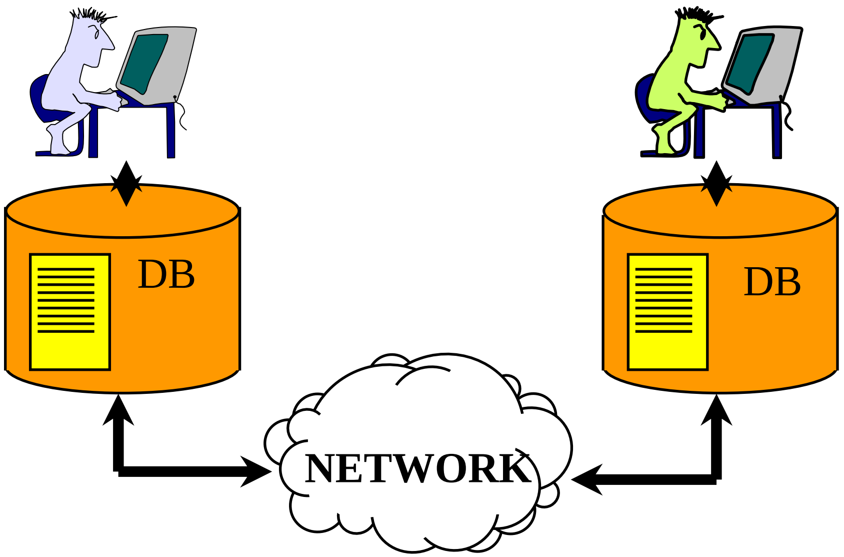
Introduction to Database Replication

- □ What is database replication
- □ The advantages of database replication
- □ A taxonomy of replication strategies:
  - Synchronous
  - Asynchronous
  - Update everywhere
  - Primary copy
- □ Discussion on the various replication strategies.

Database Replication

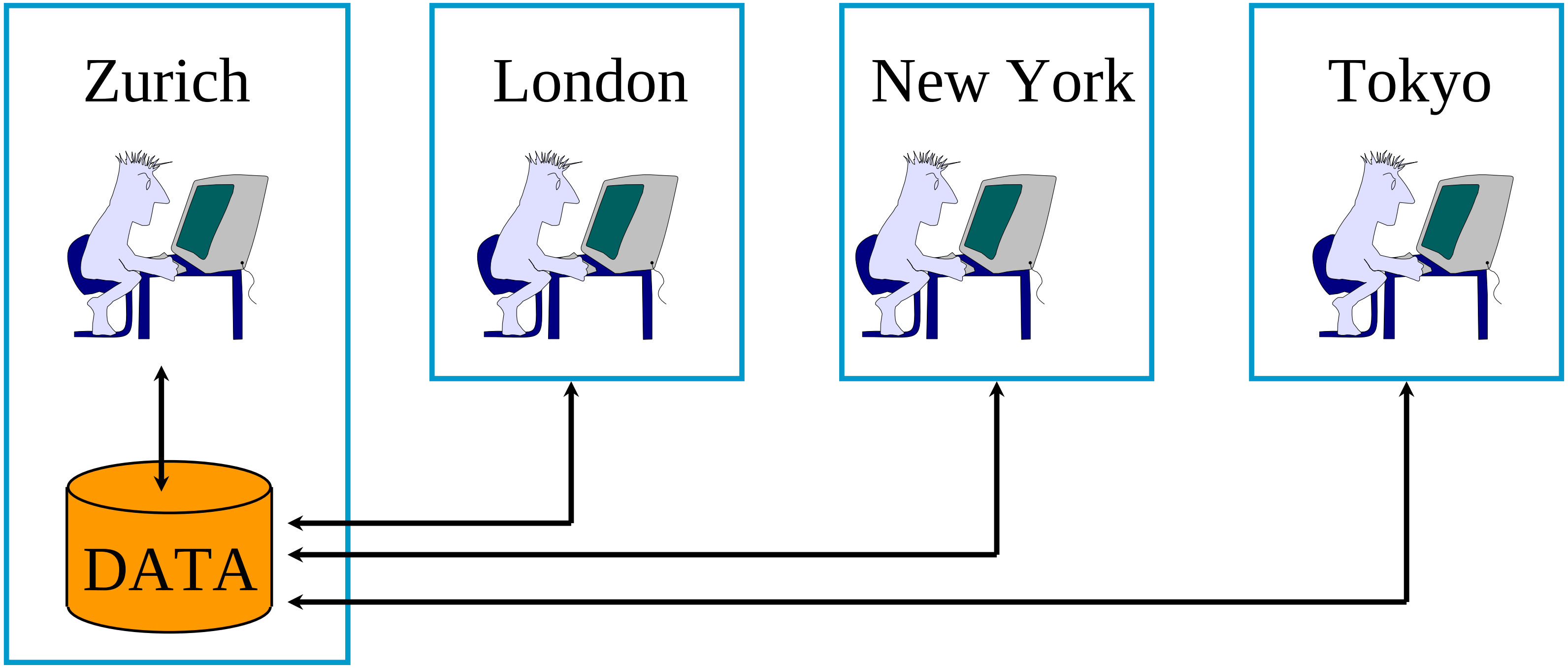


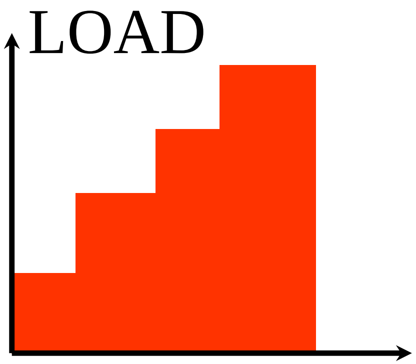
- □ Replication is a common strategy in data management: RAID technology (Redundant Array of Independent Disks), Mirror sites for web pages, Back up mechanisms (1-safe, 2-safe, hot/cold stand by)
- □ Here we will focus our attention on replicated databases but many of the ideas we will discuss apply to other environments as well.

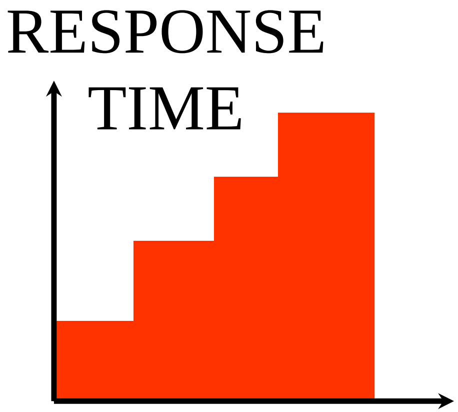
Why replication?

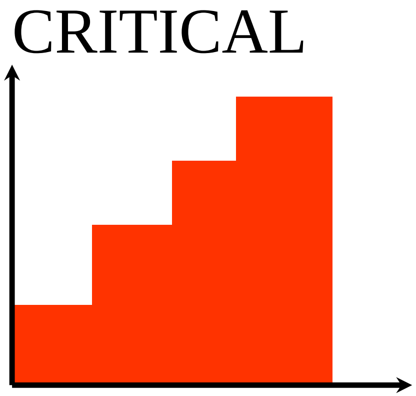
- □ PERFORMANCE: Location transparency is difficult to achieve in a distributed environment. Local accesses are fast, remote accesses are slow. If everything is local, then all accesses should be fast.
- □ FAULT TOLERANCE: Failure resilience is also difficult to achieve. If a site fails, the data it contains becomes unavailable. By keeping several copies of the data at different sites, single site failures should not affect the overall availability.
- □ APPLICATION TYPE: Databases have always tried to separate queries form updates to avoid interference. This leads to two different application types OLTP and OLAP, depending on whether they are update or read intensive.

Remote access to data?

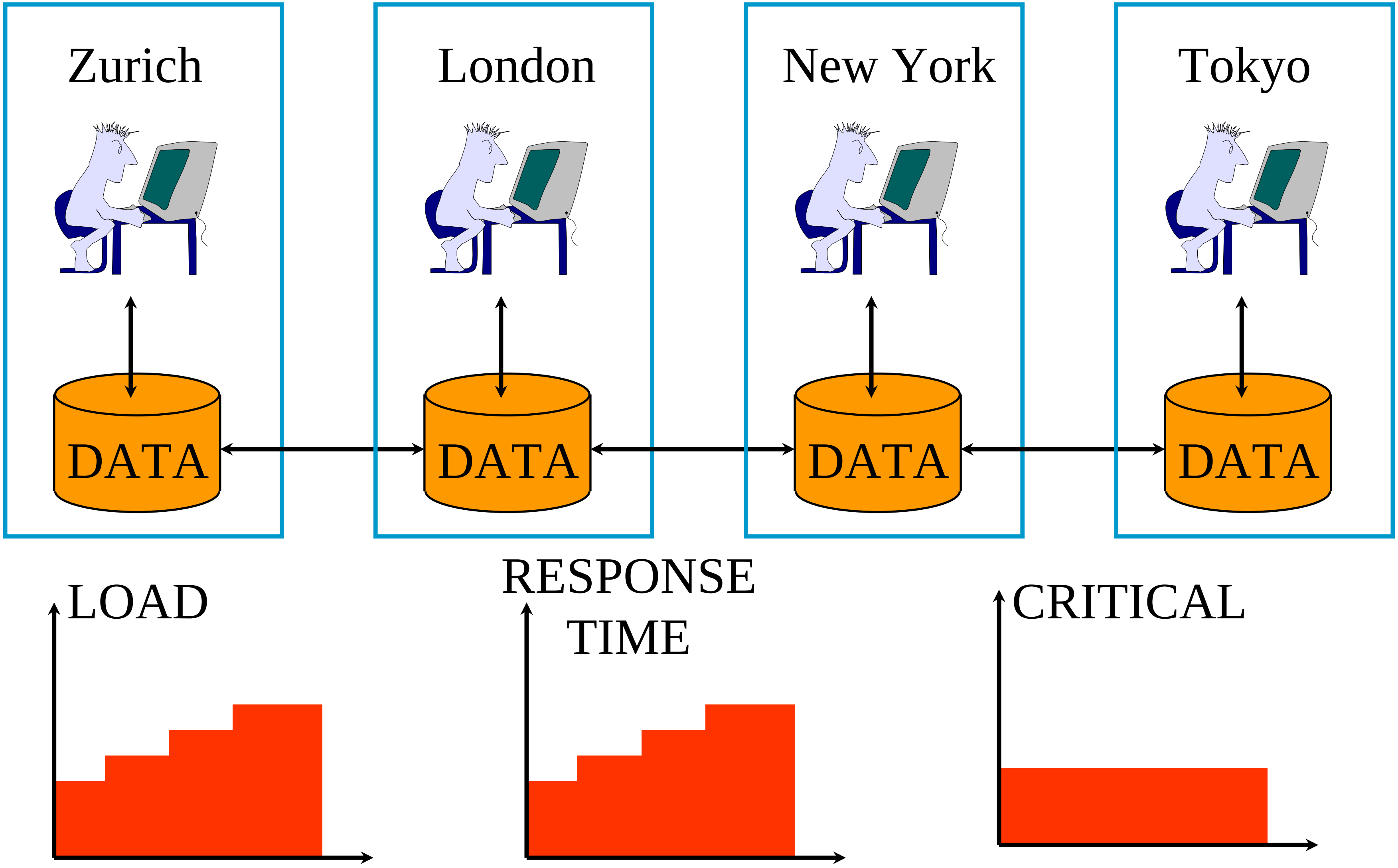








Replication



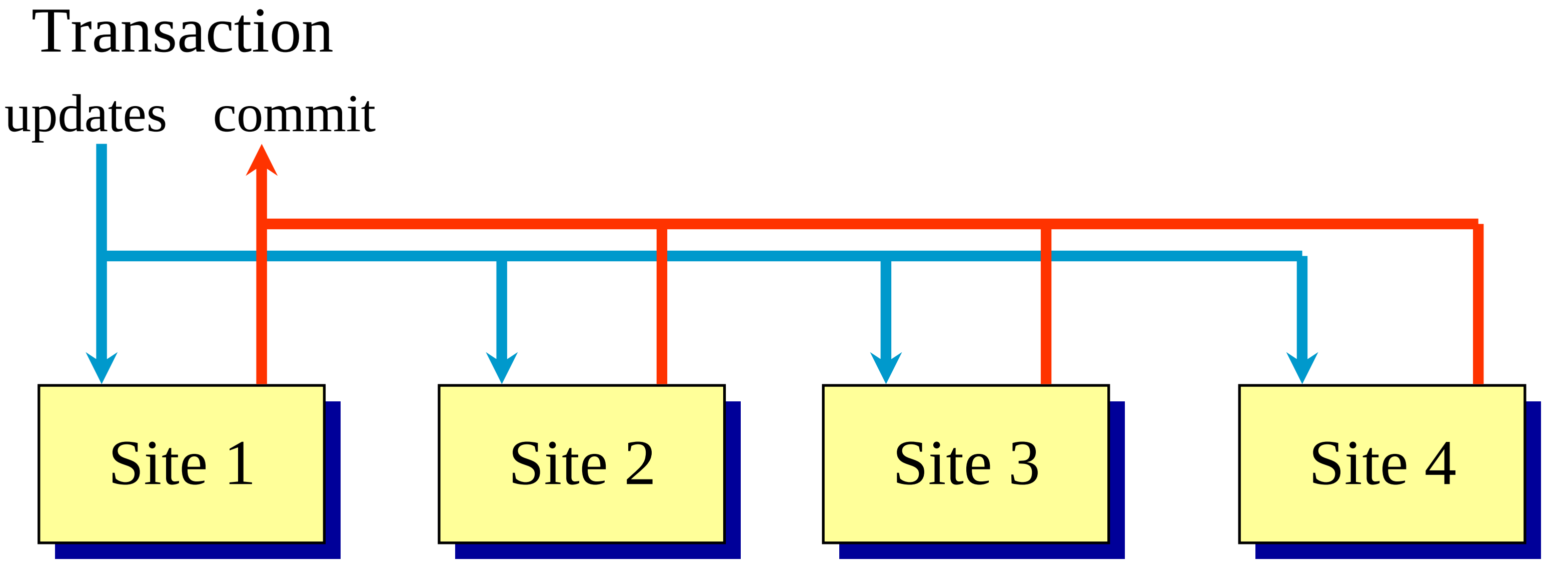
How to replicate data?

- There are two basic parameters to select when designing a replication strategy: where and when.
- Depending on when the updates are propagated:
  - Synchronous (eager)
  - Asynchronous (lazy)
- Depending on where the updates can take place:
  - Primary Copy (master)
  - Update Everywhere (group)

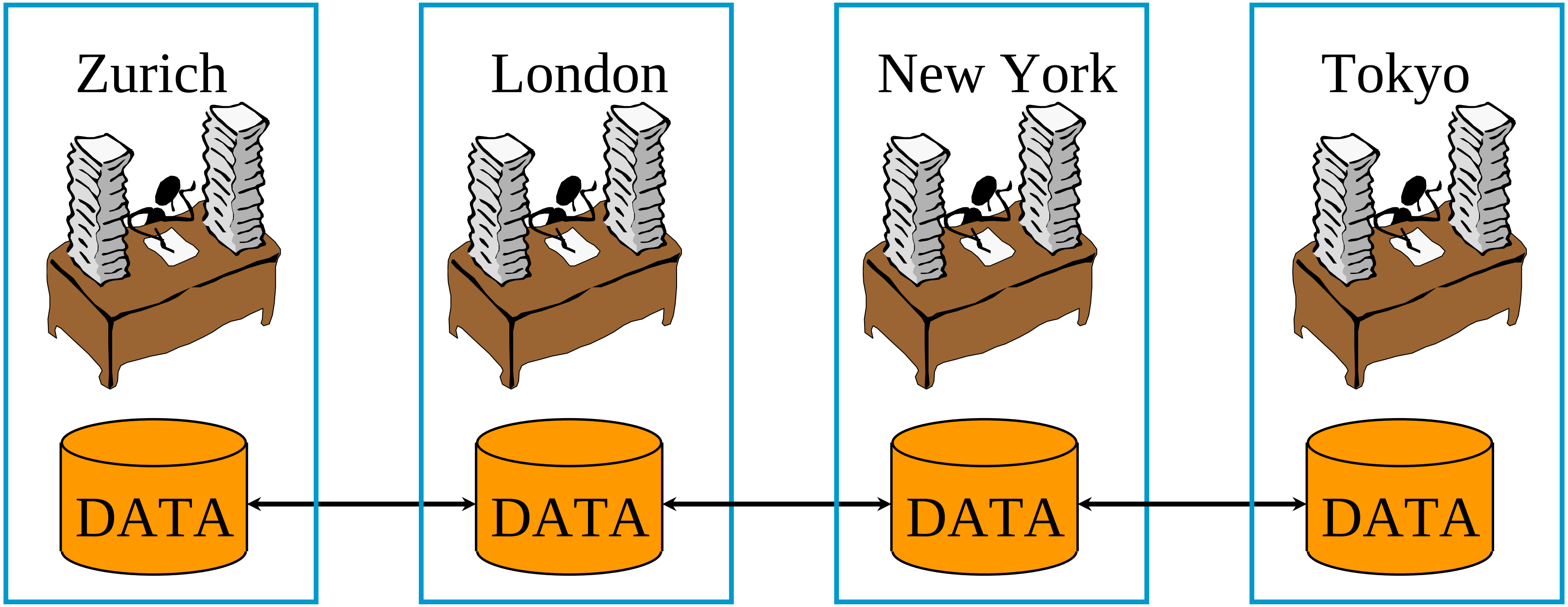
	master	group
Sync		
Async		

Synchronous Replication

- □ Synchronous replication propagates any changes to the data immediately to all existing copies. Moreover, the changes are propagated within the scope of the transaction making the changes. The ACID properties apply to all copy updates.



