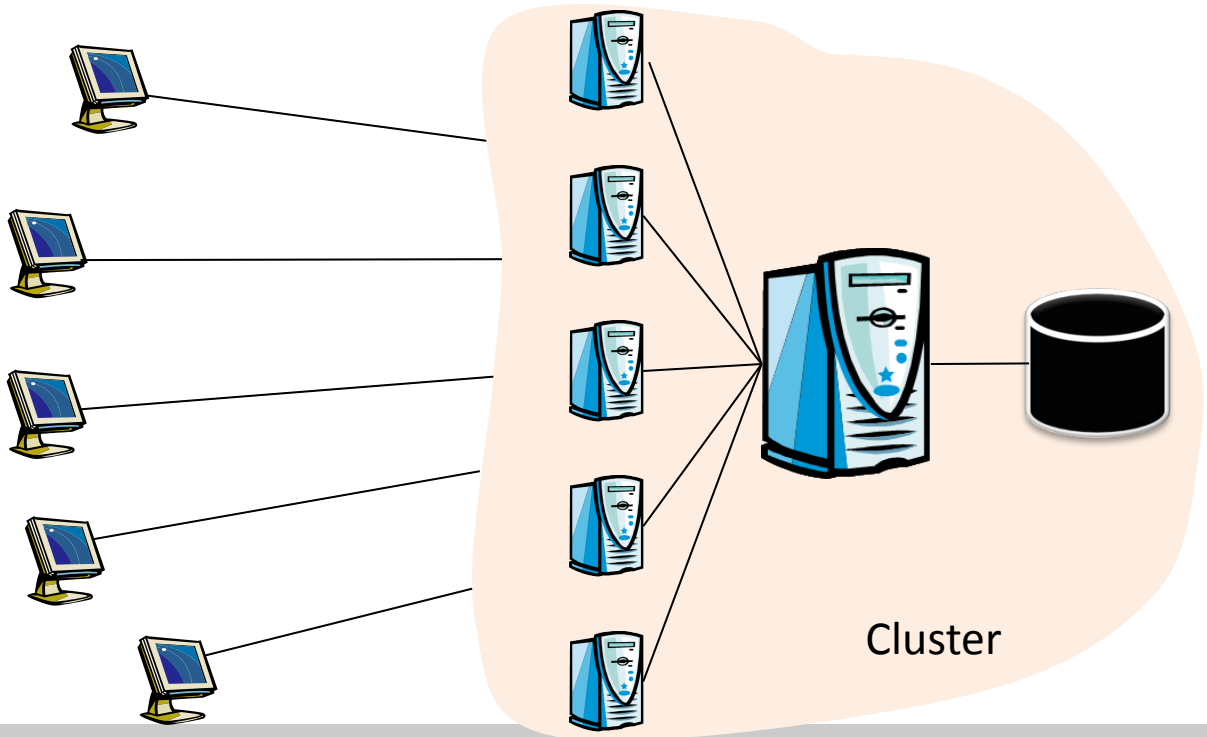
Clusters In Client/Server Applications (2/2)



Client request service from cluster

- Cluster selects node for performing requested function
  - Selection can be done by client-stub of cluster
- Node accesses shared data
  - Physically shared disc, I/O shipping, shared DB,...

➔ 3-Tier-Architecture: Convenient implementation!



Definitions

- System called available if it is up and running ***and*** produces correct results
- The availability of a system is the fraction of time it is available
- A system is high available if its availability is close to 1
- Thus, high availability is the property of a system to be up and running all the time, always producing correct results
- I.e. the famous requirement is...