Synchronous Replication



Zurich

London

New York

Tokyo

DATA

DATA

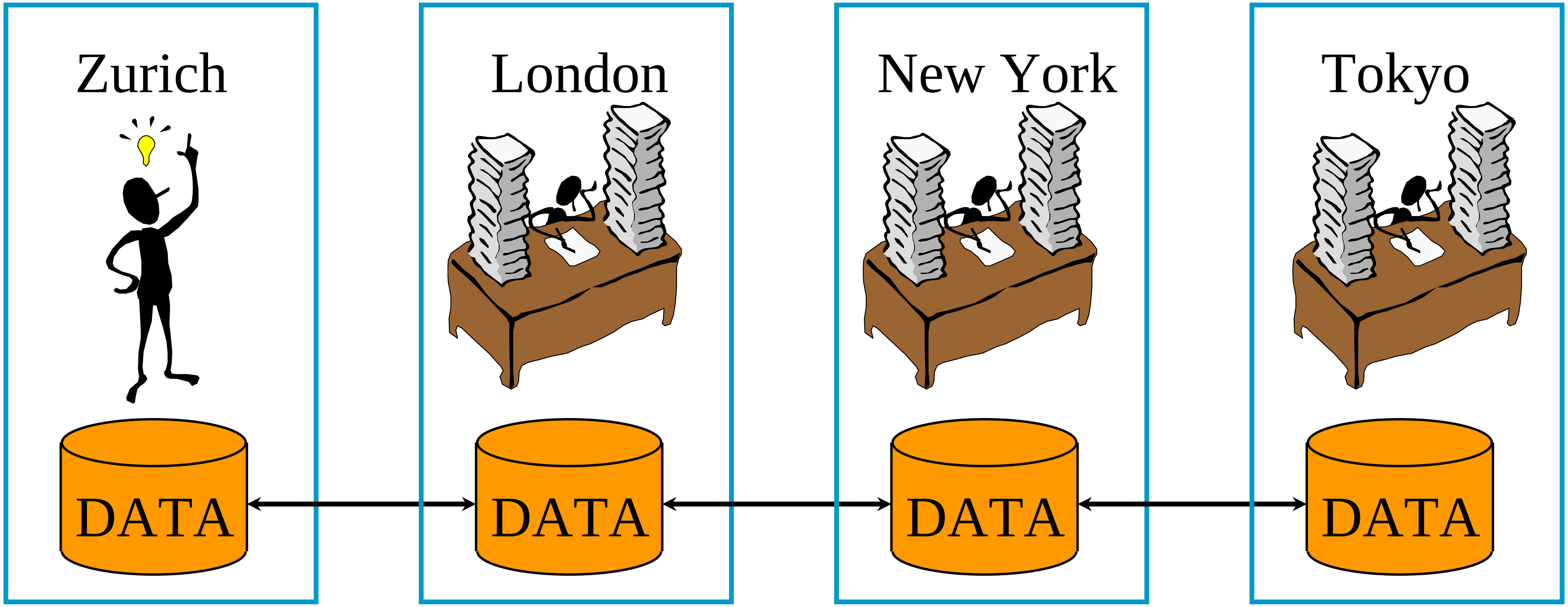
DATA

DATA

Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

DATA IS CONSISTENT AT ALL SITES

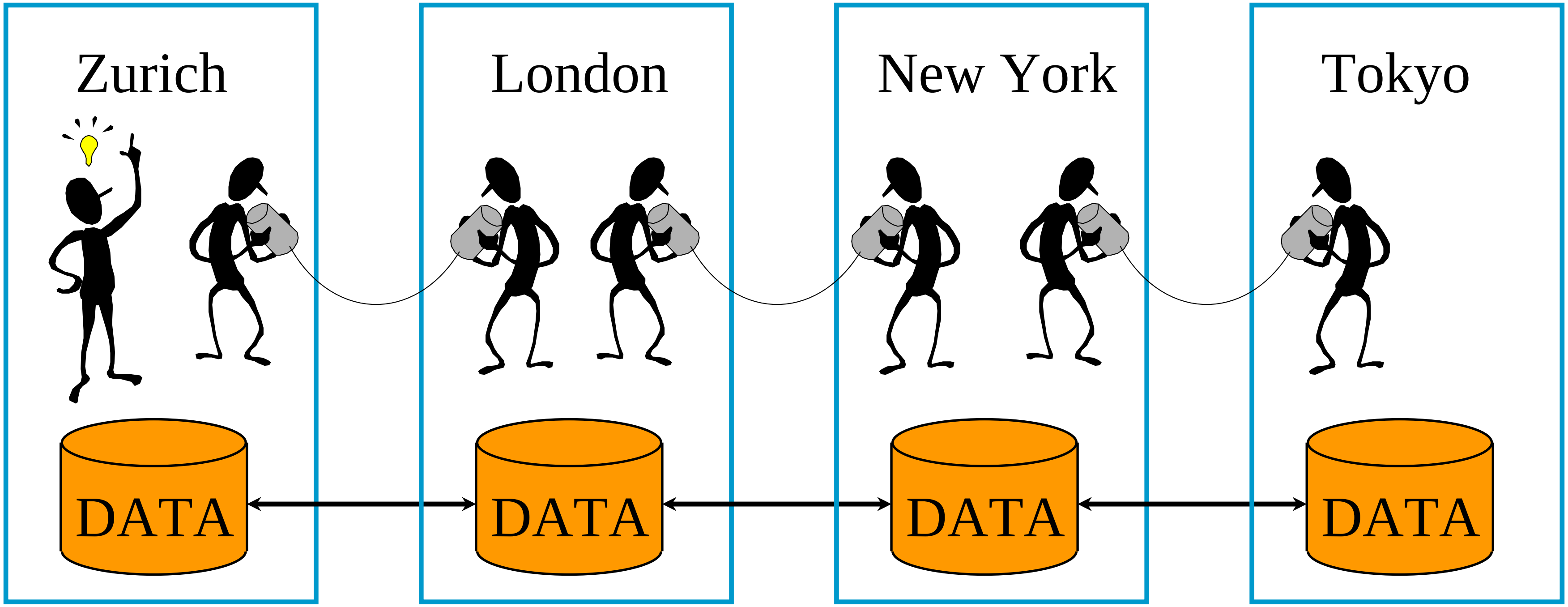
Synchronous Replication



Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

A SITE WANTS TO UPDATE THE PRICE ...

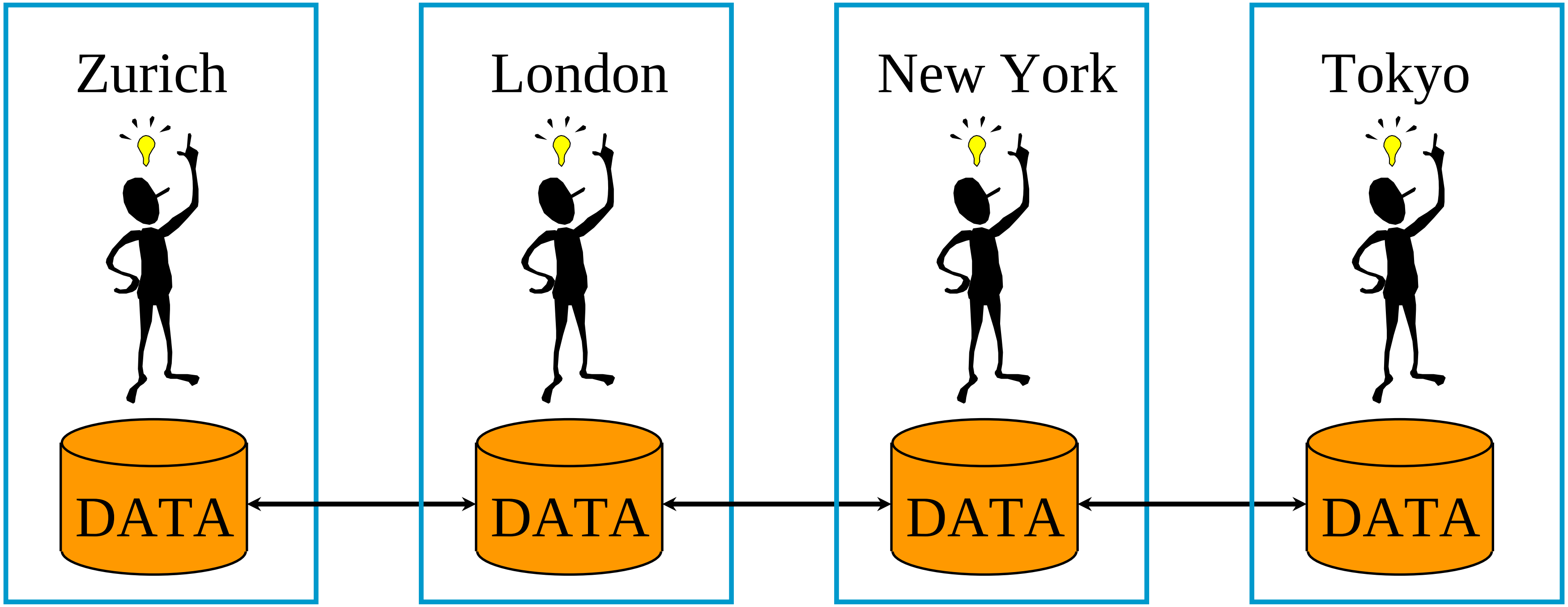
Synchronous Replication



Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

… IT FIRST CONSULTS WITH EVERYBODY ELSE ...

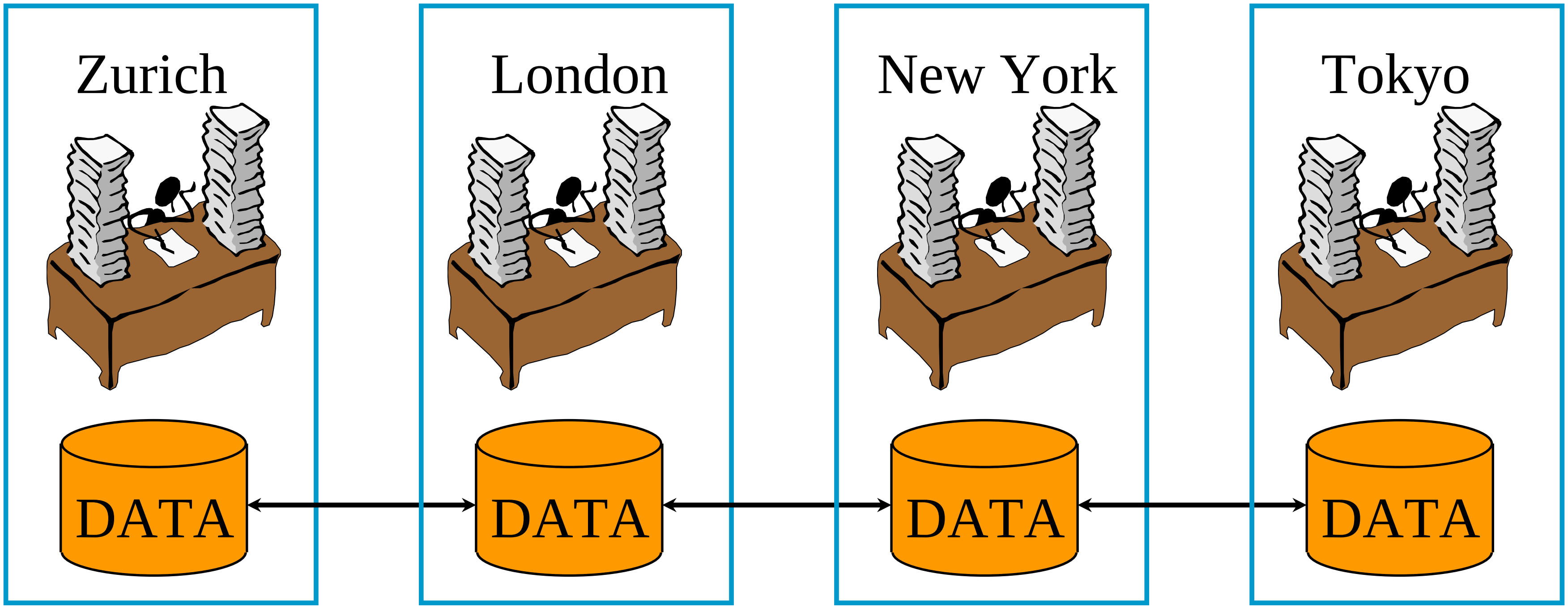
Synchronous Replication



Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

… AN AGREEMENT IS REACHED ...

Synchronous Replication

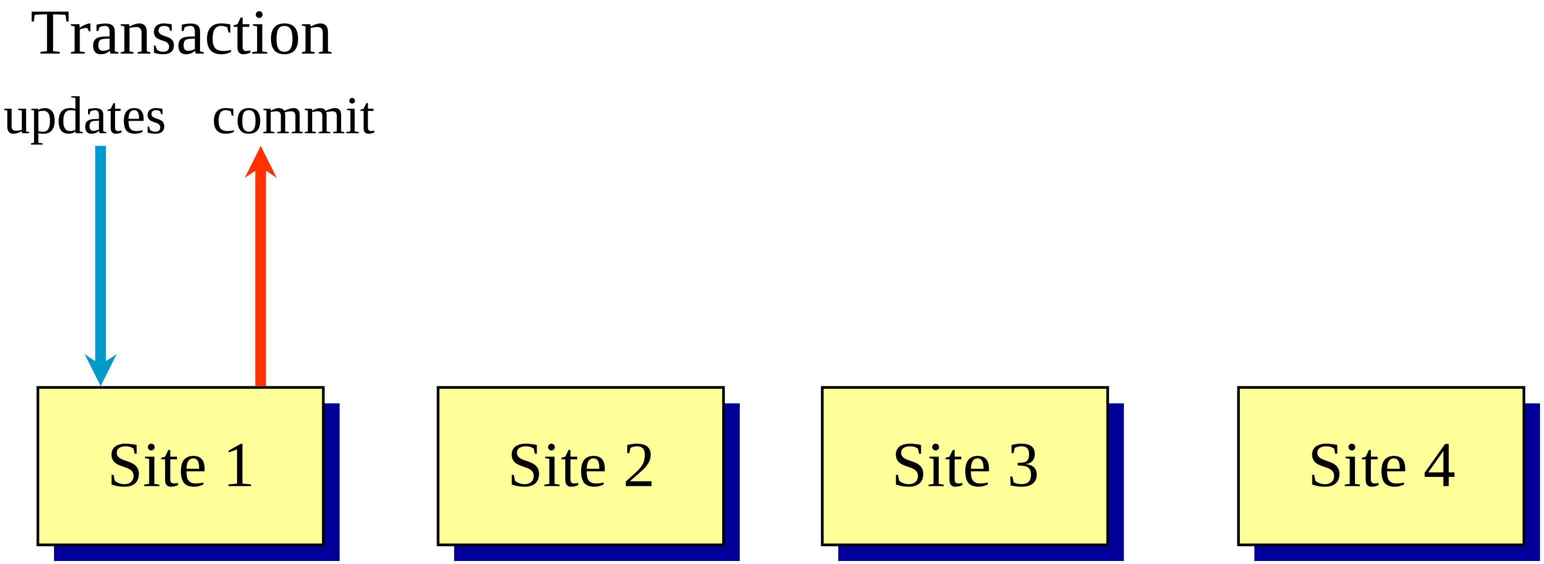


Price = $ 100 **Price = $ 100** **Price = $ 100** **Price = $ 100**

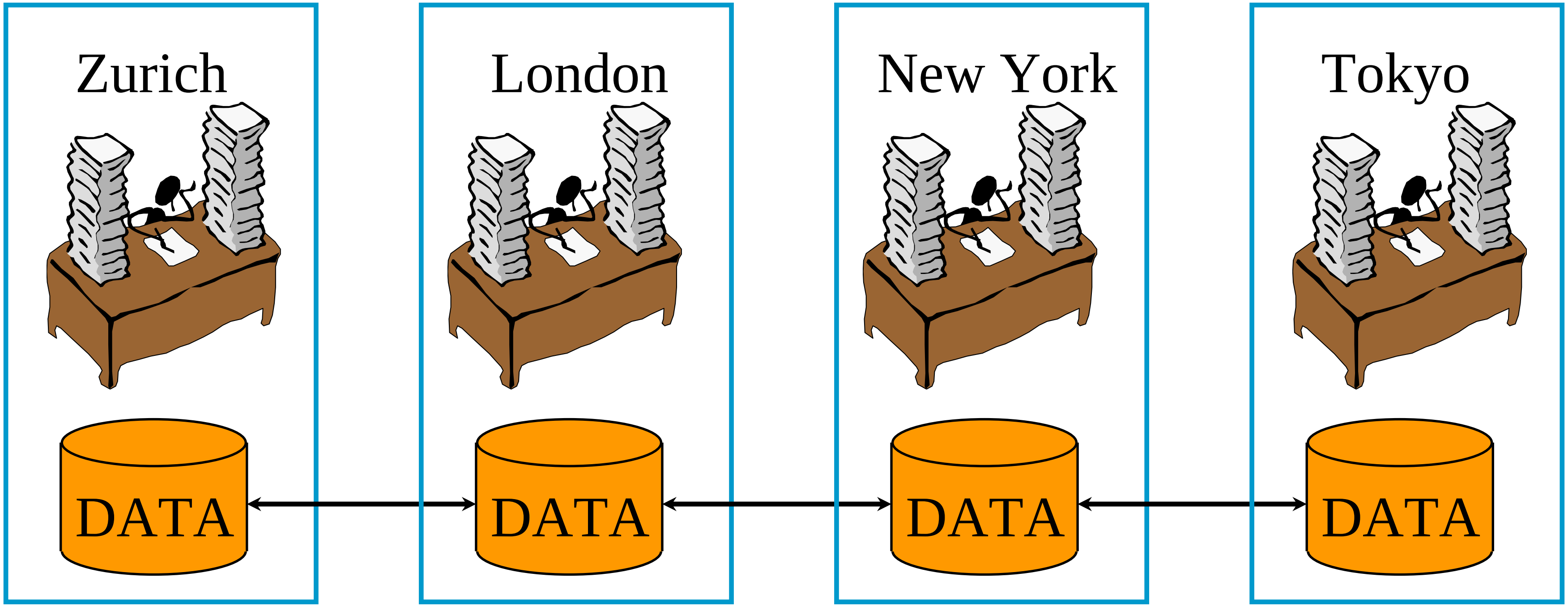
… THE PRICE IS UPDATED AND PROCESSING CONTINUES.

Asynchronous Replication

- □ Asynchronous replication first executes the updating transaction on the local copy. Then the changes are propagated to all other copies. While the propagation takes place, the copies are inconsistent (they have different values).
- □ The time the copies are inconsistent is an adjustable parameter which is application dependent.



Asynchronous Replication



Zurich

London

New York

Tokyo

DATA

DATA

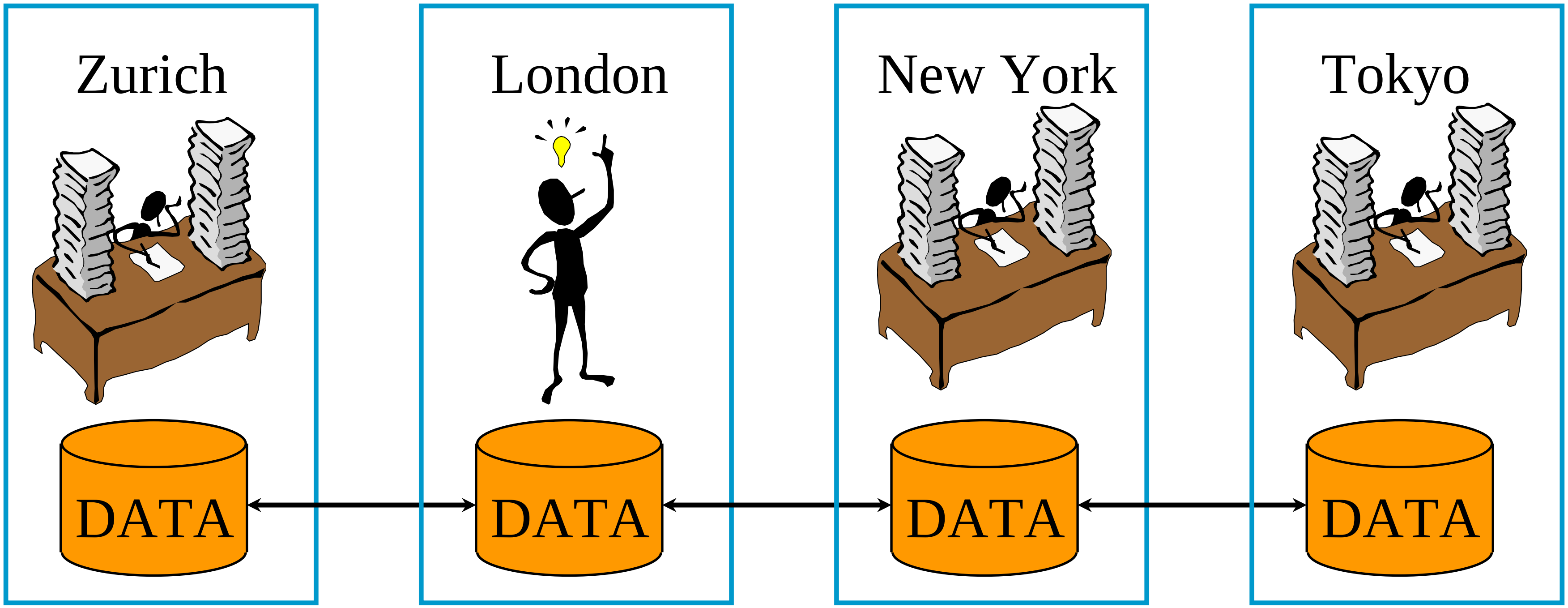
DATA

DATA

Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

DATA IS CONSISTENT AT ALL SITES

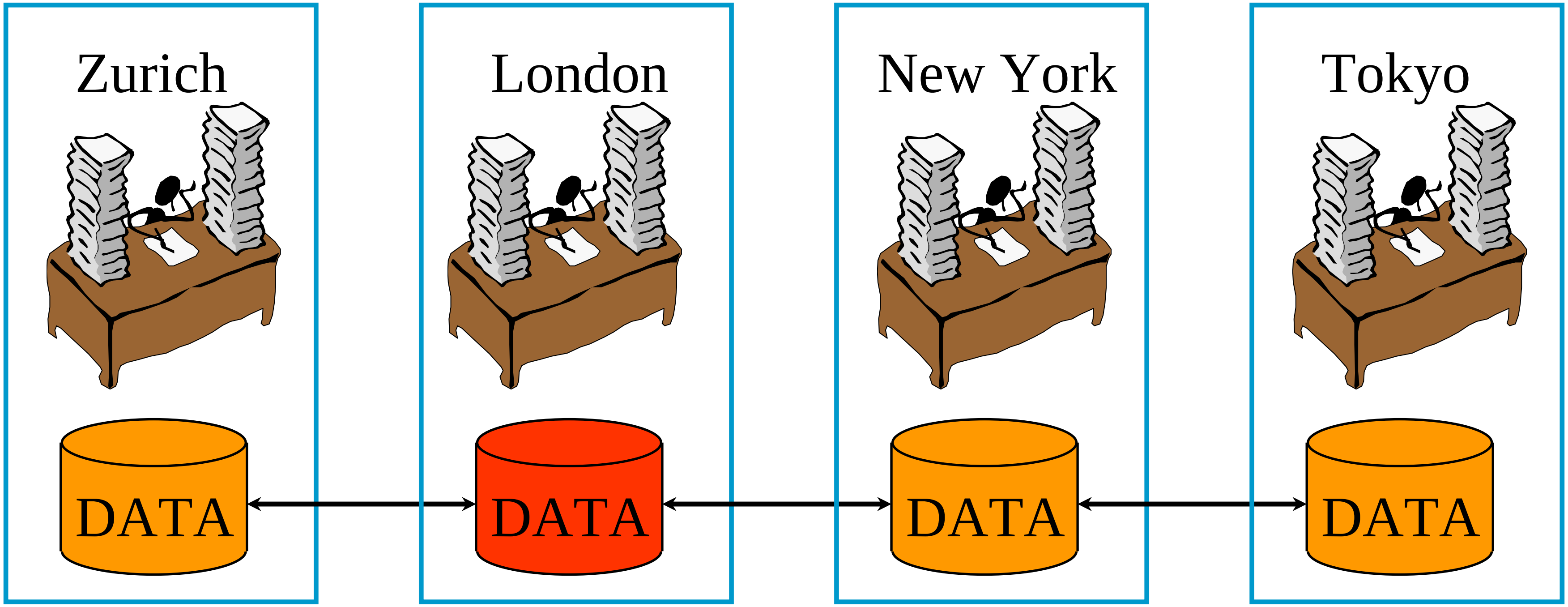
Asynchronous Replication



Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

A SITE WANTS TO UPDATE THE PRICE ...

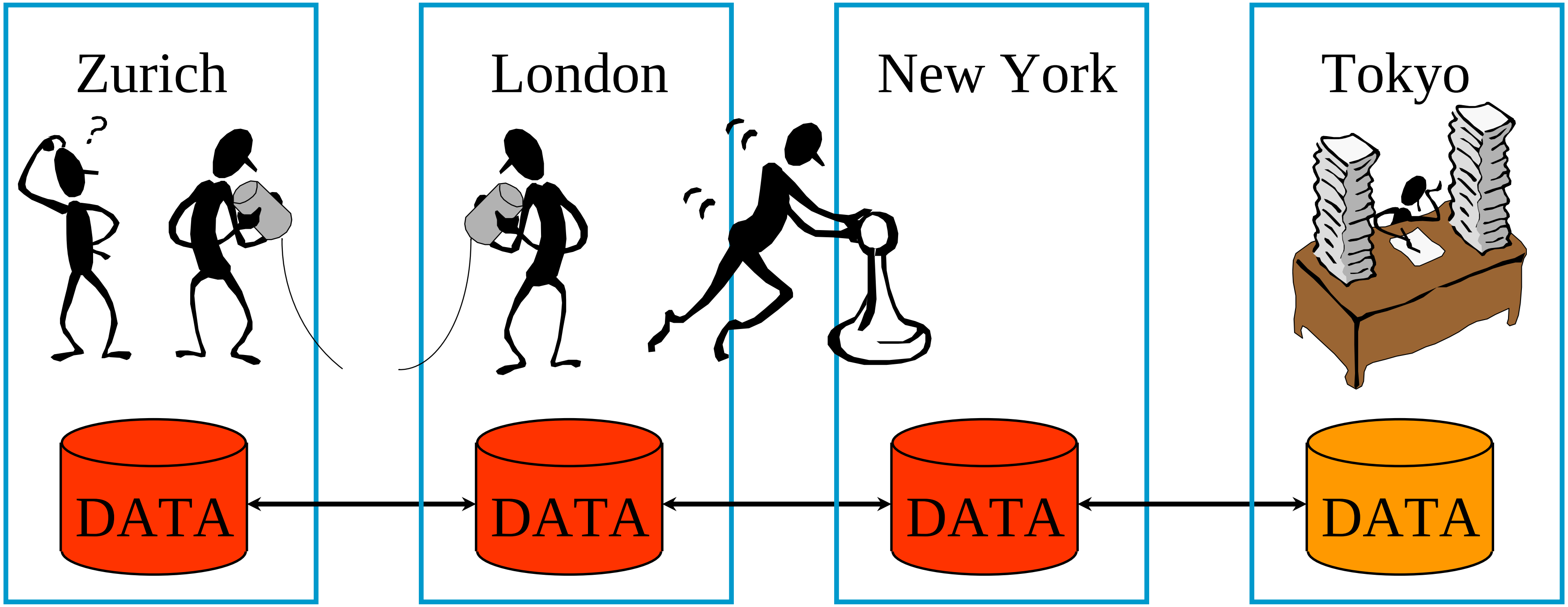
Asynchronous Replication



Price = $ 50 **Price = $ 100** **Price = $ 50** **Price = $ 50**

THEN IT UPDATES THE PRICE LOCALLY AND CONTINUES PROCESSING (DATA IS NOT CONSISTENT!)...

Asynchronous Replication

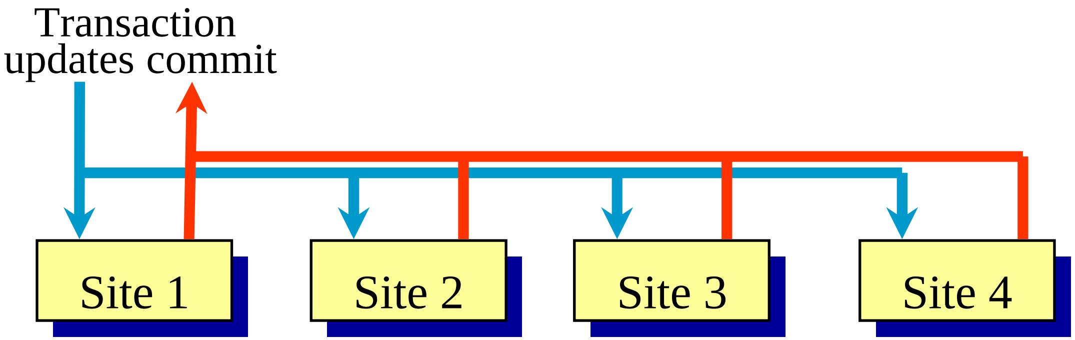


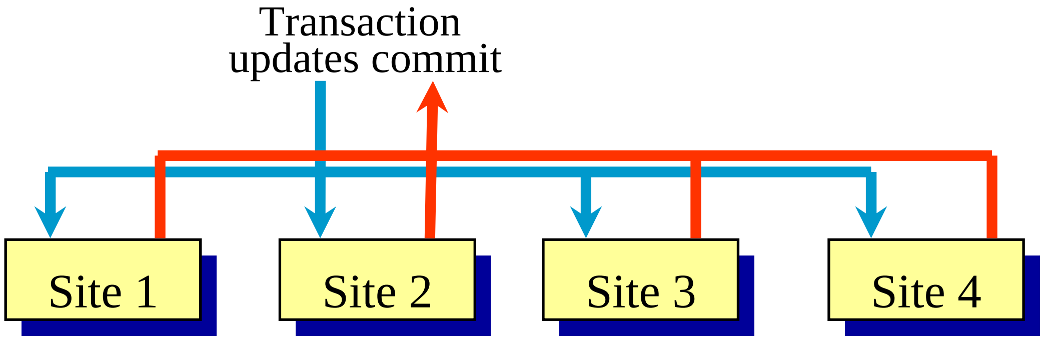
Price = $ 100 **Price = $ 100** **Price = $ 100** **Price = $ 50**

THE UPDATE IS EVENTUALLY PROPAGATED TO ALL SITES (PUSH, PULL MODELS)

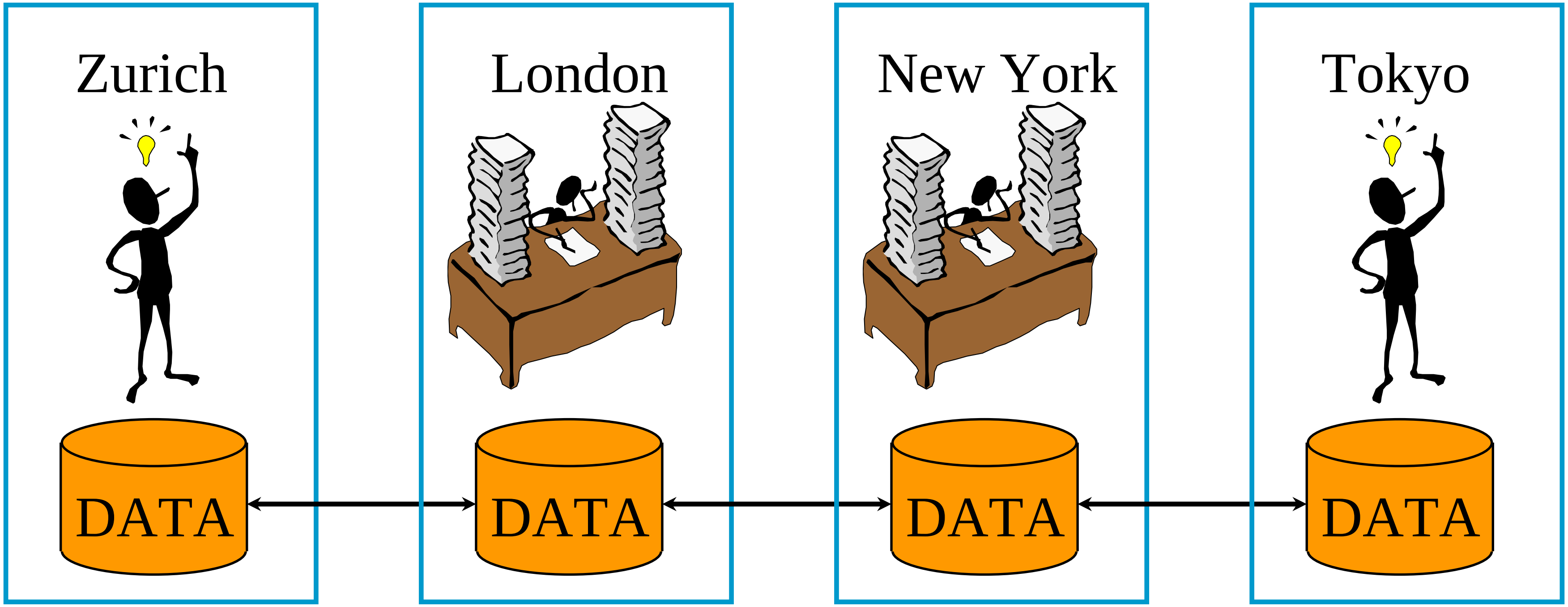
Update Everywhere

- With an update everywhere approach, changes can be initiated at any of the copies. That is, any of the sites which owns a copy can update the value of the data item





Update Everywhere

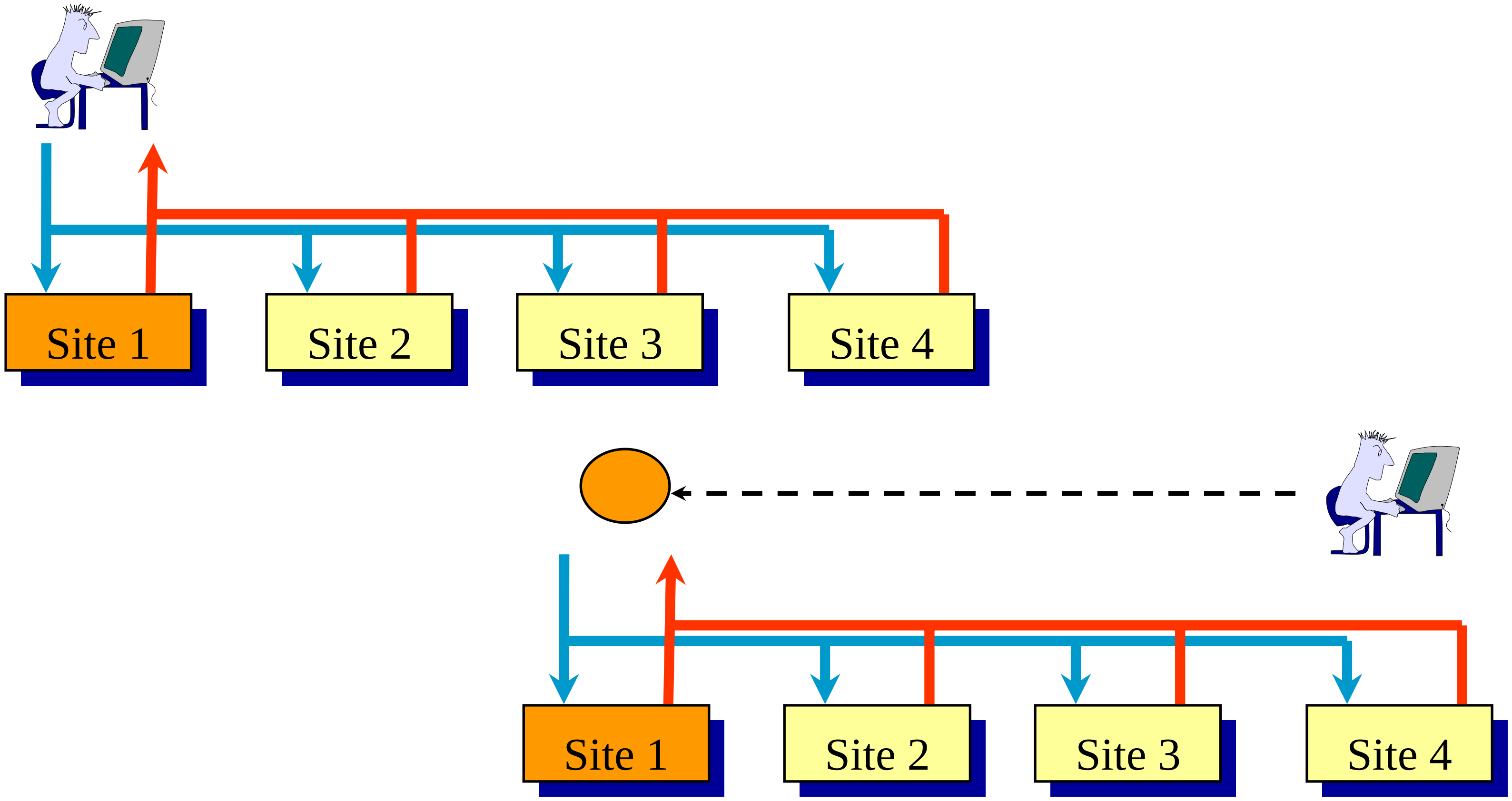


Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

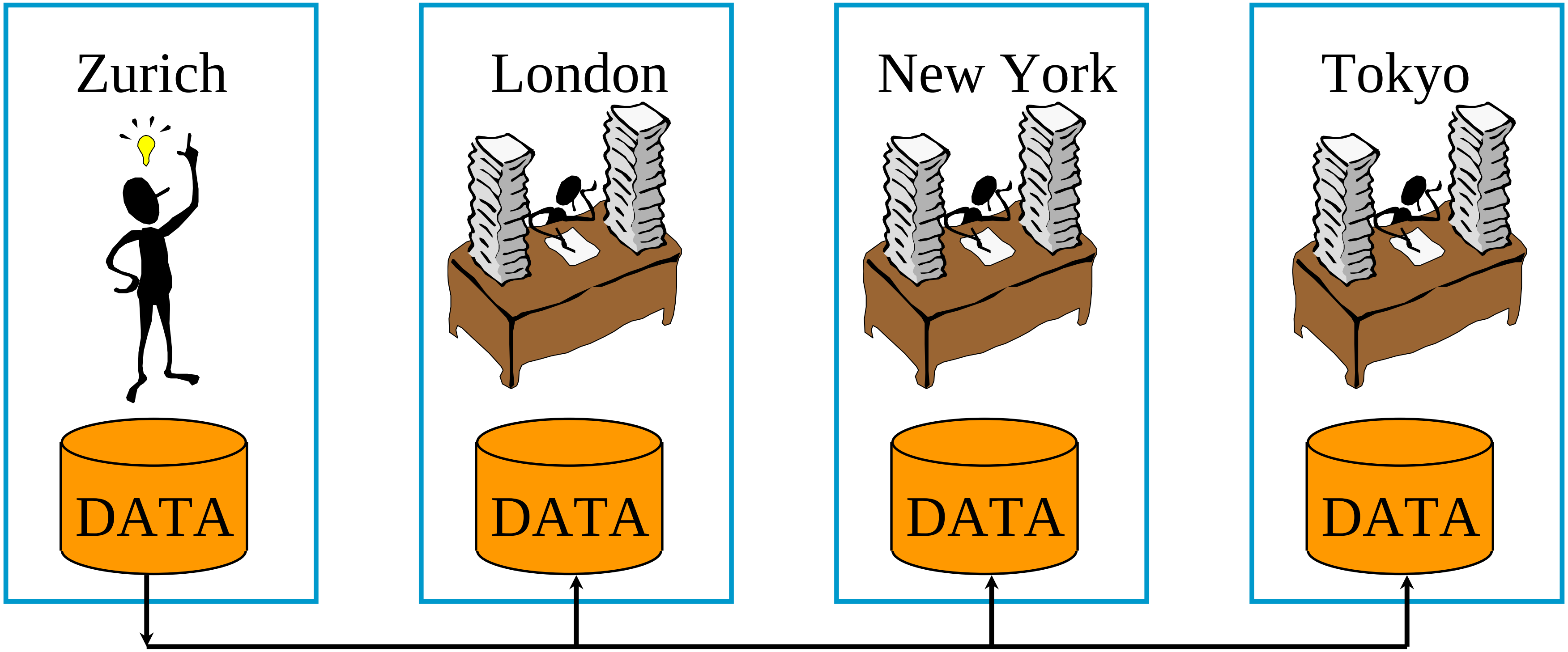
ALL SITES ARE ALLOWED TO UPDATE THEIR COPY

Primary Copy

- □ With a primary copy approach, there is only one copy which can be updated (the master), all others (secondary copies) are updated reflecting the changes to the master.