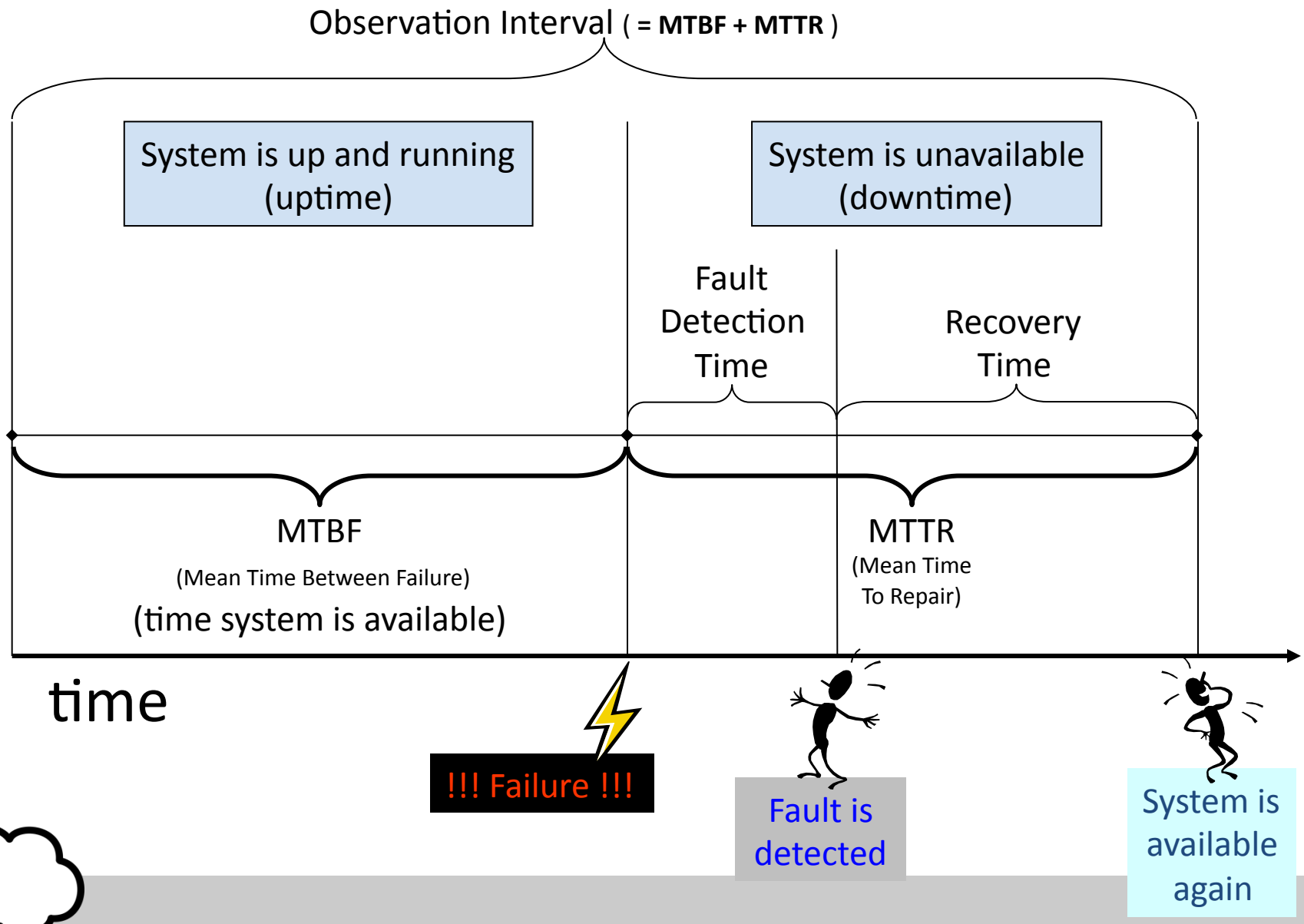
$$7 \times 24$$

- Mission critical systems (DBMS, TPM, WFMS,...) must be high available
- A system fails if it gives a wrong answer or no answer at all

- Thus:

  - The more a system fails, the less it is available
  - The longer it takes to repair a system after it fails, the less it is available

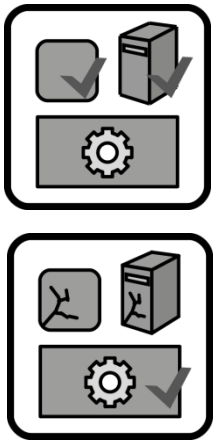
Relevant Points in Time



Formalization

- Mean Time Between Failures (**MTBF** or φ)
  - ... the average time a system runs before it fails
- MTBF measures system reliability
- Mean Time To Repair (**MTTR** or ϱ)
  - ... the average time it takes to repair a system after failure
- MTTR measures system downtime
- Availability α: $$\alpha = \frac{\varphi}{\varphi + \varrho}$$
- Thus, availability can be improved by
  - improving reliability, i.e. MTBF $\varphi \to \infty : \lim_{\varphi \to \infty} \frac{\varphi}{\varphi + \varrho} = 1$
  - decreasing downtime, i.e. MTTR $\varrho \to 0 : \lim_{\varrho \to 0} \frac{\varphi}{\varphi + \varrho} = 1$

Availability Classes

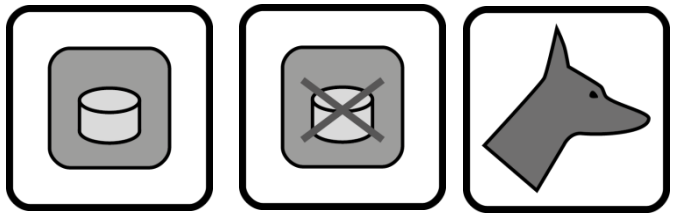


Downtime	Availability [%]	Class	System Type
1 hour / day	95.8 (~ 96)	1	Unmanaged
	Customers minimum expectation today		
1 hour / week	99.41 (~ 99)	2	Managed
1 hour / month (few minutes / day)	99.86 (~ 99.9)	3	Well managed
1 hour / year	99.9886 (~ 99.99)	4	Fault tolerant
1 hour / 20 years (5 minutes / year)	99.99942 (~ 99.999)	5	High available
30 seconds / year (3 seconds / year)	99.9999048 (~ 99.9999)	6	✌

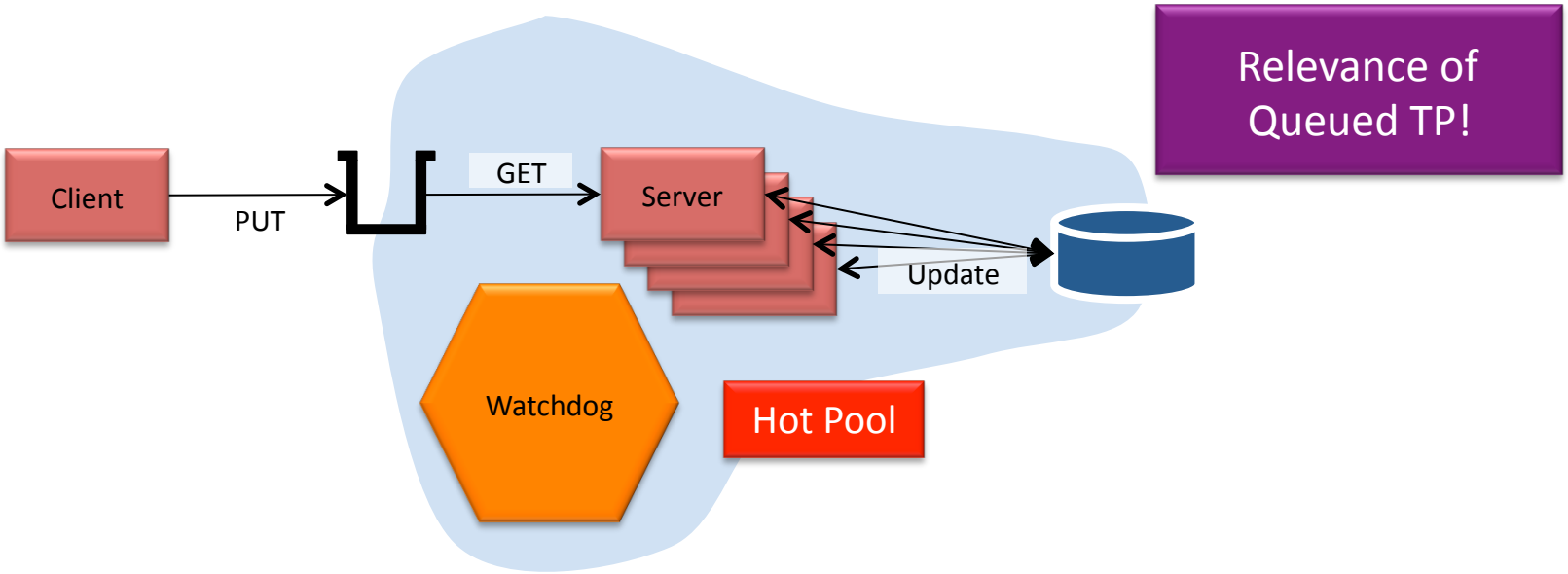
The number of 9's is characterizing the system type