Primary Copy



Price = $ 50 **Price = $ 50** **Price = $ 50** **Price = $ 50**

**ONLY ONE SITE IS ALLOWED TO DO UPDATES,
THE OTHER ARE READ ONLY COPIES**

Forms of replication

Synchronous

- □ Advantages:
  - No inconsistencies (identical copies)
  - Reading the local copy yields the most up to date value
  - Changes are atomic
- □ Disadvantages: A transaction has to update all sites (longer execution time, worse response time)

Asynchronous

- □ Advantages: A transaction is always local (good response time)
- □ Disadvantages:
  - Data inconsistencies
  - A local read does not always return the most up to date value
  - Changes to all copies are not guaranteed
  - Replication is not transparent

Update everywhere

- □ Advantages:
  - Any site can run a transaction
  - Load is evenly distributed
- □ Disadvantages:
  - Copies need to be synchronized

Primary Copy

- □ Advantages:
  - No inter-site synchronization is necessary (it takes place at the primary copy)
  - There is always one site which has all the updates
- □ Disadvantages:
  - The load at the primary copy can be quite large
  - Reading the local copy may not yield the most up to date value

Replication Strategies

The previous ideas can be combined into 4 different replication strategies:

Synchronous (eager)	**synchronous primary copy**	**synchronous update everywhere**
Asynchronous (lazy)	**asynchronous primary copy**	**asynchronous update everywhere**
	Primary copy	**Update everywhere**

Replication Strategies

	Primary copy	Update everywhere
Synchronous	Advantages: Updates do not need to be coordinated No inconsistencies Disadvantages: Longest response time Only useful with few updates Local copies are can only be read	Advantages: No inconsistencies Elegant (symmetrical solution) Disadvantages: Long response times Updates need to be coordinated
Asynchronous	Advantages: No coordination necessary Short response times Disadvantages: Local copies are not up to date Inconsistencies	Advantages: No centralized coordination Shortest response times Disadvantages: Inconsistencies Updates can be lost (reconciliation)

Replication (Ideal)

Synchronous (eager)	Globally correct Remote writes	Globally correct Local writes
Asynchronous (lazy)	Inconsistent reads	Inconsistent reads Reconciliation
	Primary copy	Update everywhere

Replication (Practical)

	Primary copy	Update everywhere
Synchronous (eager)	Too Expensive (usefulness?)	Too expensive (does not scale)
Asynchronous (lazy)	Feasible	Feasible in some applications

Summary - I

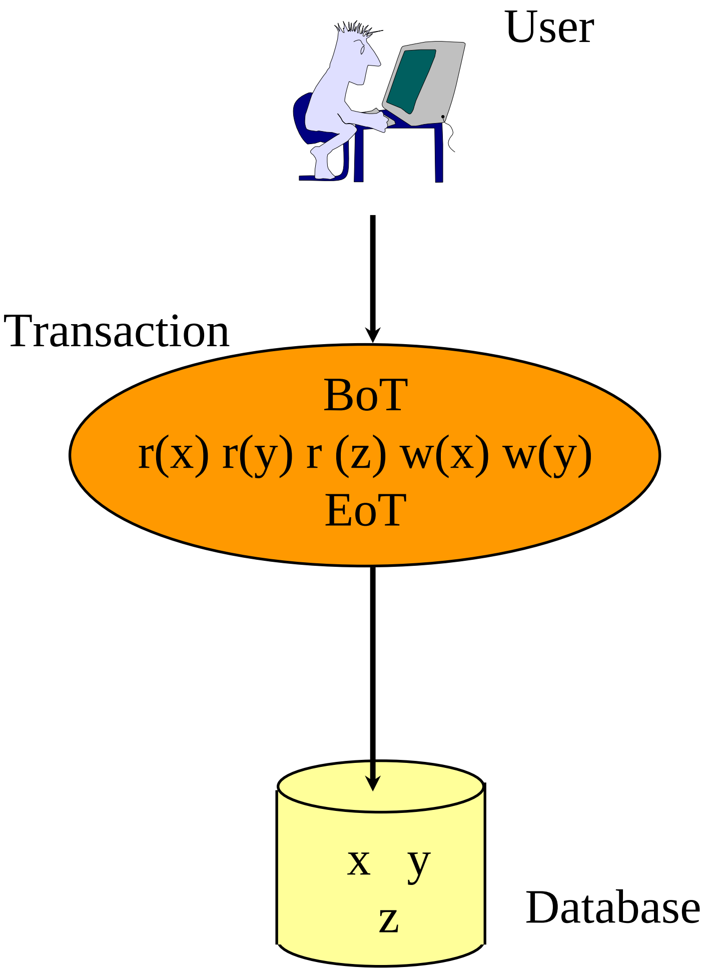
- □ Replication is used for performance and fault tolerant purposes.
- □ There are four possible strategies to implement replication solutions depending on whether it is synchronous or asynchronous, primary copy or update everywhere.
- □ Each strategy has advantages and disadvantages which are more or less obvious given the way they work.
- □ There seems to be a trade-off between correctness (data consistency) and performance (throughput and response time).
- □ The next step is to analyze these strategies in more detail to better understand how they work and where the problems lie.

Database Replication Strategies

- □ Database environments
- □ Managing replication
- □ Technical aspects and correctness/performance issues of each replication strategy:
    - Synchronous - primary copy
    - Synchronous - update everywhere
    - Asynchronous - primary copy
    - Asynchronous - update everywhere

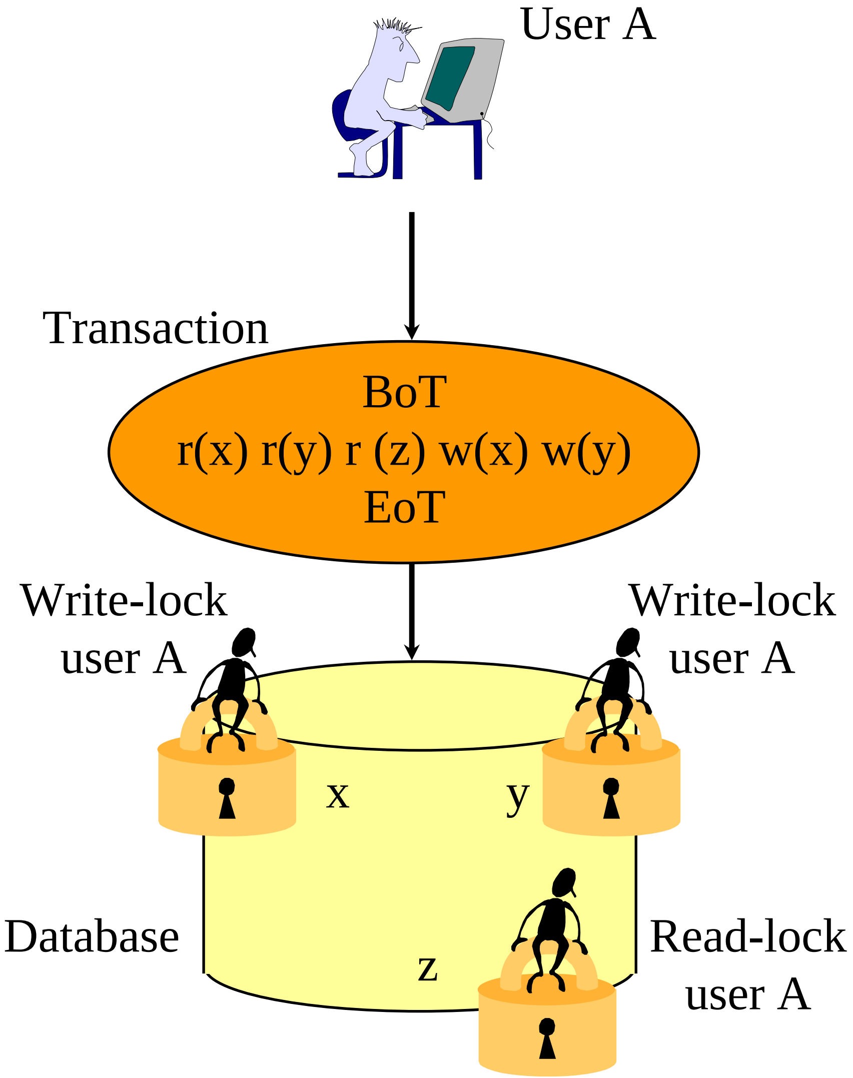
Basic Database Notation

- □ A user interacts with the database by issuing read and write operations.
- □ These read and write operations are grouped into transactions with the following properties:
  - Atomicity: either all of the transaction is executed or nothing at all.
  - Consistency: the transaction produces consistent changes.
  - Isolation: transactions do not interfere with each other.
  - Durability: Once the transaction commits, its changes remain.



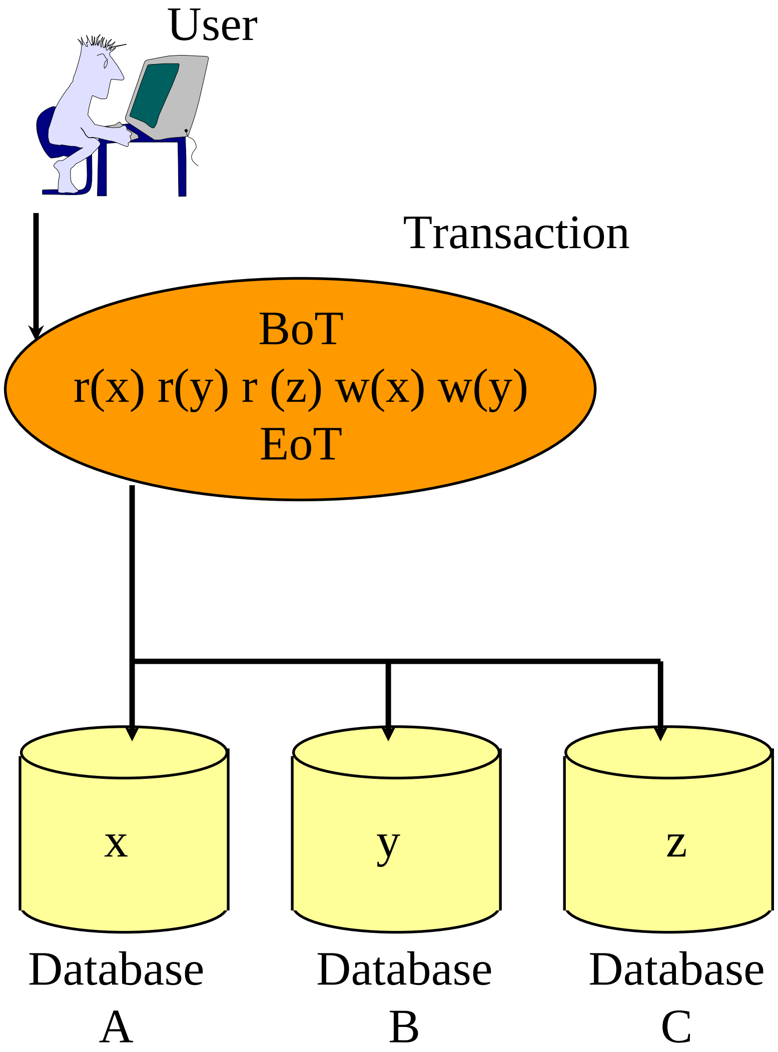
Isolation

- □ Isolation is guaranteed by a concurrency control protocol.
- □ In commercial databases, this is usually 2 Phase Locking (2PL):
  - conflicting locks cannot coexist (writes conflict with reads and writes on the same item)
  - Before accessing an item, the item must be locked.
  - After releasing a lock, a transaction cannot obtain any more locks.



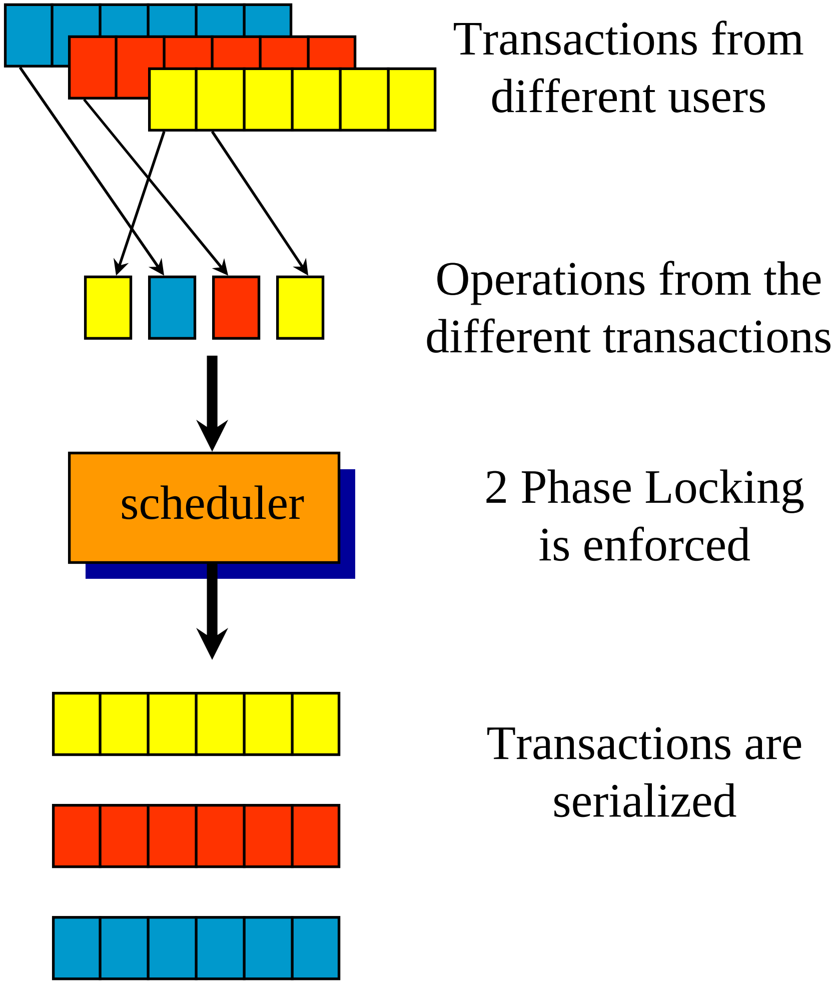
Atomicity

- □ A transaction must commit all its changes.
- □ When a transaction executes at various sites, it must execute an atomic commitment protocol, i.e., it must commit at all sites or at none of them.
- □ Commercial systems use 2 Phase Commit:
  - A coordinator asks everybody whether they want to commit
  - If everybody agrees, the coordinator sends a message indicating they can all commit



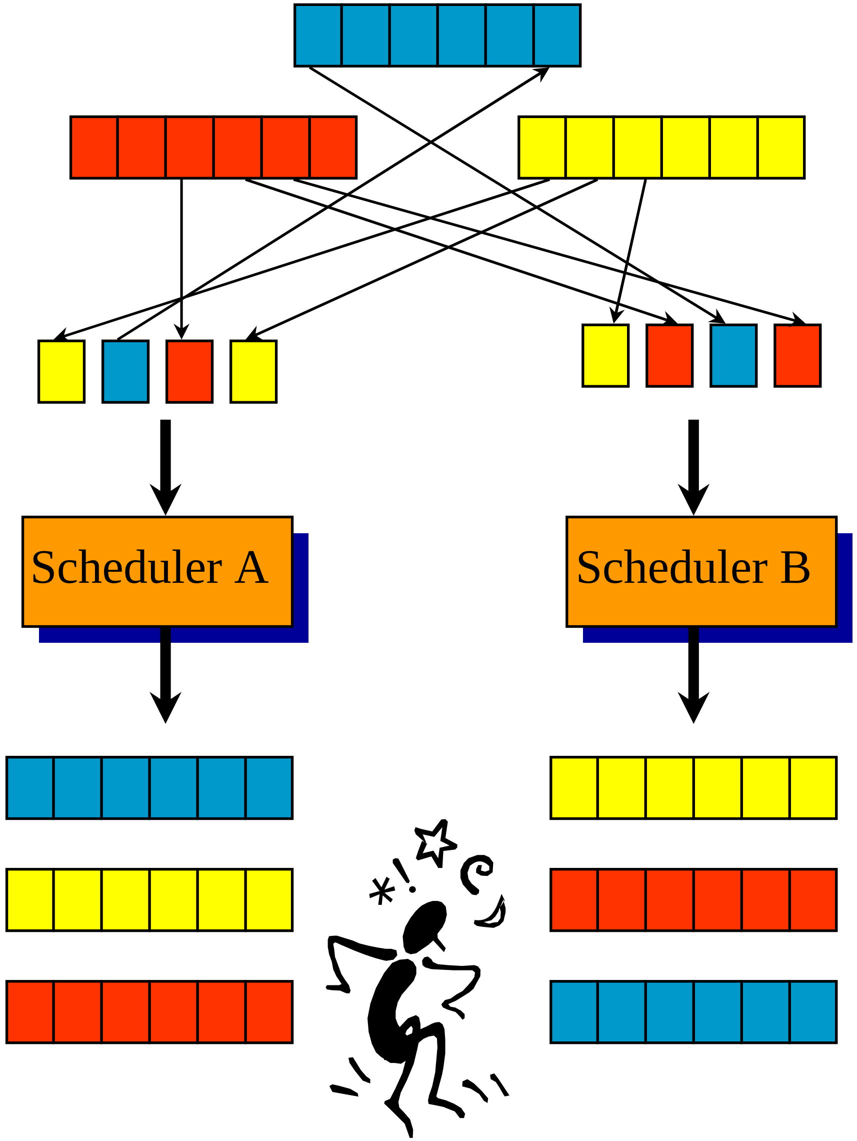
Transaction Manager

- □ The transaction manager takes care of isolation and atomicity.
- □ It acquires locks on behalf of all transactions and tries to come up with a serializable execution, i.e., make it look like the transactions were executed one after the other.
- □ If the transactions follow 2 Phase Locking, serializability is guaranteed. Thus, the scheduler only needs to enforce 2PL behaviour.


Transactions from different users
Operations from the different transactions
scheduler
2 Phase Locking is enforced
Transactions are serialized

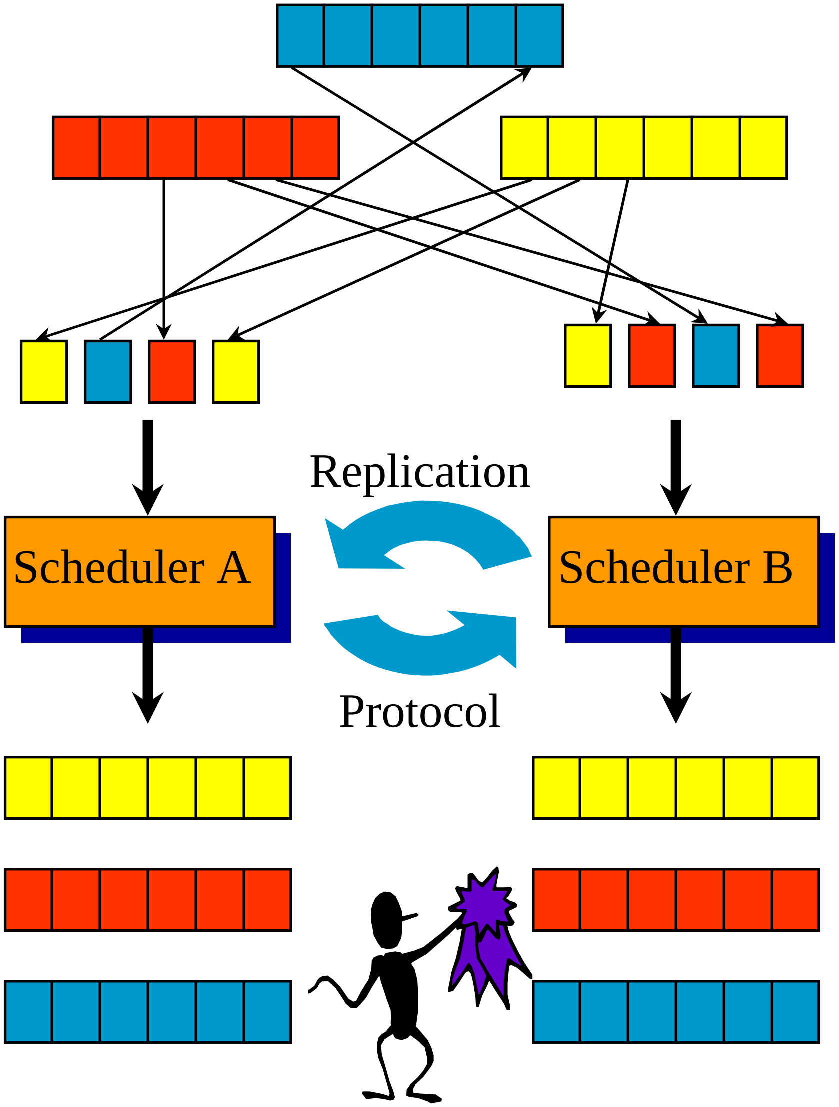
Managing Replication

- □ When the data is replicated, we still need to guarantee atomicity and isolation.
- □ Atomicity can be guaranteed by using 2 Phase Commit. This is the easy part.
- □ The problem is how to make sure the serialization orders are the same at all sites, i.e., make sure that all sites do the same things in the same order (otherwise the copies would be inconsistent).



Managing Replication

- □ To avoid this, replication protocols are used.
- □ A replication protocol specifies how the different sites must be coordinated in order to provide a concrete set of guarantees.
- □ The replication protocols depend on the replication strategy (synchronous, asynchronous, primary copy, update everywhere).

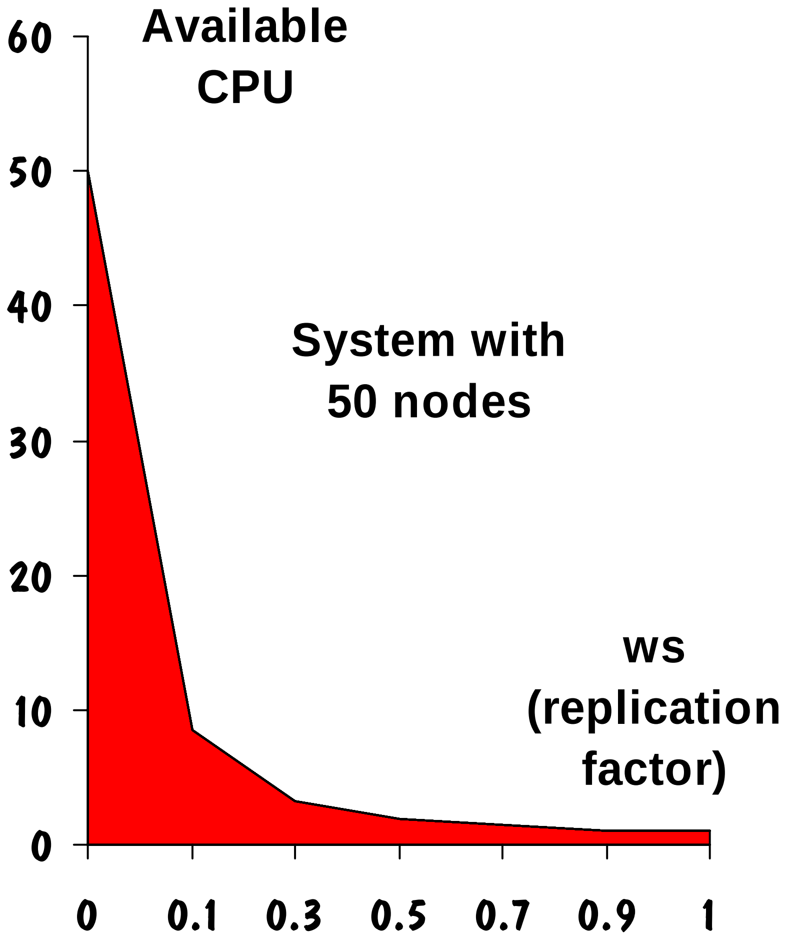


Replication Strategies

Now we can analyze the advantages and disadvantages of each strategy:

	Primary copy	Update everywhere
Synchronous (eager)	synchronous primary copy	synchronous update everywhere
Asynchronous (lazy)	asynchronous primary copy	asynchronous update everywhere

Cost of Replication



- □ Assume a 50 node replicated system where a fraction s of the data is replicated and w represents the fraction of updates made (ws = replication factor)
- □ Overall computing power of the system:

$$\frac{N}{1 + w \cdot s \cdot (N - 1)}$$

- □ No performance gain with large ws factor (many updates or many replicated data items)
- □ Reads must be local to get performance advantages.

Synchronous - update everywhere

Assume all sites contain the same data.

<u>READ ONE-WRITE ALL</u>

- □ Each sites uses 2 Phase Locking.
- □ Read operations are performed locally.
- □ Write operations are performed at all sites (using a distributed locking protocol).

This protocol guarantees that every site will behave as if there were only one database. The execution is serializable (correct) and all reads access the latest version.

This simple protocol illustrates the main idea behind replication, but it needs to be extended in order to cope with realistic environments:

- □ Sites fail, which reduces the availability (if a site fails, no copy can be written).
- □ Sites eventually have to recover (a recently recovered site may not have the latest updates).

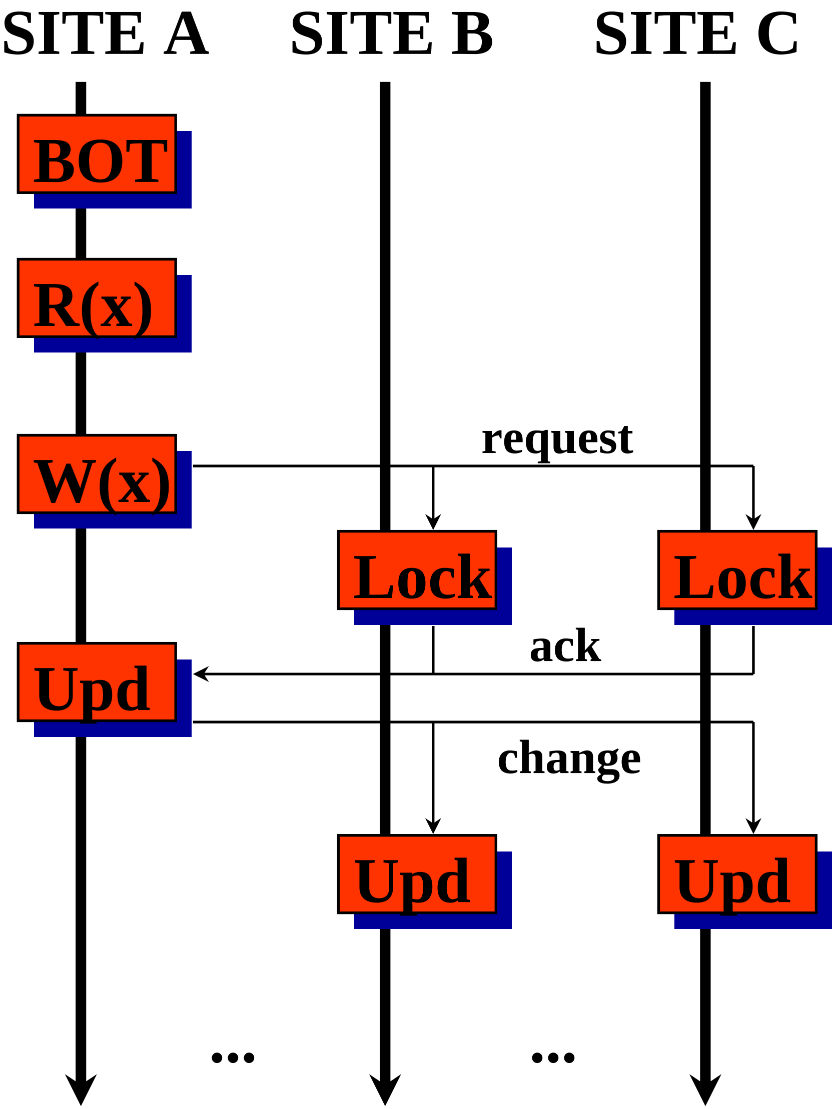
Dealing with Site Failures

Assume, for the moment, that there are no communication failures. Instead of writing to all copies, we could

WRITE ALL AVAILABLE COPIES

- □ READ = read any copy, if time-out, read another copy.
- □ WRITE = send Write(x) to all copies. If one site rejects the operation, then abort. Otherwise, all sites not responding are "missing writes".
- □ VALIDATION = To commit a transaction
  - Check that all sites in "missing writes" are still down. If not, then abort the transaction.
  - Check that all sites that were available are still available. If some do not respond, then abort.

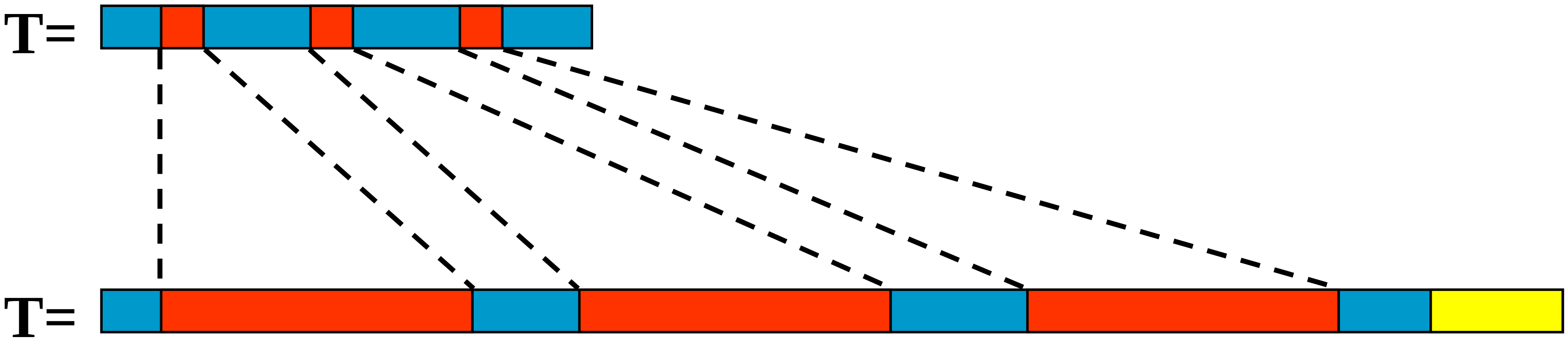
Synchronous - Update Everywhere Protocol



- □ Each site uses 2PL
- □ Read operations are performed locally
- □ Write operations involve locking all copies of the data item (request a lock, obtain the lock, receive an acknowledgement)
- □ The transaction is committed using 2PC
- □ Main optimizations are based on the idea of quorums (but all we will say about this protocol also applies to quorums)

Response Time and Messages

centralized database 🟥 update

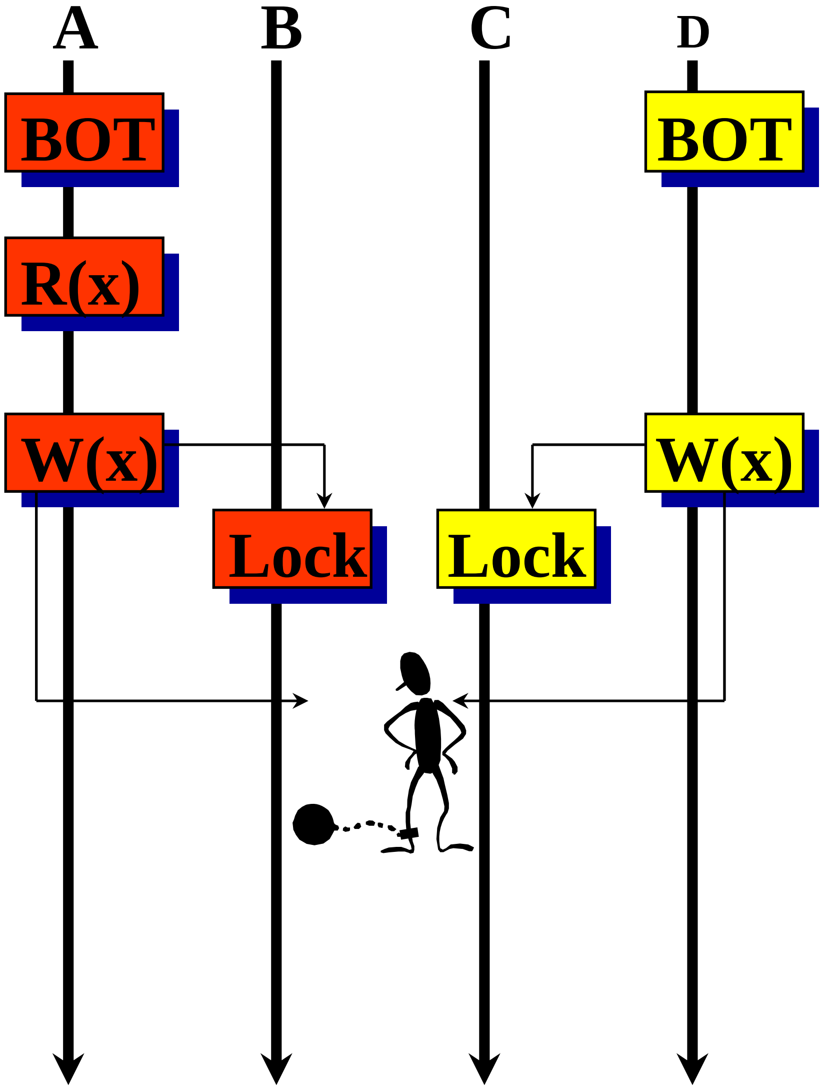


replicated database 🟥 update: 2N messages

🟨 2PC

The way replication takes place (one operation at a time), increases the response time and, thereby, the conflict profile of the transaction. The message overhead is too high (even if broadcast facilities are available).

The Deadlock Problem



□ Approximated deadlock rate:

$$\frac{TPS^2 \cdot Action\_Time \cdot Actions^5 \cdot N^3}{4 \cdot DB\_Size^2}$$

if the database size remains constant, or

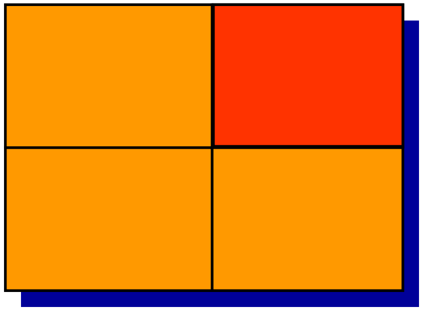
$$\frac{TPS^2 \cdot Action\_Time \cdot Actions^5 \cdot N}{4 \cdot DB\_Size^2}$$

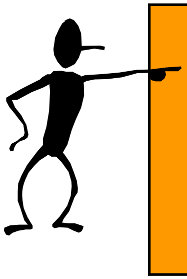
if the database size grows with the number of nodes.

□ Optimistic approaches may result in too many aborts.

Synchronous - update everywhere

- □ Advantages:
  - No inconsistencies
  - Elegant (symmetrical solution)
- □ Disadvantages:
  - Very high number of messages involved
  - Transaction response time is very long
  - The system will not scale because of deadlocks (as the number of nodes increases, the probability of getting into a deadlock gets too high)

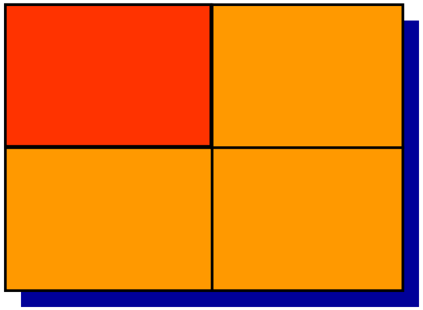


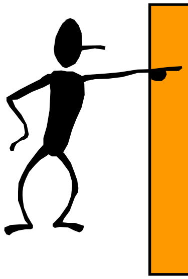


Data consistency is guaranteed. Performance may be seriously affected with this strategy. The system may also have scalability problems (deadlocks). High fault tolerance.

Synchronous - primary copy

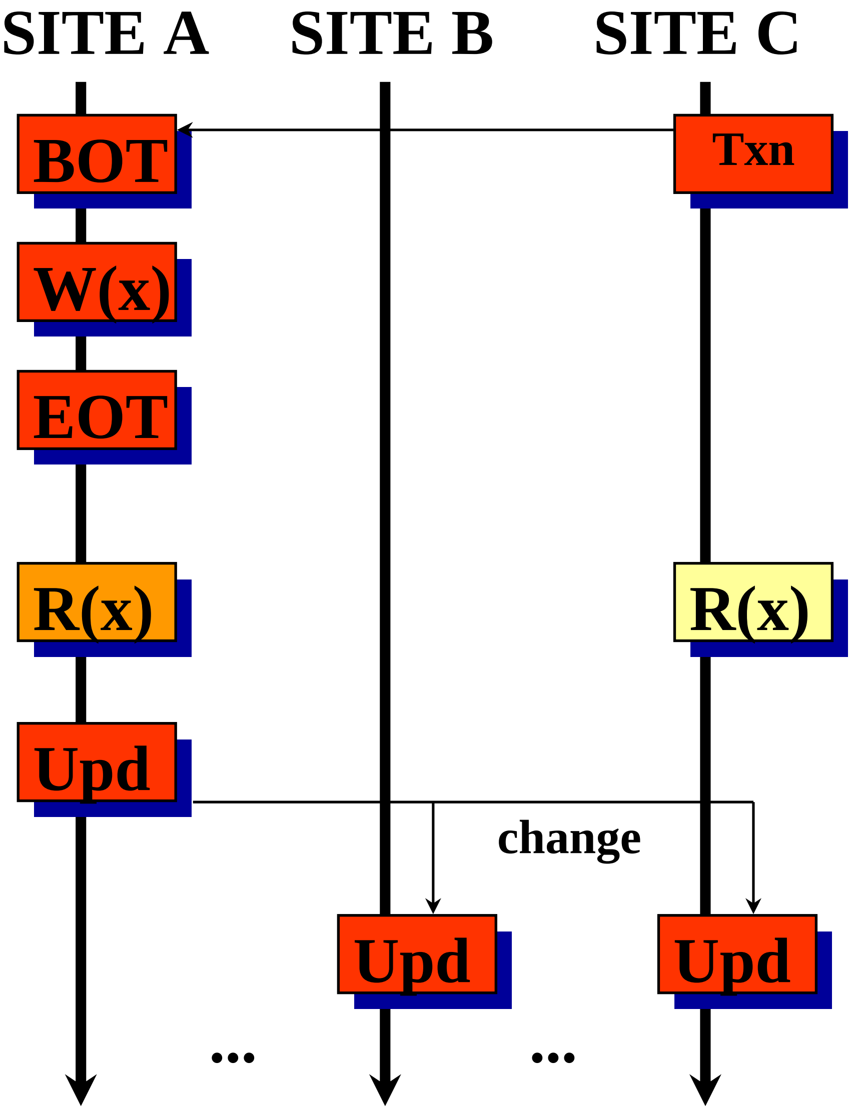
- □ Advantages:
  - ➜ Updates do not need to be coordinated
  - ➜ No inconsistencies, no deadlocks.
- □ Disadvantages:
  - ➜ Longest response time
  - ➜ Only useful with few updates (primary copy is a bottleneck)
  - ➜ Local copies are almost useless
  - ➜ Not used in practice





Similar problems to those of Sync - update everywhere. Including scalability problems (bottlenecks). Data consistency is guaranteed. Fault tolerant.