- Class 1: This really bad!
- Class 2: Commodity uni-processor system
- Class 3: Standard open-system cluster
- Class 4: Cluster with special HW/SW - Achieving class 4 is already a challenge!
- Class 5: IBM S/390 Parallel Sysplex HW
- Class 6: In-flight aircraft computer
- ☺ Some vendors already call class 3...4 "High Available" ☺

Application Server Hot-Pooling

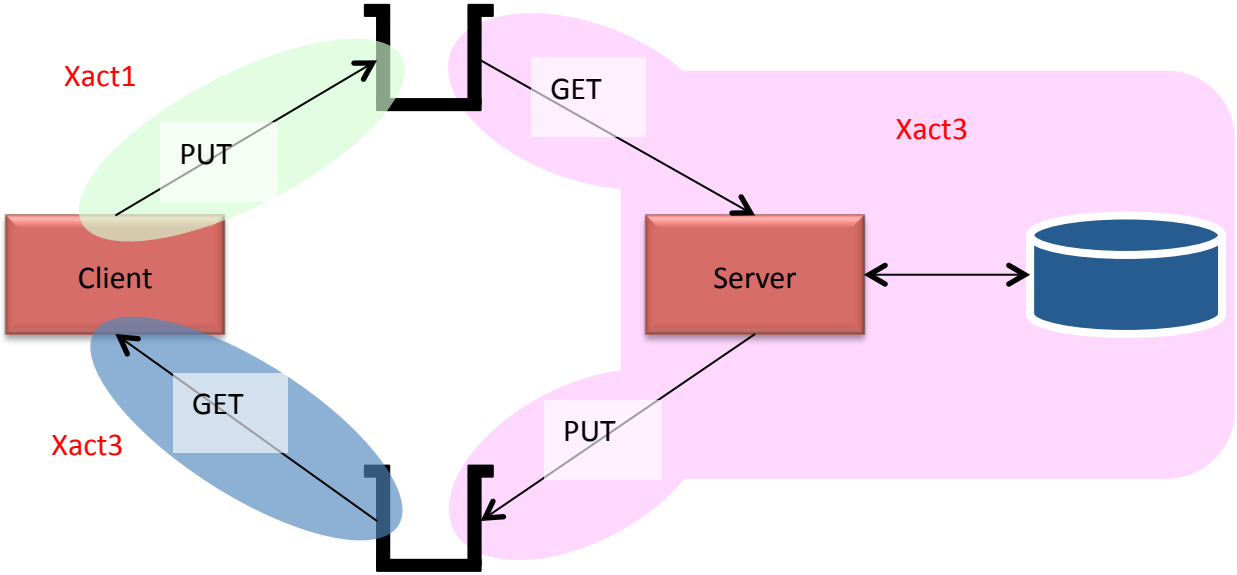


- **Hot Pool**:
  - A collection of application servers with identical functionality sharing a common pool input queue
  - Client sends requests to pool input queue
  - Whenever a member of the hot pool finished processing particular request it immediately gets next request from pool input queue
- When a member fails the other members will continue processing requests from pool input queue: Hot pool appears as "virtual application server"
- Watchdog will initiate failed member's restart



Ensuring Availability In Hot Pools: Transactions

- Assumption:
  - pool input queue is persistent
  - each hot pool member ensures message integrity
- Consequence: As long as single member is running “virtual application server” is available
- MTTR is further reduced!



Probability Considerations For Hot Pools

- **Binomial Distribution:**
  Let E be an event that occurs with probability P; perform the same experiment N times independently (e.g. in parallel) and count how often a particular event E occurs. Then, the probability that E occurs exactly k times is

$$P_k = \binom{N}{k} P^k \left(1-P\right)^{N-k}$$

- The availability of hot pool members is binomial distributed:
  Each member in a hot pool runs a copy of the same software, so MTTR and MTBF, thus the availability is the same for each member. The N independent experiments consists of running N hot pool members in parallel, and the event observed is “member available”. Thus, the probability that exactly k members are available of a hot pool having N members is

$$P_k = \binom{N}{k} \alpha^k \left(1-\alpha\right)^{N-k}$$

- The probability of at least k members being available is

$$P_{\geq k} = \sum_{i=k}^{N} P_i = \sum_{i=k}^{N} \binom{N}{i} \alpha^i \left(1-\alpha\right)^{N-i}$$

Availability Of A Hot Pool

- The services provided by its hot pool members is available as long as at least one member of the hot pool is available:

$$\alpha_{hot\ \text{pool}} = P_{\geq 1} = \sum_{i=1}^{N} \binom{N}{i} \alpha^i (1-\alpha)^{N-i}$$
$$= \sum_{i=0}^{N} \binom{N}{i} \alpha^i (1-\alpha)^{N-i} - \left(\binom{N}{0} \alpha^0 (1-\alpha)^N \right)$$
$$= \left(\alpha + (1-\alpha)\right)^N - (1-\alpha)^N$$

- Thus, the availability of $\alpha_{hot\text{pool}}$ of a hot pool with N members each member having the same availability α_{member} is:

$$\alpha_{hot\ \text{pool}} = 1 - \left(1 - \alpha_{member}\right)^N$$

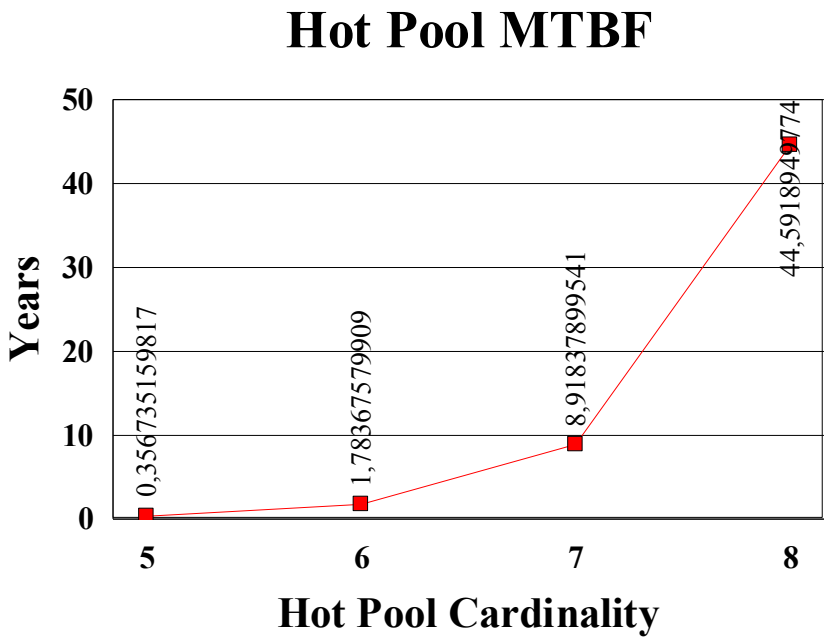
- If an event has a probability much less than one and if it is memoryless, i.e. its occurrence is not influenced by the occurrence of the same event before, then **the mean time to such an event is the reciprocal of the probability of the event**

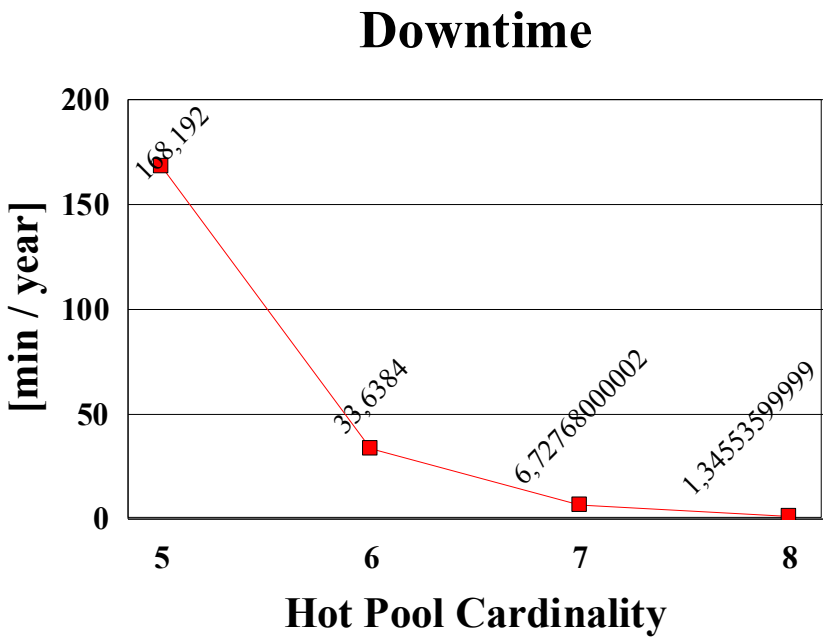
- The failing of a hot pool is such an event having the probability $1-\alpha_{hot\ \text{pool}}$, thus:

$$MTBF_{hot\ \text{pool}} = \left(1 / \left(1 - \alpha_{hot\ pool}\right)\right)$$

Hot Pool Availability (Sample Numbers)

- Assumptions: Hot pool member MTBF=24h, MTTR=6h, resulting member availability is 80% (i.e. unacceptable)
- **But**: Hot pool gets availability class 5 already with 8 members

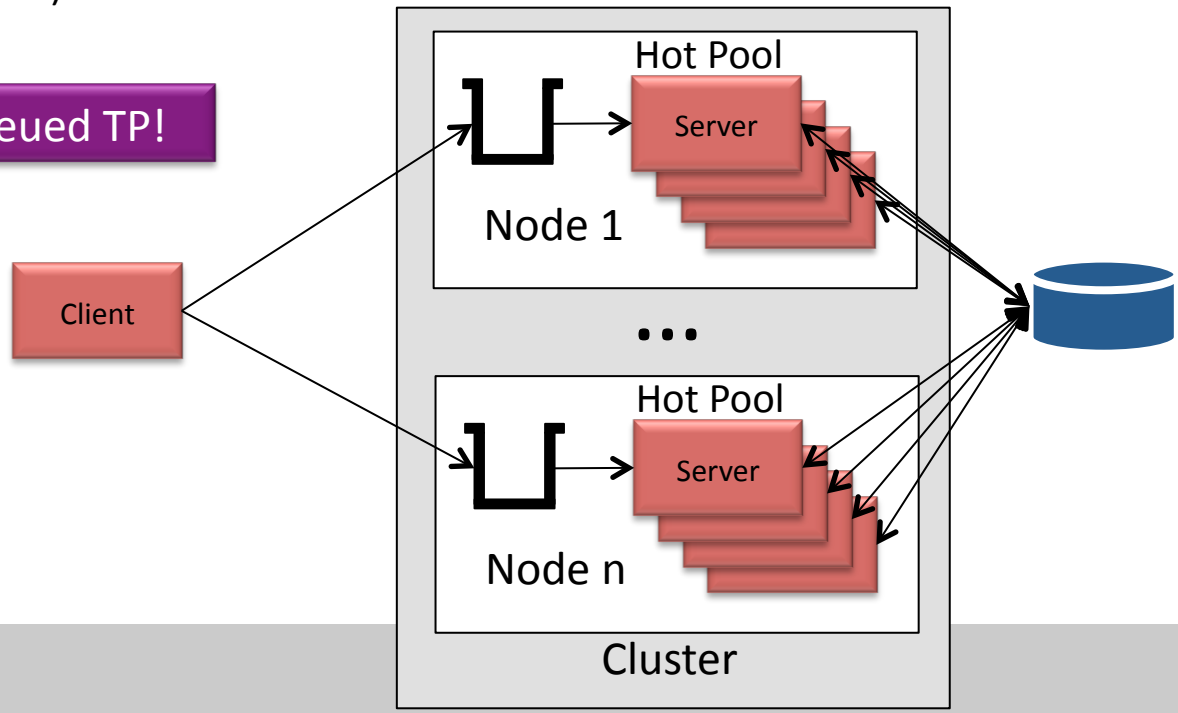




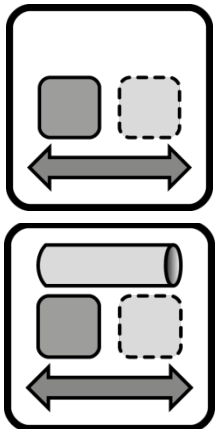
Application Clusters

- An **Application Cluster** is defined to be
  - a collection of interchangeable nodes
  - each node hosts a hot pool of one and the same (application) server
  - the nodes hosting the cluster fail independently
- Especially:
  - All systems have access to all of the data (thus the DBMS is critical w.r.t. availability of the cluster! Thus, the DBMS may run in a separate cluster!)
  - The cluster provides a single system image, i.e. clients submit requests to the cluster not to a particular system (especially, all systems run the same kind of hot pool)