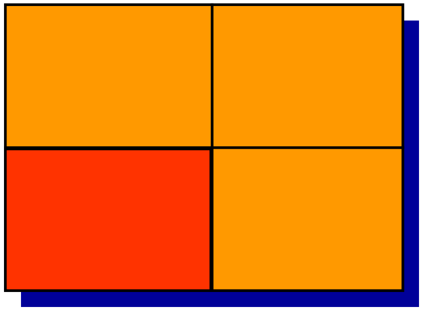
Async - primary copy protocol



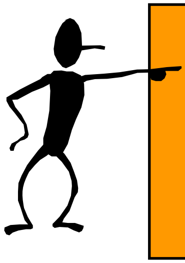
- □ Update transactions are executed at the primary copy site
- □ Read transactions are executed locally
- □ After the transaction is executed, the changes are propagated to all other sites
- □ Locally, the primary copy site uses 2 Phase Locking
- □ In this scenario, there is no atomic commitment problem (the other sites are not updated until later)

Asynchronous - primary copy

- Advantages:
  - No coordination necessary
  - Short response times (transaction is local)

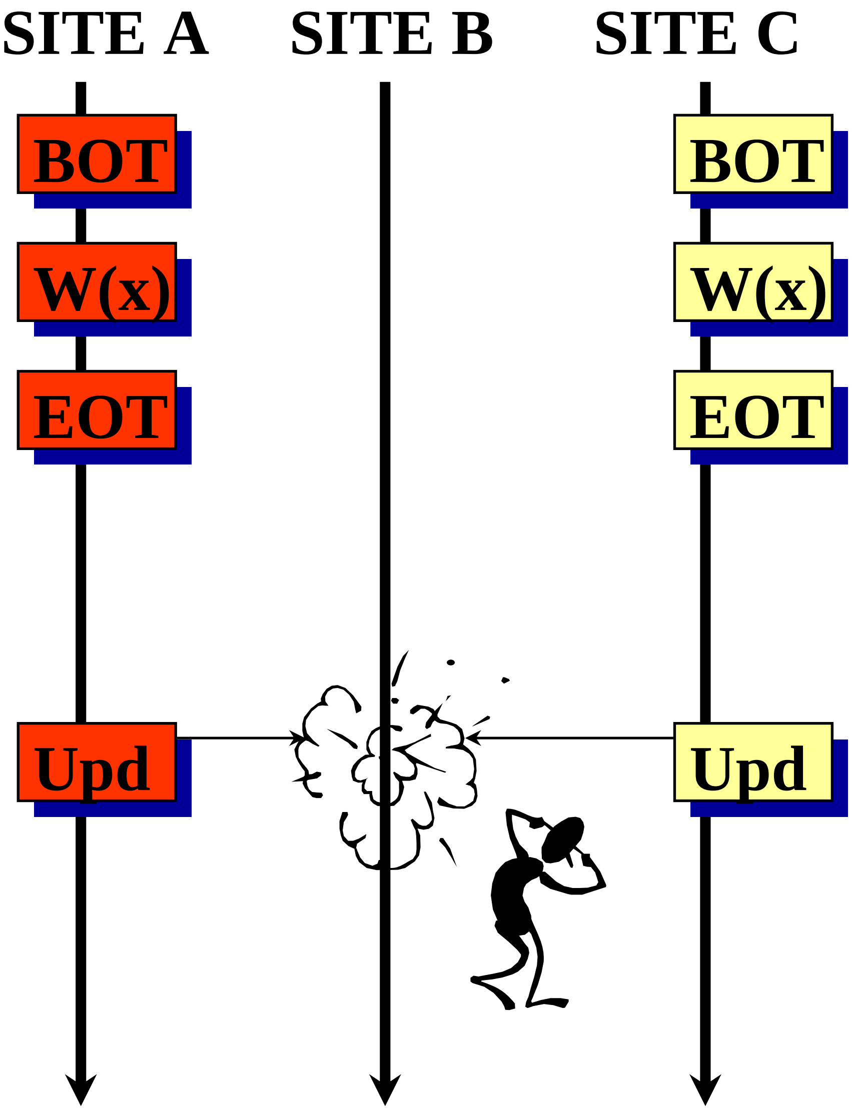


- Disadvantages:
  - Local copies are not up to date (a local read will not always include the updates made at the local copy)
  - Inconsistencies (different sites have different values of the same data item)



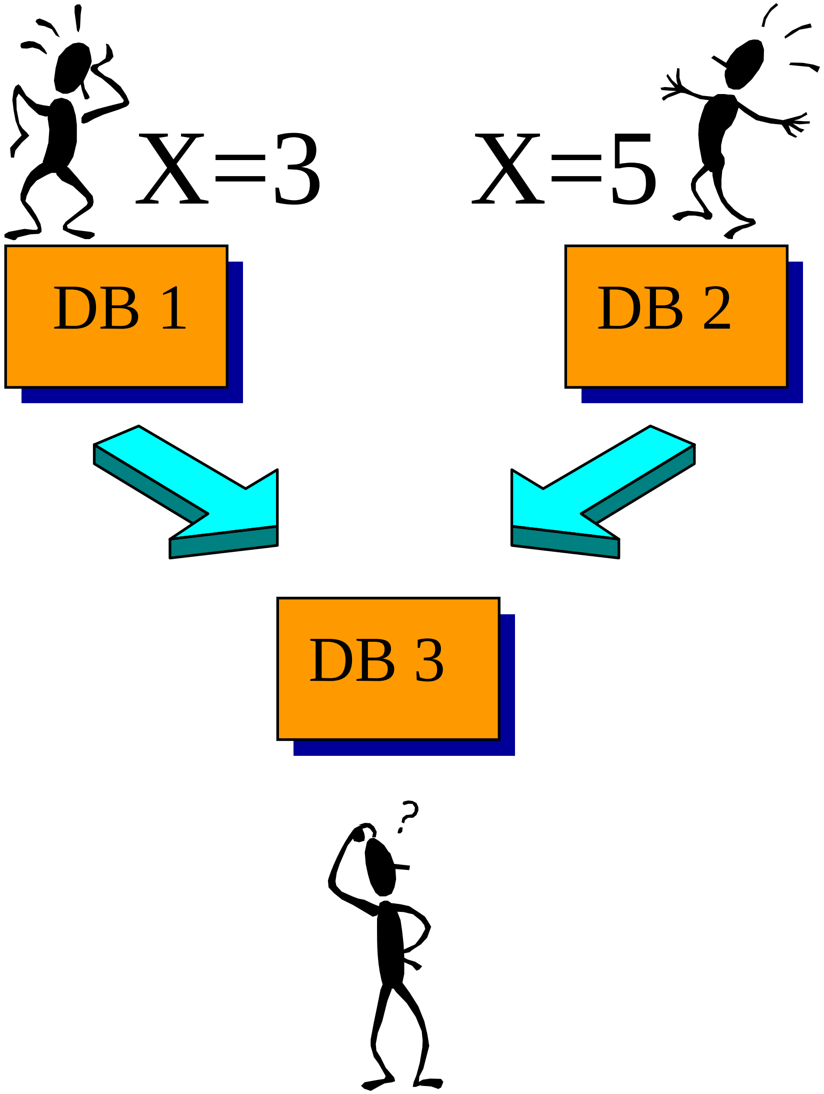
Performance is good (almost same as if no replication). Fault tolerance is limited. Data inconsistencies arise.

Async - update everywhere protocol



- □ All transactions are executed locally
- □ After the transaction is executed, the changes are propagated to all other sites
- □ Locally, a site uses 2 Phase Locking
- □ In this scenario, there is no atomic commitment problem (the other sites are not updated until later)
- □ However, unlike with primary copy, updates need to be coordinated

Async / Update Everywhere



X=3

X=5

DB 1

DB 2

DB 3

- □ Probability of needing reconciliation:

$$\frac{\mathbf{TPS}^2 \cdot \mathbf{Action\_time} \cdot \mathbf{Actions}^3 \cdot \mathbf{N}^3}{\mathbf{2} \cdot \mathbf{DB\_Size}}$$

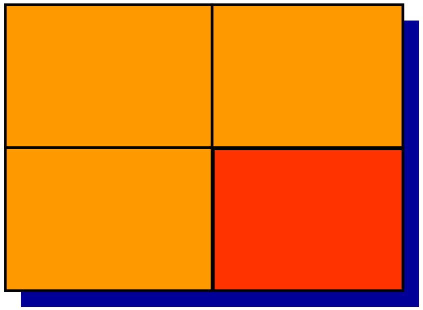
- □ What does it mean to commit a transaction locally? There is no guarantee that a committed transaction will be valid (it may be eliminated if "the other value" wins).

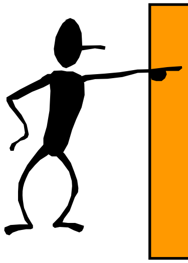
Reconciliation

- □ Such problems can be solved using pre-arranged patterns:
  - Latest update win (newer updates preferred over old ones)
  - Site priority (preference to updates from headquarters)
  - Largest value (the larger transaction is preferred)
- □ or using ad-hoc decision making procedures:
  - identify the changes and try to combine them
  - analyze the transactions and eliminate the non-important ones
  - implement your own priority schemas

Asynchronous - update everywhere

- □ Advantages:
  - No centralized coordination
  - Shortest response times
- □ Disadvantages:
  - Inconsistencies
  - Updates can be lost (reconciliation)





Performance is excellent (same as no replication). High fault tolerance. No data consistency. Reconciliation is a tough problem (to be solved almost manually).

Summary - II

- □ We have seen the different technical issues involved with each replication strategy
- □ Each replication strategy has well defined problems (deadlocks, reconciliation, message overhead, consistency) related to the way the replication protocols work
- □ The trade-off between correctness (data consistency) and performance (throughput and response time) is now clear
- □ The next step is to see how these ideas are implemented in practice

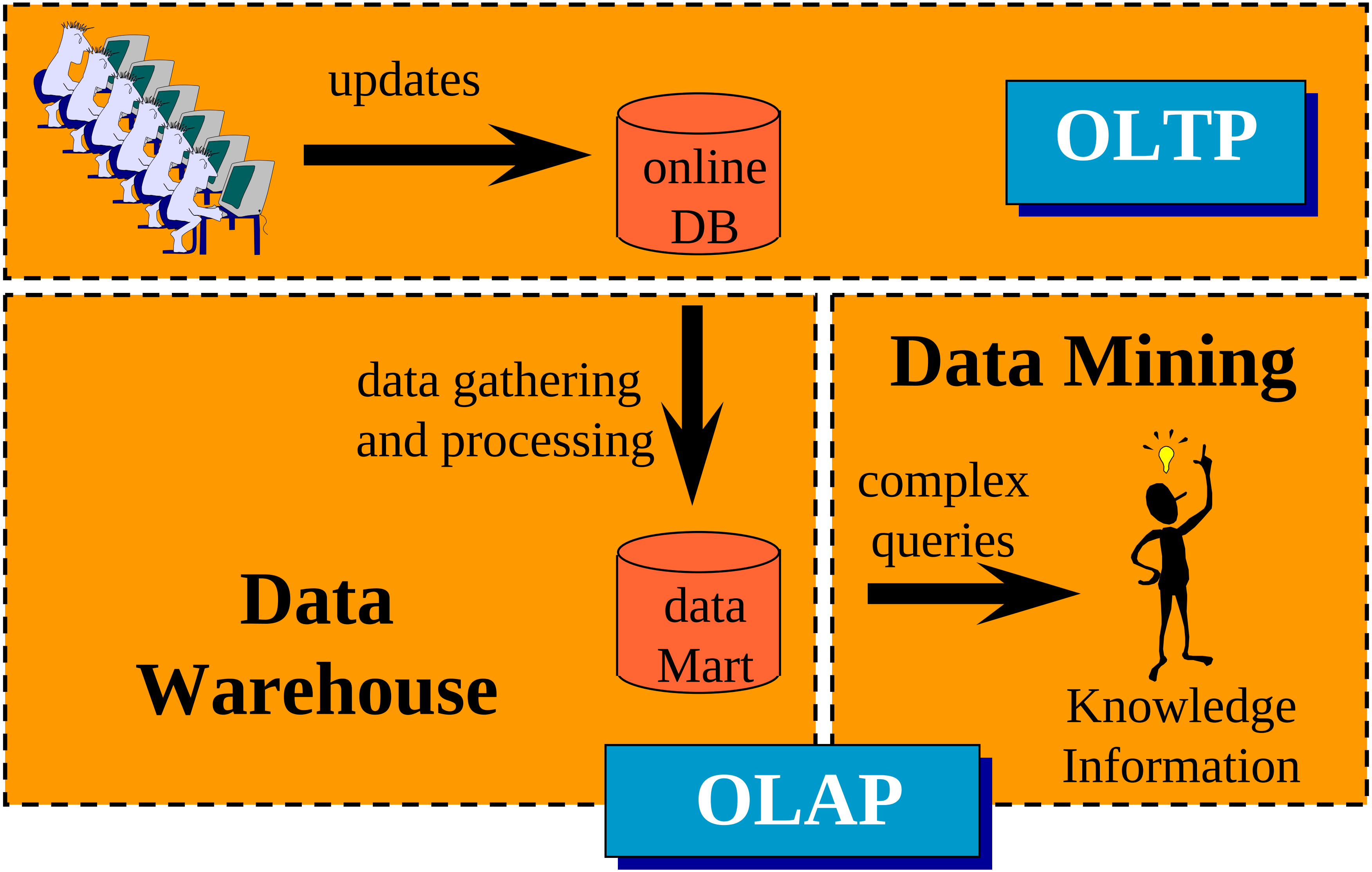
Replication in Practice

- □ Replication scenarios
- □ On Line Transaction Processing (OLTP)
- □ On Line Analytical Processing (OLAP)
- □ Replication in Sybase
- □ Replication in IBM
- □ Replication in Oracle
- □ Replication in Lotus Notes

Replication Scenarios

- □ In practice, replication is used in many different scenarios. Each one has its own demands. A commercial system has to be flexible enough to implement several of these scenarios, otherwise it would not be commercially viable.
- □ Database systems, however, are very big systems and evolve very slowly. Most were not designed with replication in mind. Commercial solutions are determined by the existing architecture, not necessarily by a sound replication strategy. Replication is fairly new in commercial databases!
- □ The focus on OLTP and OLAP determines the replication strategy in many products.
- □ From a practical standpoint, the trade-off between correctness and performance seems to have been resolved in favor of performance.
- □ It is important to understand how each system works in order to determine whether the system will ultimately scale, perform well, require frequent manual intervention ...

OLTP vs. OLAP



updates

online DB

OLTP

data gathering and processing

Data Warehouse

data Mart

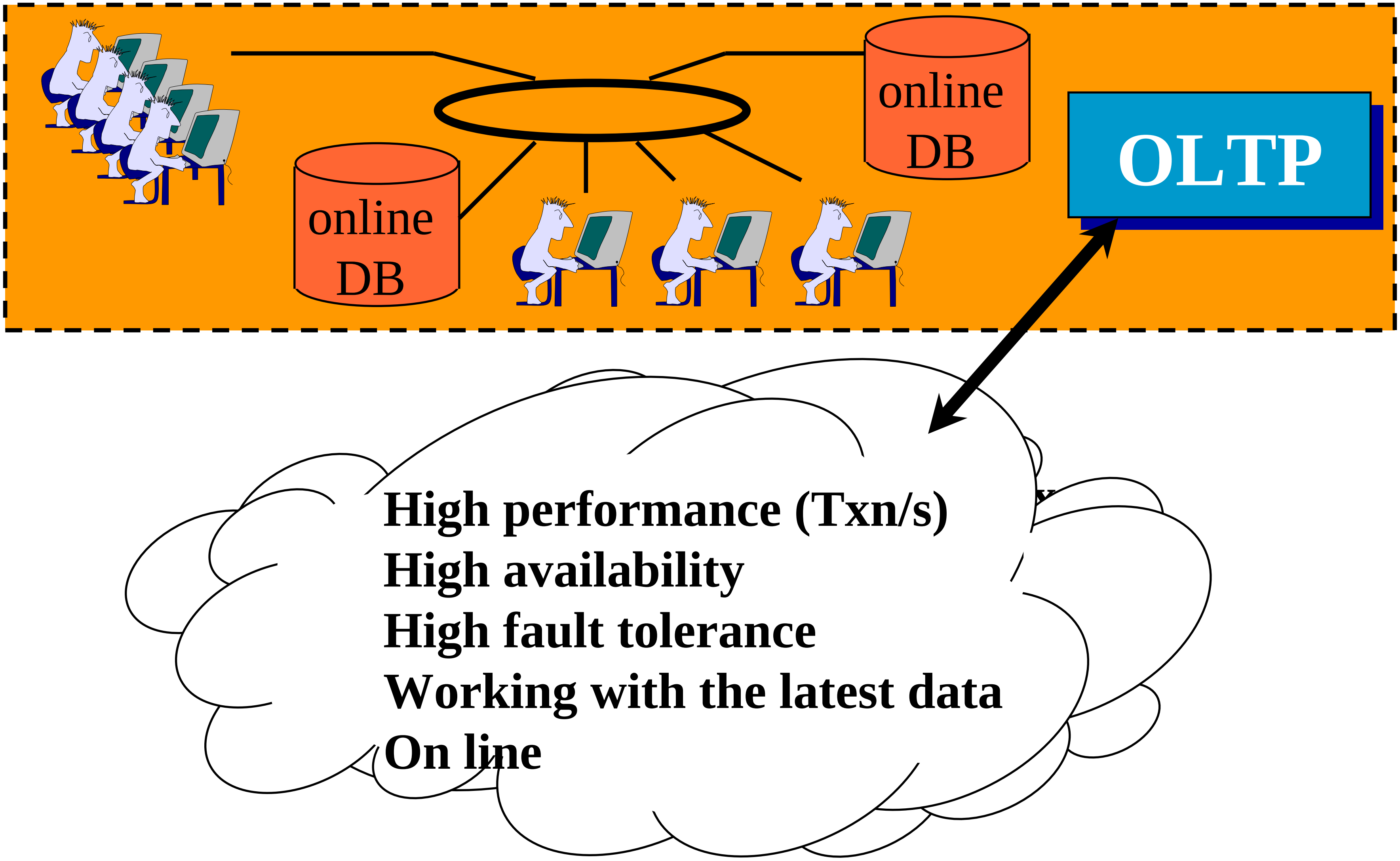
Data Mining

complex queries

Knowledge Information

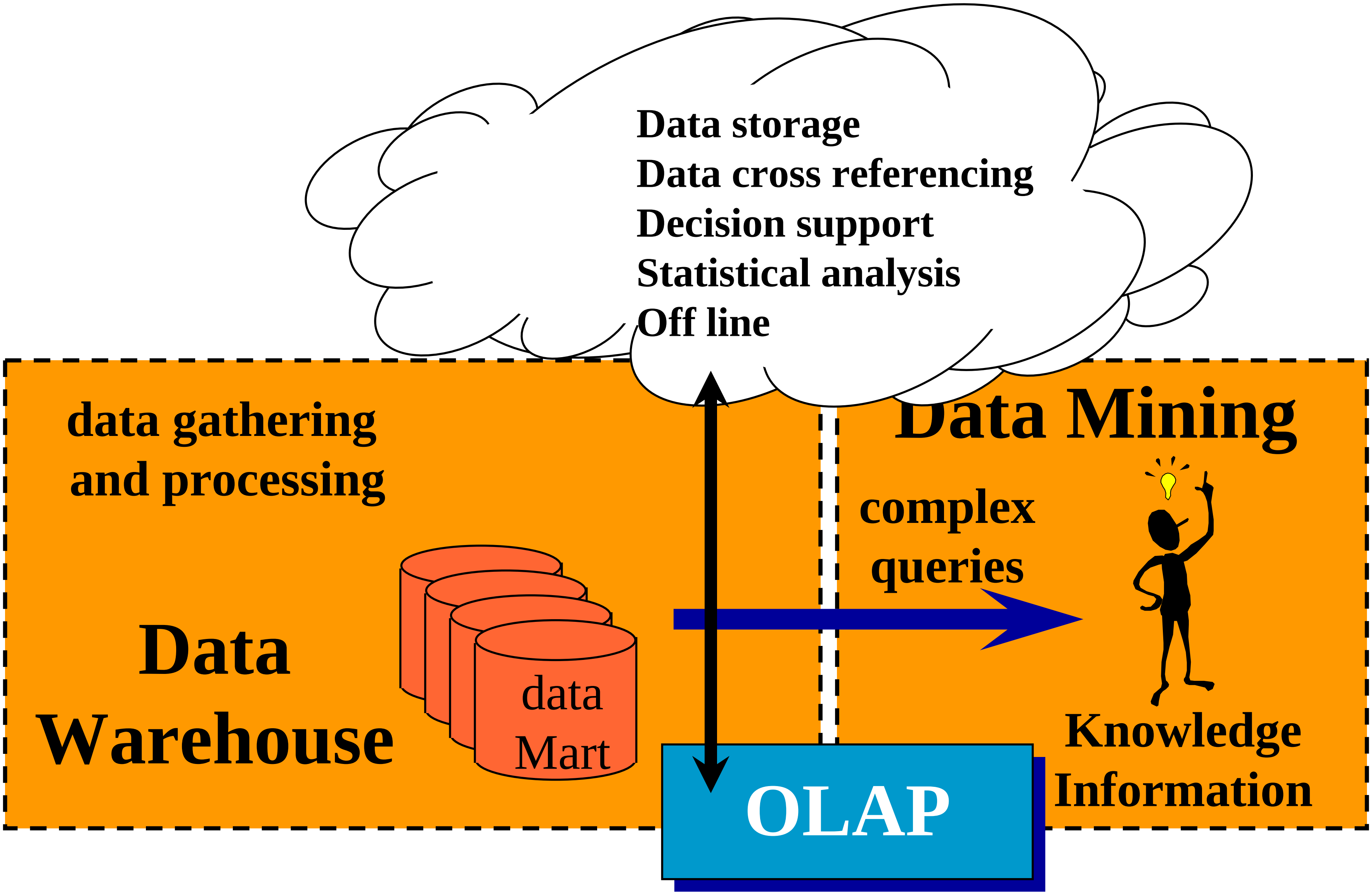
OLAP

OLTP



High performance (Txn/s)
High availability
High fault tolerance
Working with the latest data
On line

OLAP



Data storage
Data cross referencing
Decision support
Statistical analysis
Off line

data gathering and processing

Data Warehouse

data Mart

Data Mining

complex queries

Knowledge Information

OLAP

Commercial replication

When evaluating a commercial replication strategy, keep in mind:

□ The customer base (who is going to use it?).
□ The underlying database (what can the system do?).
□ What competitors are doing (market pressure).
□ There is no such a thing as a "better approach".
□ The complexity of the problem.