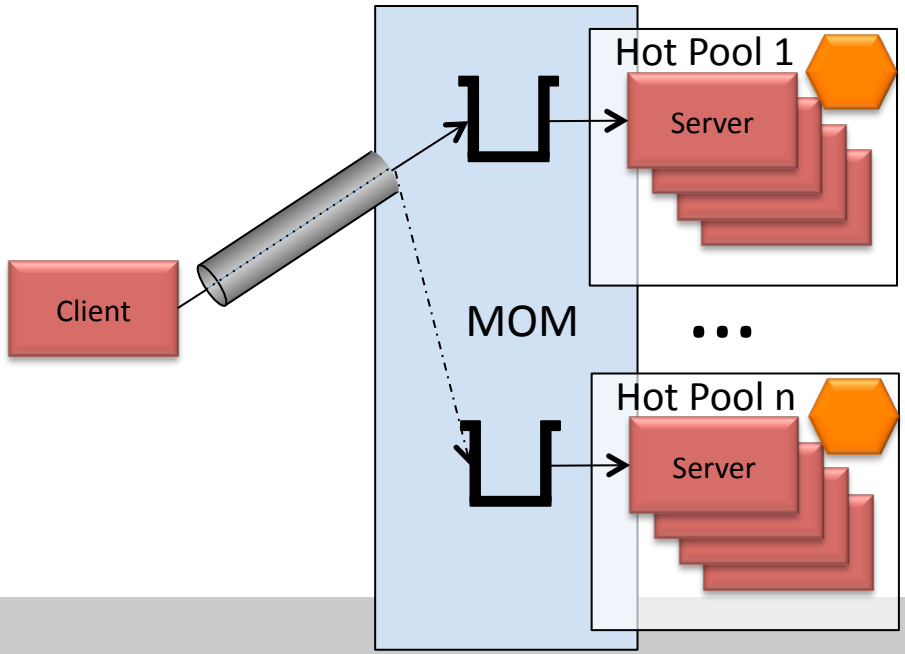
Relevance of Queued TP!



Take-Over Across Hot Pools



- If complete hot pool fails (e.g. OS or processor fails) automatic reroute of requests to available hot pool on different system: MTTR is further reduced
- Either…
  - via cluster architecture (hardware, system software,...)
  - or underlying messaging system supports reroute directly (aka virtual queue or cluster queue)
  - or client exploits specific enhancements of underlying MOM (e.g. server APIs)
- Thus, client behaves as if it has a session with a hot-pool-plex
- Mechanism to restart the failed hot pool depends on particular fault



Cluster Availability Versus Cost

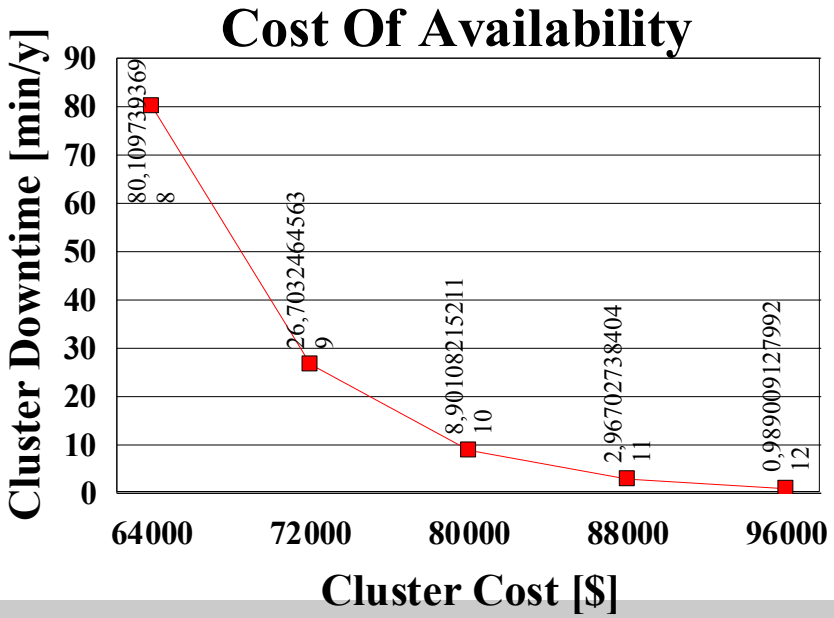
- Let α_{system} denote the availability of a system (i.e. a hosting environment), M the number of systems in the cluster

- The services provided by a cluster is available as long as at least one system of the cluster is available. As before:

$$\alpha_{cluster} = 1 - \left(1 - \alpha_{system}\right)^{M}$$

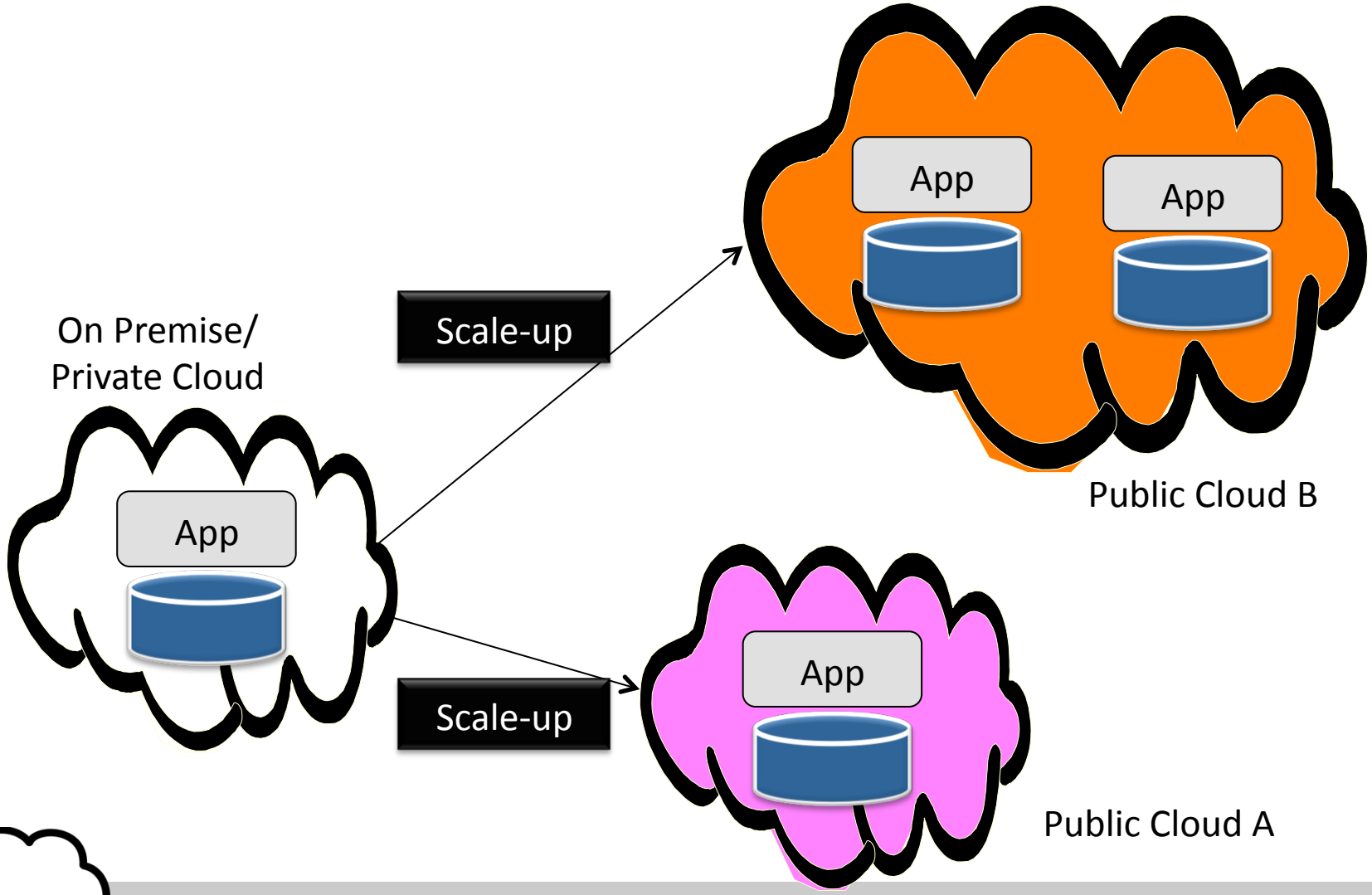
- Basically, the cost $C_{cluster}$ of a cluster is the sum of the cost C_{system} of its systems : $C_{cluster} = MC_{system}$

- Assuming a unit price C_{system} of \$8,000 per system and an availability α_{system} of 66% the following results:



Scaling Applications Up in the Cloud

- Scale-up results in (massive) distributed systems



The Problem

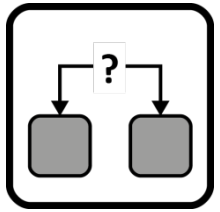
- Massive distributed systems operate on a large (even world-wide) scale

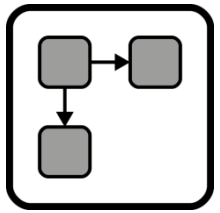
- Large scale creates additional challenges:

  When a system processes billions (even trillions) of requests, events that normally have a low probability of occurrence are now guaranteed to happen

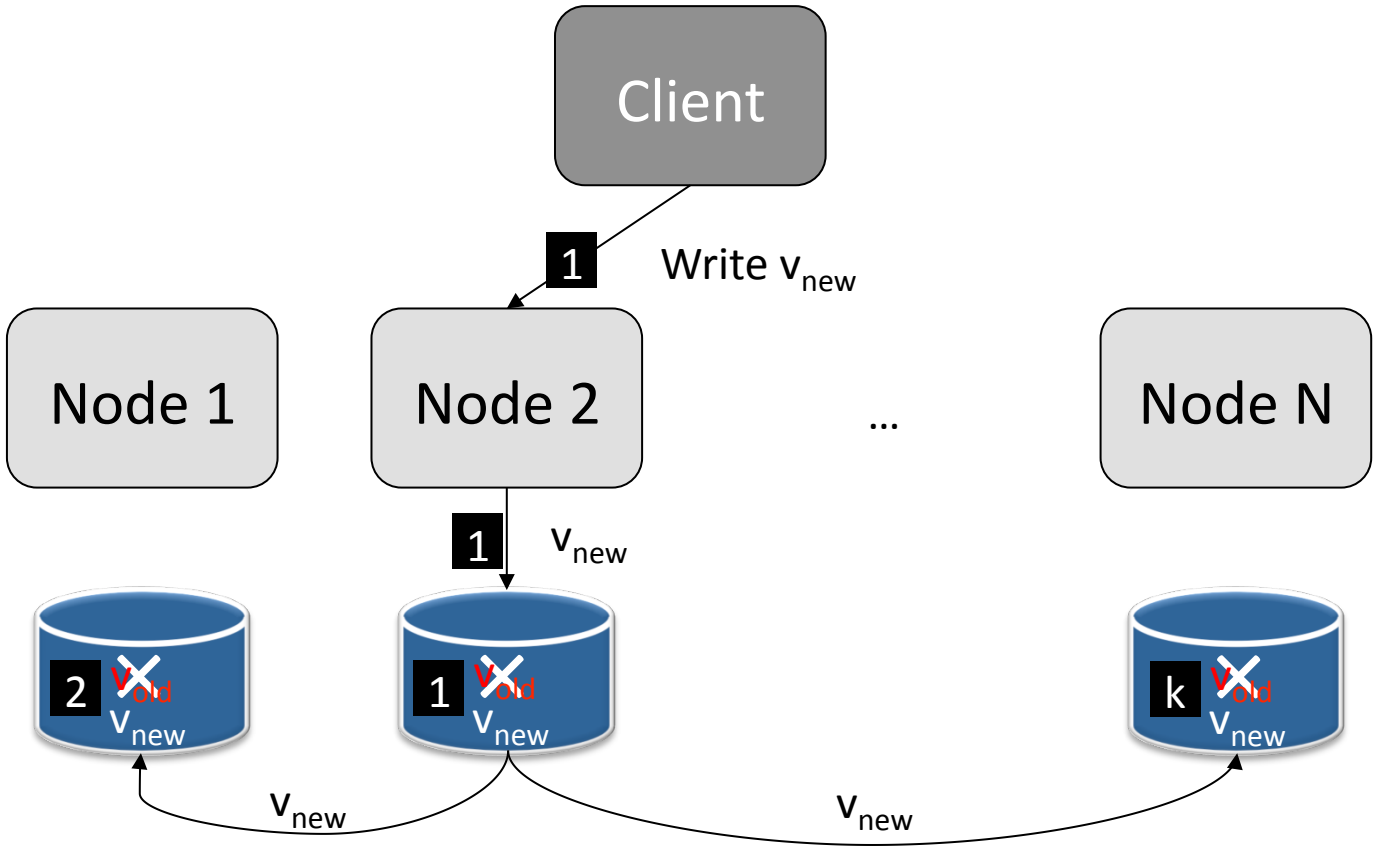
- These events need to be accounted for up front in the design and architecture of the system