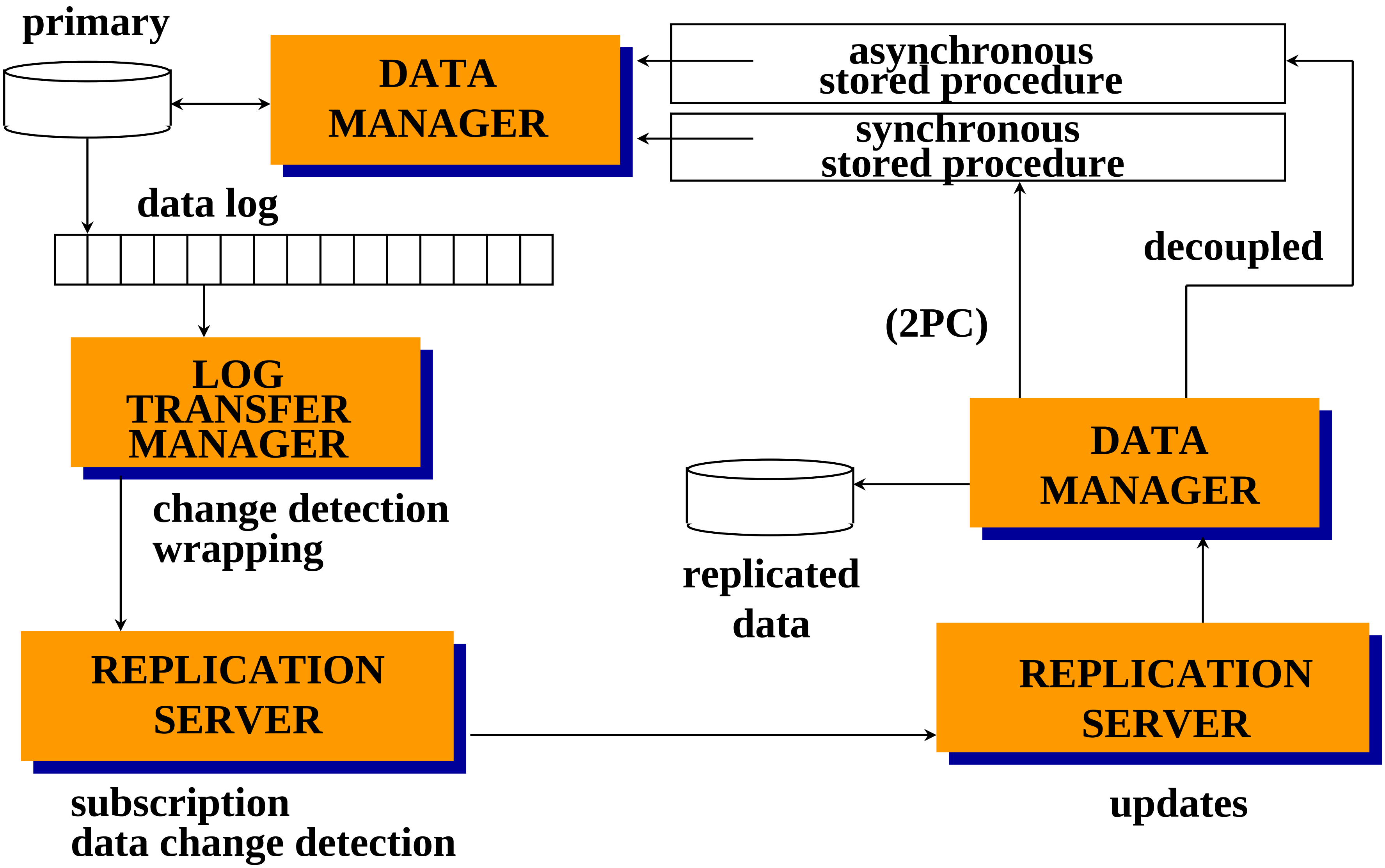
Replication will keep evolving in the future, current systems may change radically.

Sybase Replication Server

(http://www.sybase.com)

- □ Goal of replication: Avoid server bottlenecks by moving data to the clients. To maintain performance, asynchronous replication is used (changes are propagated only after the transaction commits). The changes are propagated on a transaction basis (get the replicas up-to-date as quickly as possible). Capture of changes is done "off-line", using the log to minimize the impact on the running server.
- □ Applications: OLTP, client/server architectures, distributed database environments.

Sybase Replication Architecture



Sybase Replication (basics)

- □ Loose consistency (= asynchronous). Primary copy.
- □ PUSH model: replication takes place by "subscription". A site subscribes to copies of data. Changes are propagated from the primary as soon as they occur. The goal is to minimize the time the copies are not consistent but still within an asynchronous environment (updates are sent only after they are committed).
- □ Updates are taken from the log in stable storage (only committed transactions).
- □ Remote sites update using special stored procedures (synchronous or a synchronous).
- □ Persistent queues are used to store changes in case of disconnection.
- □ The Log Transfer Manager monitors the log of Sybase SQL Server and notifies any changes to the replication server. It acts as a light weight process that examines the log to detect committed transactions (a wrapper). It is possible to write your own Log Transfer Manager for other systems. Usually runs in the same system as the source database. When a transaction is detected, its log records are sent to the:
- □ The Replication Server usually runs on a different system than the database to minimize the load. It takes updates, looks who is subscribed to them and send them to the corresponding replication servers at the remote site. Upon receiving these changes, a replication server applies them at the remote site.

Sybase Replication (updates)

Primary copy. All updates must be done at the primary using either :

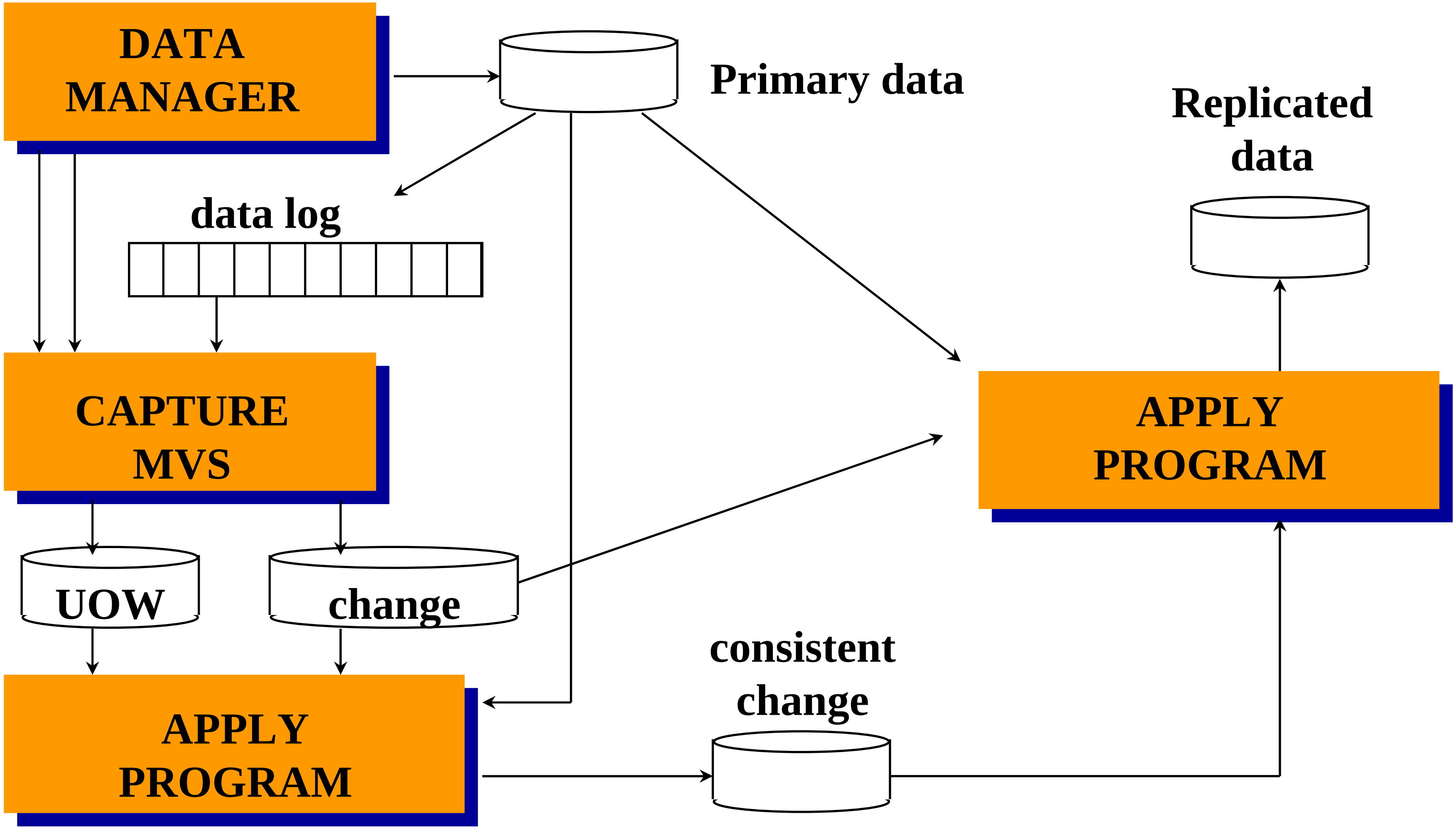
- □ Synchronous stored procedures, which reside at the primary and are invoked (RPC) by any site who wants to update. 2 Phase Commit is used.
- □ Stored procedures for asynchronous transactions: invoked locally, but sent asynchronously to the primary for execution. If the transaction fails manual intervention is required to fix the problem.
- □ It is possible to fragment a table and make different sites the primary copy for each fragment.
- □ It is possible to subscribe to selections of tables using WHERE clauses.

IBM Data Propagator

(http://www.ibm.com/)

- □ Goal: Replication is seen as part of the "Information Warehousing" strategy. The goal is to provide complex views of the data for decision-support. The source systems are usually highly tuned, the replication system is designed to interfere as less as possible with them: replication is asynchronous and there are no explicit mechanisms for updating.
- □ Applications: OLAP, decision-support, data warehousing, data mining.

IBM Replication (architecture)



IBM Data Propagator (basics)

- □ Asynchronous replication.
- □ No explicit update support (primary copy, if anything).
- □ PULL MODEL: (smallest interval 1 minute) the replicated data is maintained by querying either the primary data, the change table, the consistent change table, or any combination of the three. The goal is to support sophisticated views of the data (data warehousing). Pull model means replication is driven by the recipient of the replica. The replica must "ask" for updates to keep up-to-date.
- □ Updates are taken from the main memory buffer containing log entries (both committed and uncommitted entries; this is an adjustable parameter).
- □ Updates are sent to the primary (updates converted into inserts if tuple has been deleted, inserts converted into updates if tuple already exists, as in Sybase). The system is geared towards decision support, replication consistency is not a key issue.
- □ Sophisticated data replication is possible (base aggregation, change aggregation, time slices ...)
- □ Sophisticated optimizations for data propagation (from where to get the data).
- □ Sophisticated views of the data (aggregation, time slicing).
- □ Capture/MVS is a separate address space monitor, to minimize interference it captures log records from the log buffer area

IBM Data Propagator

There are two key components in the architecture:

- □ Capture: analyzes raw log information from the buffer area (to avoid I/O). It reconstructs the logical log records and creates a "change table" and a "transaction table" (a dump of all database activity).
- □ Apply Program: takes information from the database, the change table and the transaction table to built "consistent change table" to allow consistent retrieval and time slicing. It works by "refreshing" data (copies the entire data source) or "updating" (copies changes only). It allows very useful optimizations (get the data from the database directly, reconstruct, etc.).

The emphasis is on extracting information:

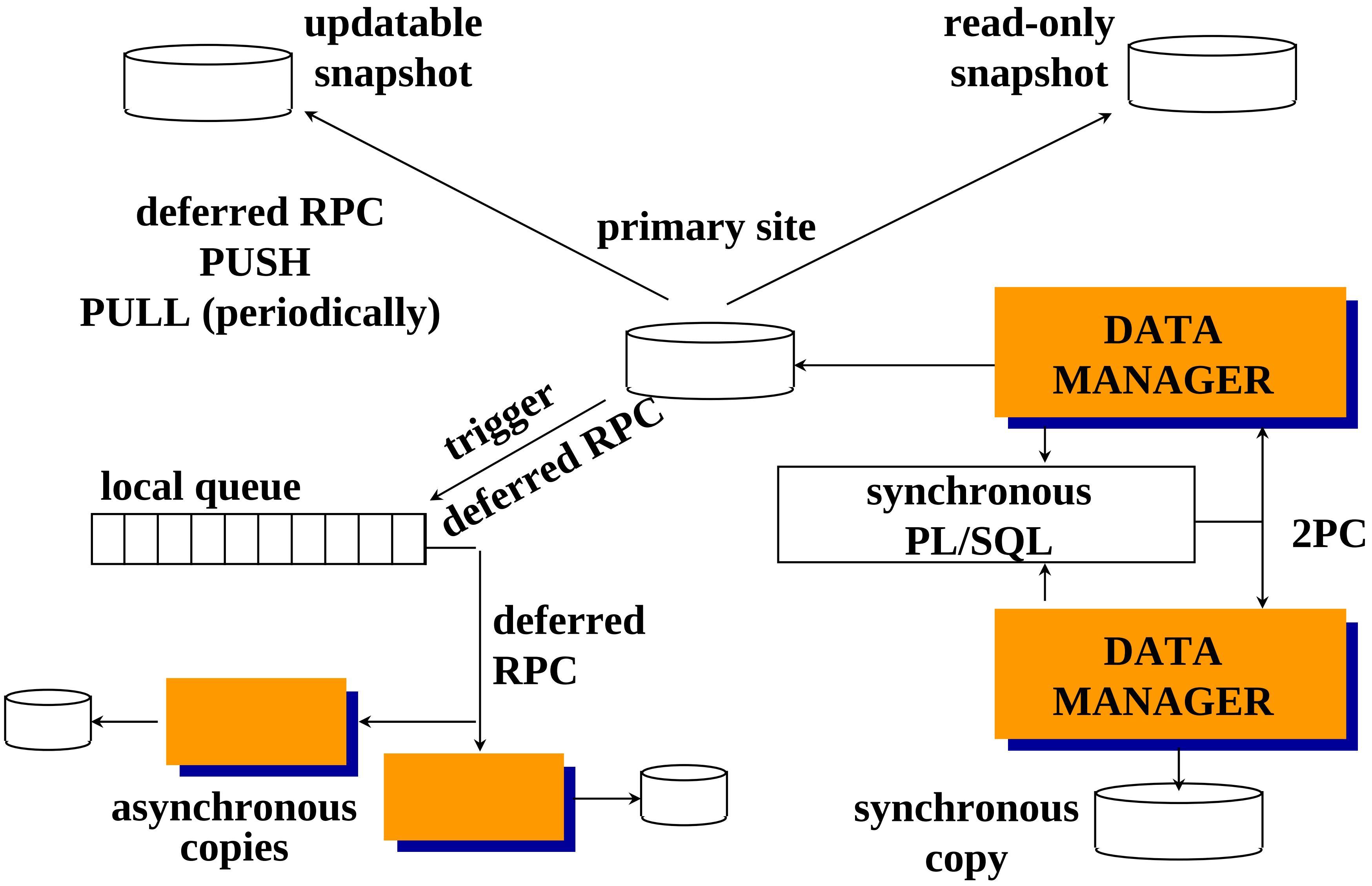
- □ Data Propagator/2 is used to subscribe and request data.
- □ It is possible to ask for the state of data at a given time (time slicing or snapshots).
- □ It is possible to ask for changes:
  - how many customers have been added?
  - how many customers have been removed?
  - how many customers were between 20 and 30 years old?
- □ This is not the conventional idea of replication!

Oracle Symmetric Replication

(http://www.oracle.com)

- □ Goals: Flexibility. It tries to provide a platform that can be tailored to as many applications as possible. It provides several approaches to replication and the user must select the most appropriate to the application. There is no such a thing as a "bad approach", so all of them must be supported (or as many as possible)
- □ Applications: intended for a wide range of applications.

Oracle Replication (architecture)



Oracle Replication

- □ "DO-IT-YOURSELF" model supporting almost any kind of replication (push model, pull model), Dynamic Ownership (the site designated as the primary can change over time), and Shared Ownership (update anywhere, asynchronously).
- □ One of the earliest implementations: Snapshot. This was a copy of the database. Refreshing was done by getting a new copy.
- □ Symmetric replication: changes are forwarded at time intervals (push) or on demand (pull).
- □ Asynchronous replication is the default but synchronous is also possible.
- □ Primary copy (static / dynamic) or update everywhere.
- □ Readable Snapshots: A copy of the database. Refresh is performed by examining the log records of all operations performed, determining the changes and applying them to the snapshot. The snapshot cannot be modified but they are periodically refreshed (complete/fast refreshes)
- □ Writable Snapshots: fast-refreshable table snapshots but the copy can be updated (if changes are sent to the master copy, it becomes a form of asynchronous - update everywhere replication).

Oracle Replication (basics)

Replication is based on these two ideas:

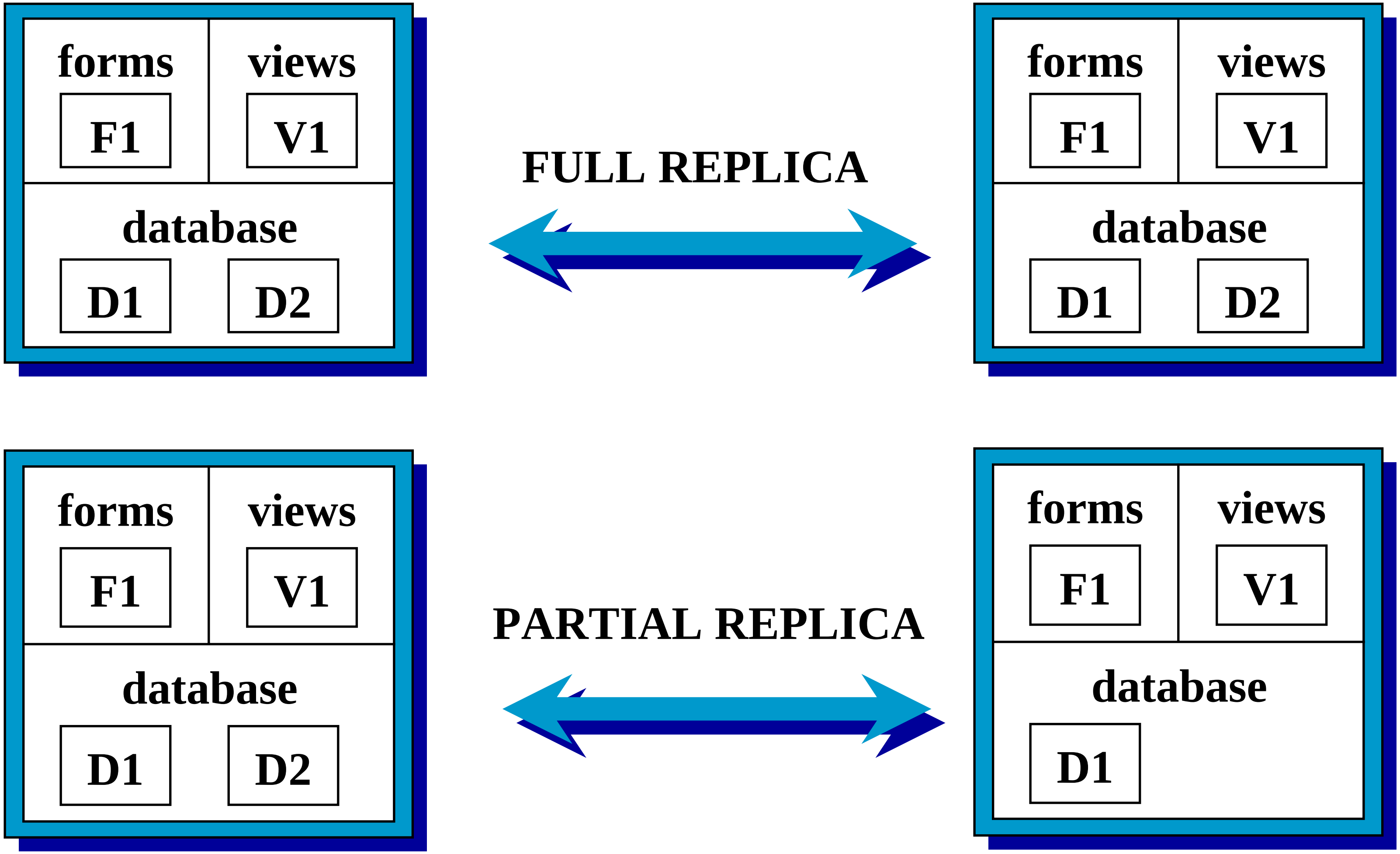
- □ Triggers: changes to a copy are captured by triggers. The trigger executes a RPC to a local queue and it inserts the changes in the queue. These changes take the form of an invocation to a stored procedure at the remote site. These triggers are "deferred" in the sense that they work asynchronously with respect to the transaction
- □ Queues: queues follow a FIFO discipline and 2PC is used to guarantee the call makes it to the queue at the remote site. At the remote site, the queue is read and the call made in the order they arrive.