Updates in Massive Distributed Systems



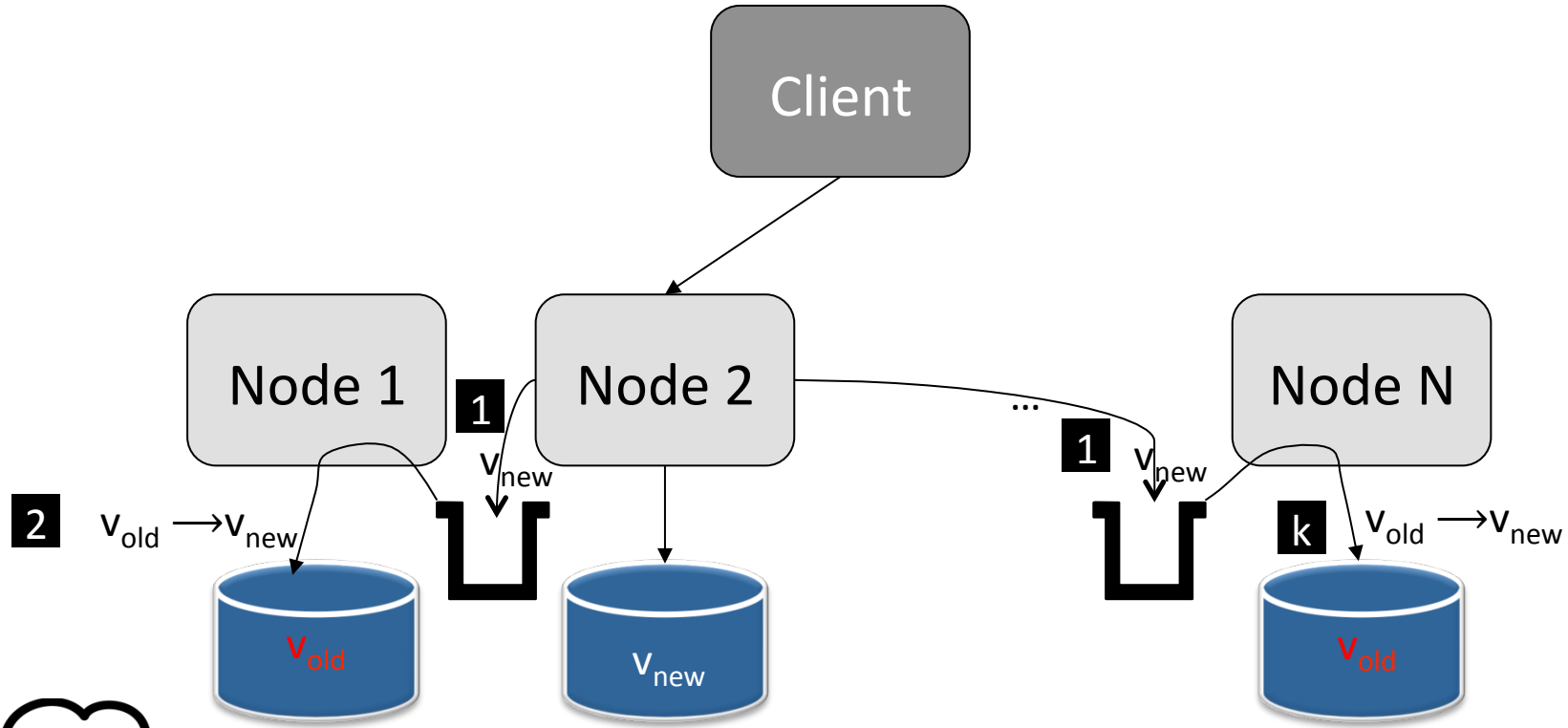


- Data are often replicated across the nodes
  - Clients updates single node of the system
  - System replicates updates to other nodes in a lazy manner



The Role of Queues in Replication

- Replication is typically based on (persistent) queues
  - Updated node sends new value to other nodes via queues
  - Other nodes update their local copy (lazily)



In an Ideal World...

- ...all replicas would be updated at the same time – resulting in data consistency
  - Each node holding a copy of a data item has the same value of the data item
    - i.e. the client can access any of the nodes to retrieve the actual latest value of the data item

C

...any responding node would offer all functions of the overall system – resulting in availability

- Failures have no impact of the available functions of the overall systems
  - i.e. the client can select any of the nodes to request all functions

A

...failure of network connections would not affect the correctness of the overall system – resulting in partition tolerance

- A partitioning of the network has no impact on the functionality of the overall system
  - i.e. the client can access any of the nodes for any function and will always get same and correct results

P

But Life is Cruel

The CAP Theorem (Brewer, 2000)

Out of the following three properties of a system:

Consistency

Availability

Partition tolerance

…only two can be achieved at the same time

- Since in larger distributed systems network partitions are a given, such systems must be build with partition tolerance in mind!
- As a consequence, in such systems consistency and availability cannot be achieved at the same time!

Architectural Relevance

There are only two choices:

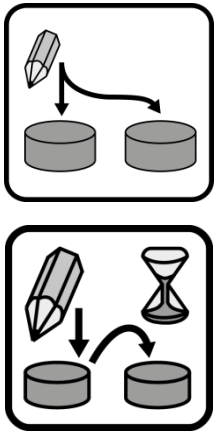
- Demand consistency

  This means that under certain conditions the system will not be available

- Relax consistency

  This will allow the system to remain highly available

Consistency Models (Vogels, 2008)



- Strong consistency
  - After update completes, any subsequent access by anybody will return the updated value
- Weak consistency
  - System does not guarantee that subsequent accesses will return the updated value
  - Period between update and the moment when it is guaranteed that any observer will read the updated value is called **inconsistency window**
- **Eventual consistency** (specific form of weak consistency)
  - Storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value

Tolerating Partitions

- Assume a partitioning is detected
    - Each partition is treated as self-contained replica set
        - i.e. reads and writes are done to partition only
- *Merge* is performed once partitioning is healed

⇛ This how Amazon's shopping cart application works

The BASE Properties

- Transactional properties for large-scale applications
  - …proposed as ACID alternative…
- **BASE** means
  - **B**asically **A**vailable

    := support partial failures without total failure
    - E.g. if a node fails, only the users on that node are affected by the failure
  - **S**oft state

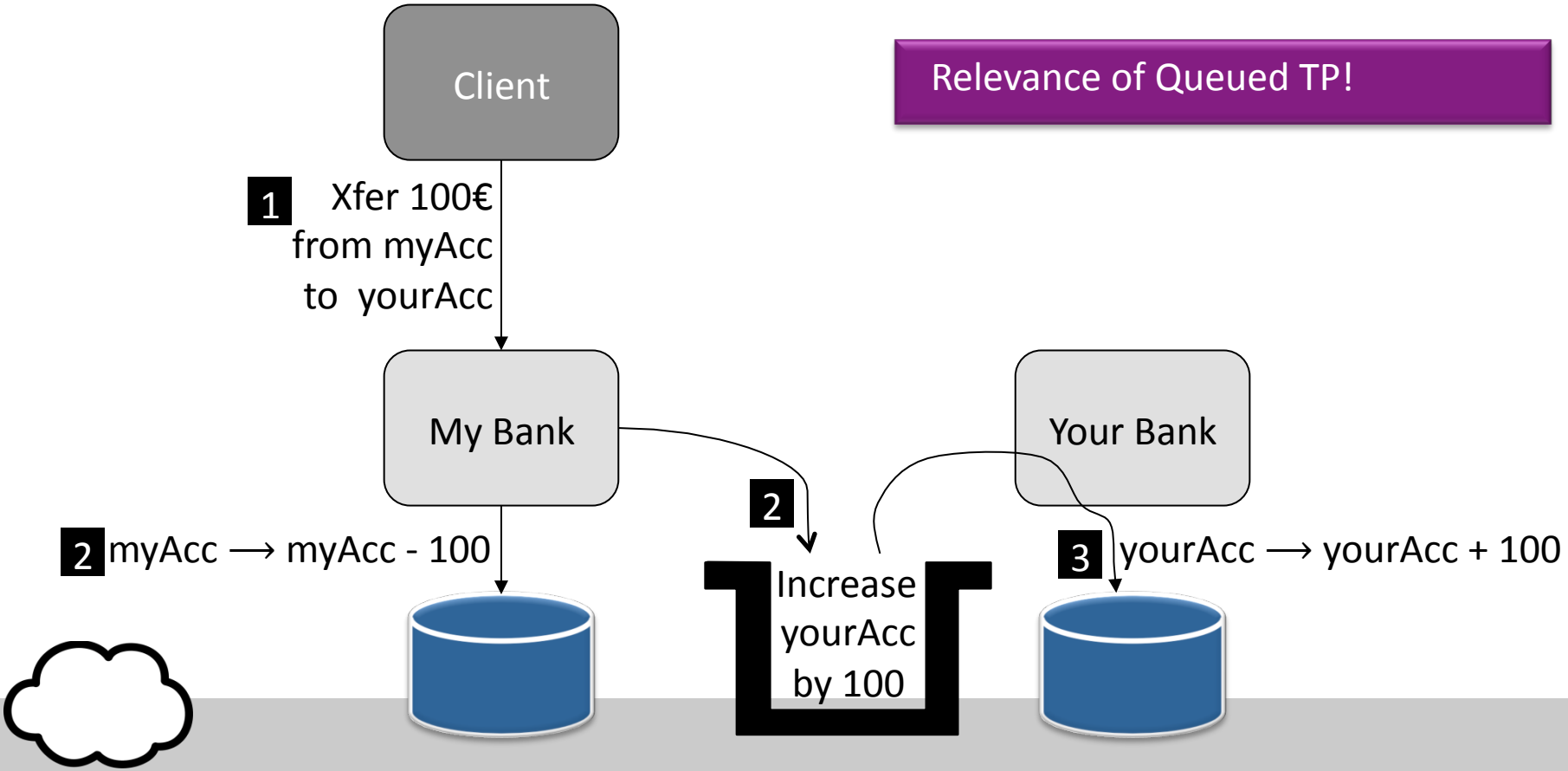
    := state changes triggered by an operation may be scattered across different components of the system
    - E.g. the debit of a funds transfer is in a database, but the credit of the funds transfer is still in a message within some input queue
  - **E**ventually consistent

    := at some point in time the effects of an operation is reflected in all affected components, and in a consistent manner
    - E.g. finally, the debit and the credit required for a successful and consistent funds transfer are reflected on both databases

Example

- Real world funds transfer follows the BASE paradigm

- Debit and Credit can succeed independently (Basically Available)
- State is scattered across Debit DB and Request queue at [2] (Soft state)
- Funds transfer finally consistently reflected in both DBs [3] (Eventually consistent)



A Note to Caution

- Consequently, large-scale distribute applications (e.g. applications distributed across clouds) require message queuing etc.
- Such middleware is available as cloud offerings
  - Amazon SQS, Microsoft Azure,…
- **But:** These queuing services behave differently than used from on-premise middleware (MQSeries, MSMQ,…)
  - No exactly-once delivery, but at-least-once delivery
  - No 2PC transactions between MOM and DBMS
  - …
- Thus, building large-scale applications in the cloud requires to consider these differences
  - Build idempotent operations
  - Detected duplicate messages
  - …

End of First Part of the Tutorial