- □ Dynamic ownership: It is possible to dynamically reassign the "master copy" to different sites. That is, the primary copy can move around (doing it well, it is then possible to always read and write locally)
- □ Shared ownership: (= update everywhere!). Conflicts are detected by propagating both the before and the after image of data. When a conflict is detected, there are several predefined routines that can be automatically called or the user can write and ad-hoc routine to resolve the conflict
- □ Synchronous, update everywhere: using the sync -update everywhere protocol previosuly discussed

Replication in Lotus Notes (Domino)

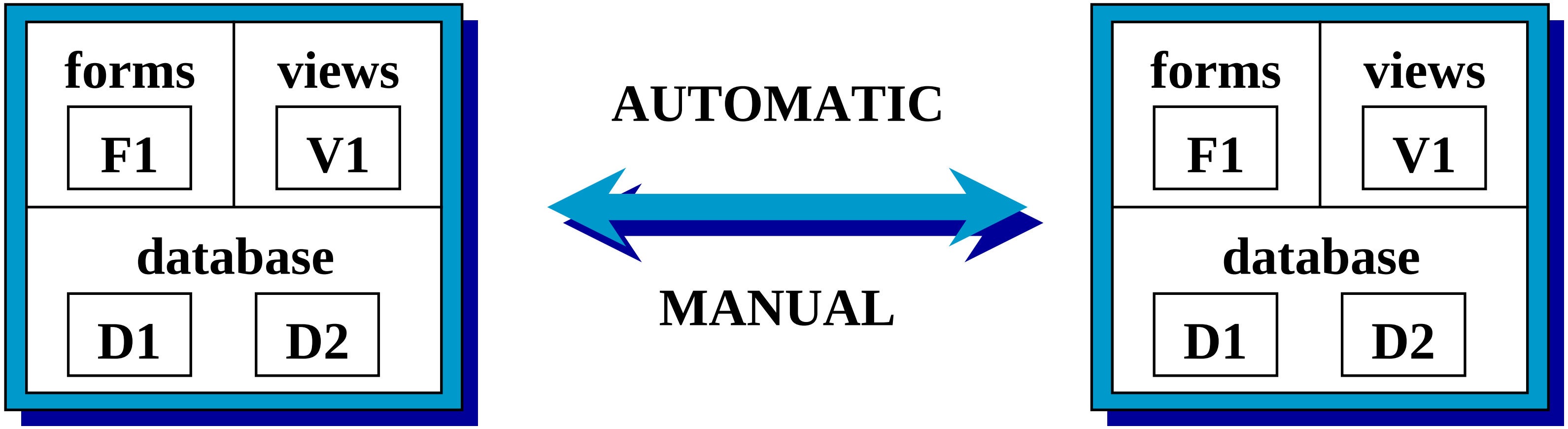
- □ Lotus Notes implements asynchronous (lazy), update every-where replication in an epidemic environment.
- □ Lotus Notes distinguishes between a replica and a copy (a snapshot). All replicas have the same id. Each copy has its own id.
- □ Lotus allows to specify what to replicate (in addition to replica stubs and field level replication) to minimize overhead.
- □ Replication conflicts are detected and some attempt is made at reconciliation (user intervention is usually required).
- □ Lotus Notes is a cooperative environment, the goal is data distribution and sharing. Consistency is largely user defined and not enforced by the system.

Replication in Lotus Notes



Replication in Lotus Notes

Notes also allows to specify when to replicate ...



forms

F1

views

V1

database

D1

D2

AUTOMATIC

MANUAL

forms

F1

views

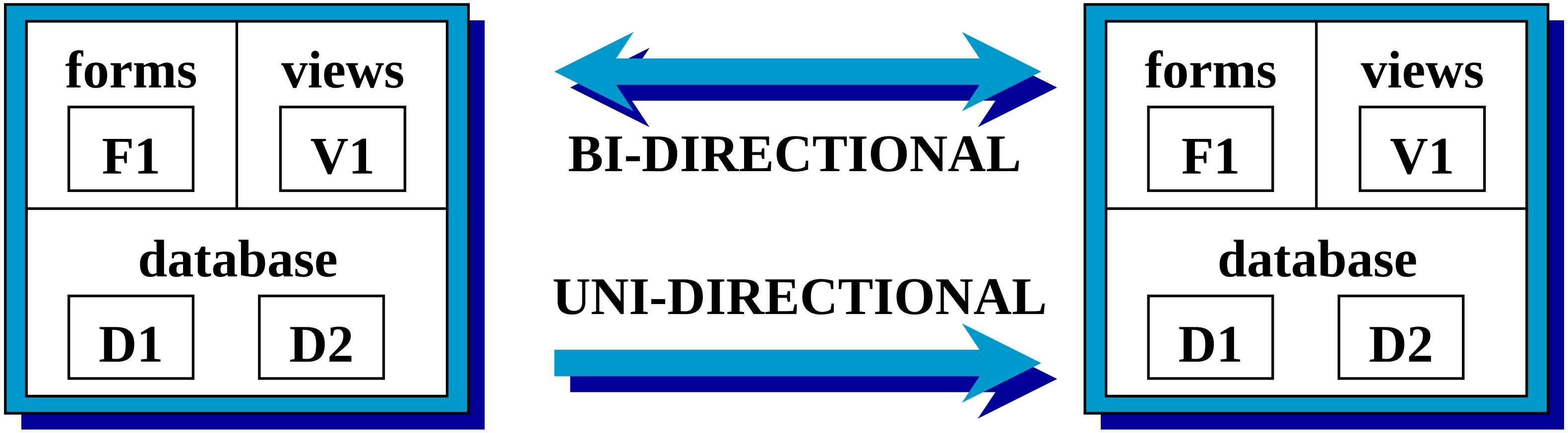
V1

database

D1

D2

.. and in which direction to replicate:



forms

F1

views

V1

database

D1

D2

BI-DIRECTIONAL

UNI-DIRECTIONAL

forms

F1

views

V1

database

D1

D2

Token Passing Protocol

Replication is used in many applications other than databases. For these applications, there is a large number of protocols and algorithms that can be used to guarantee "correctness":

□ The token based protocol is used as an example of replication in distributed systems to illustrate the problems of fault-tolerance and starvation.

Distributed Mutual Exclusion

- □ The original protocol was proposed for distributed mutual exclusion. It can be used, however, to maintain replicated data and to implement the notion of dynamic ownership (Oracle replication).

In here, it will be used for the following:

- □ Asynchronous, master copy (dynamic ownership)
- □ The protocol will be used to locate the master copy
- □ Requirements:
  - there is only one master copy at all times
  - deadlock free
  - fault-tolerant
  - starvation free

Token Passing (model)

Working assumptions

- □ Communications are by message passing
- □ Sites are fail-stop or may fail to send and receive messages
- □ Failed sites eventually recover (failure detection by time-out)
- □ Network partitions may occur
- □ No duplicate messages and FIFO delivery
- □ Causality enforced by logical clocks (Lamport)

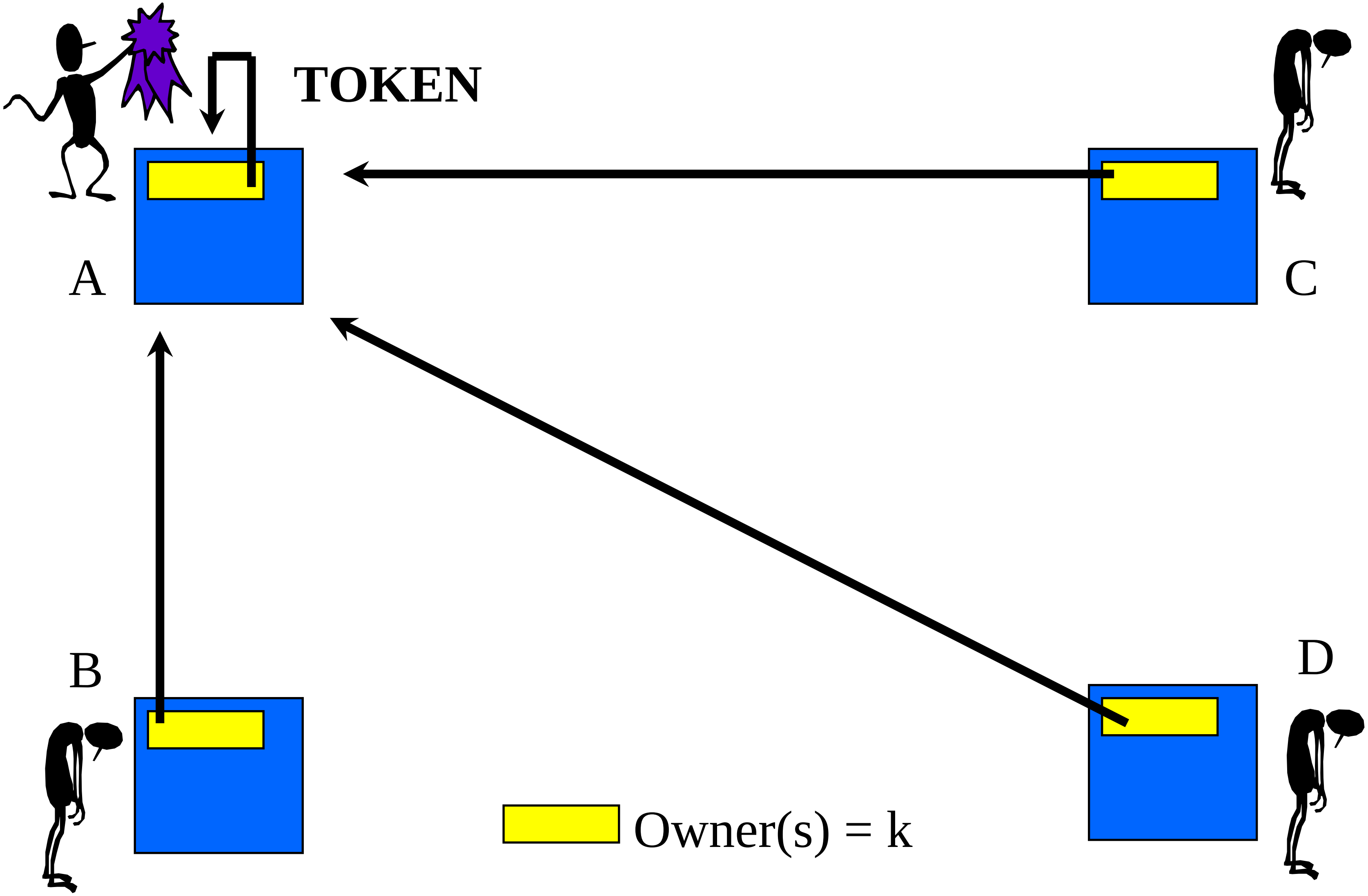
Happen Before Relation →
(1) events in a process are ordered
(2) sending(m) → receiving(m)
(3) if a → b and b → c,then a → c

Clock condition
(1) each event has a timestamp
(2) succesive events have increasing timestamps
(3) receiving(m) has a higher timestamp than sending(m)

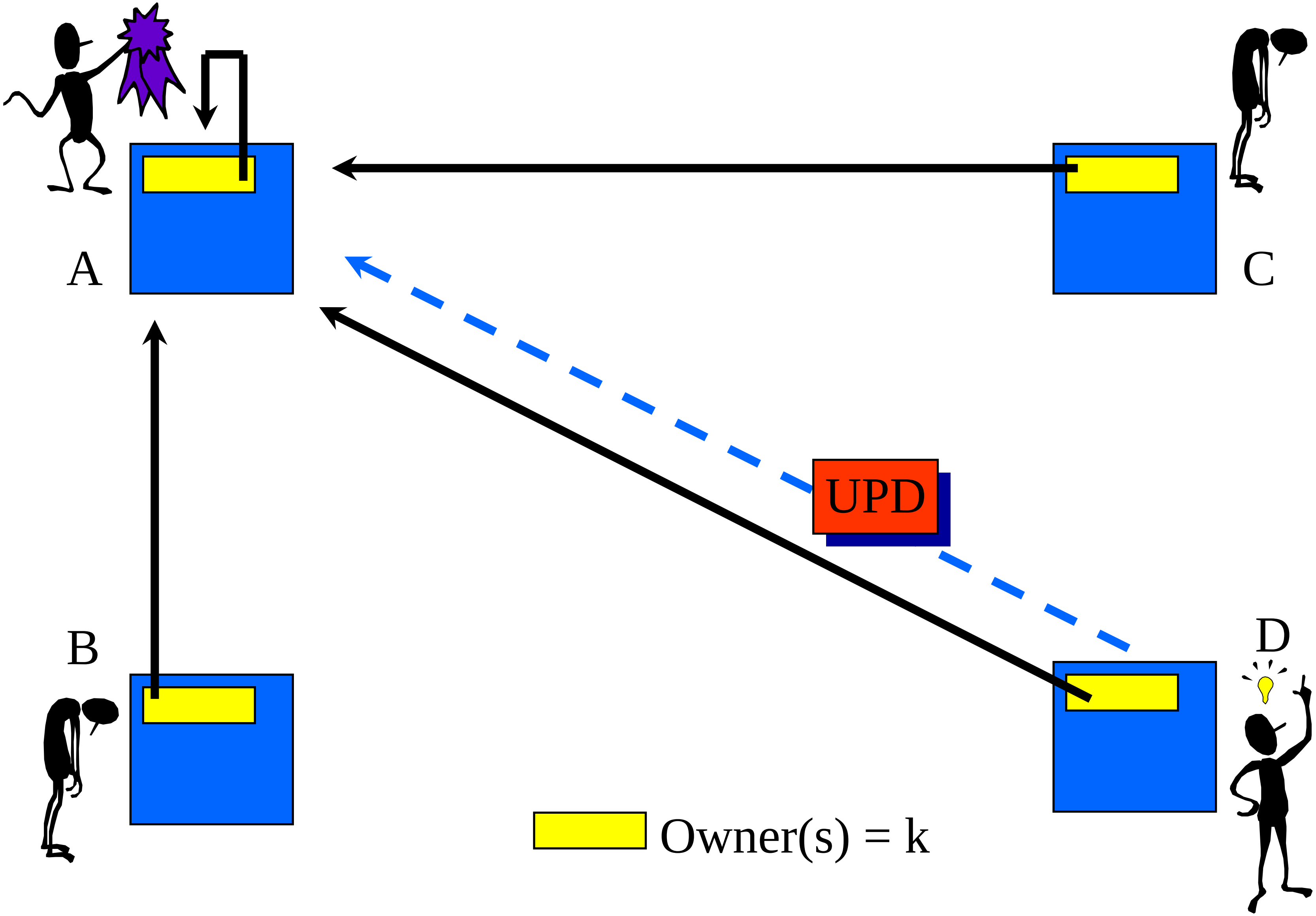
Basic Protocol (no failures)

- □ Assume no communication or site failures
- □ A node with the token is the master copy
- □ Each site, s, has a pointer, Owner(s), indicating where that site believes the master copy is located
- □ The master copy updates locally
- □ Other sites sent their updates following the pointer
- □ When the master copy reassigns the token (the master copy moves to another site), the ex-master copy readjusts its pointer so it points towards the new master copy
- □ For correctness reasons, assume the master copy is never reassigned while updates are taking place.

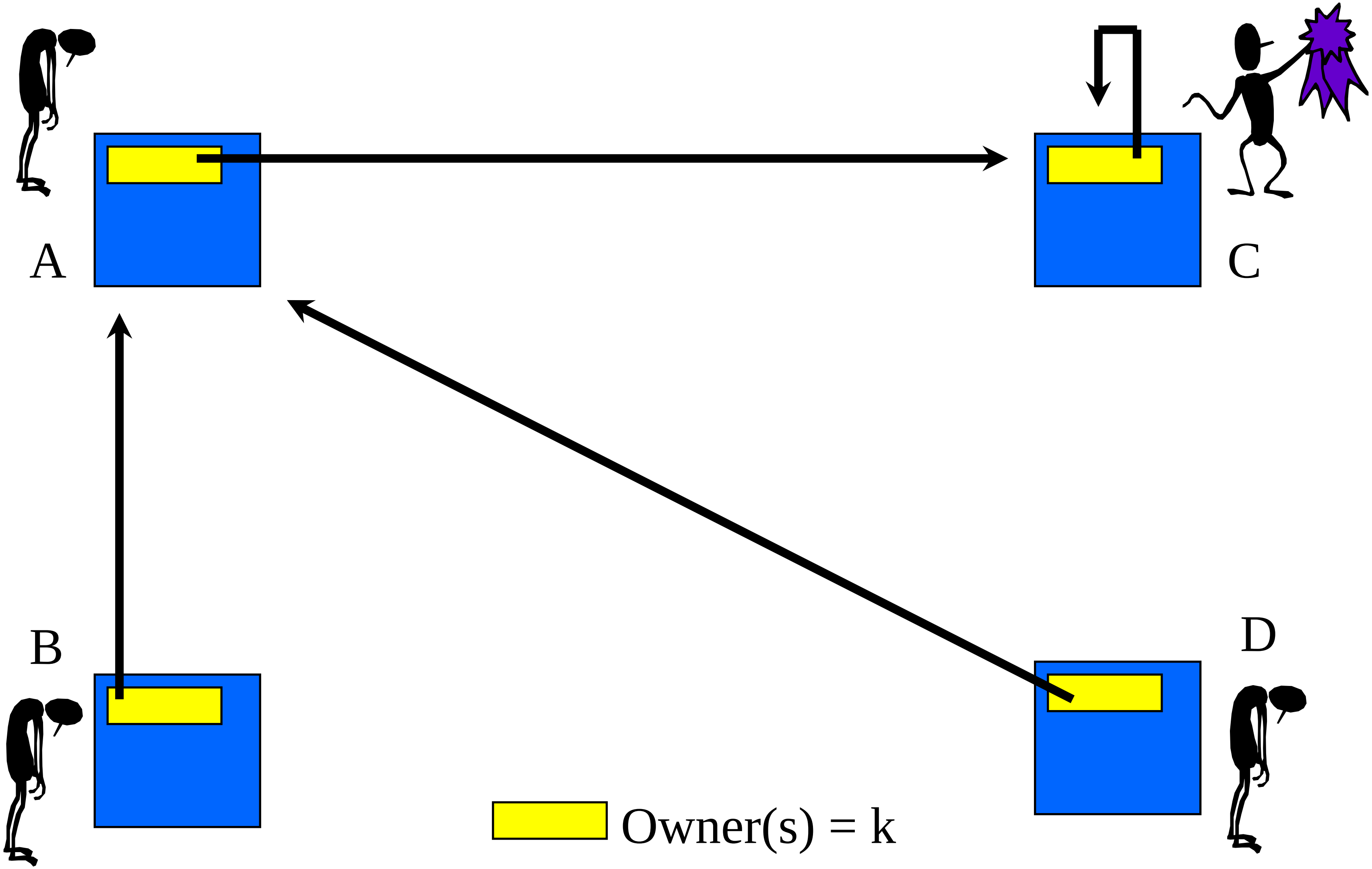
Basic Protocol (owner)



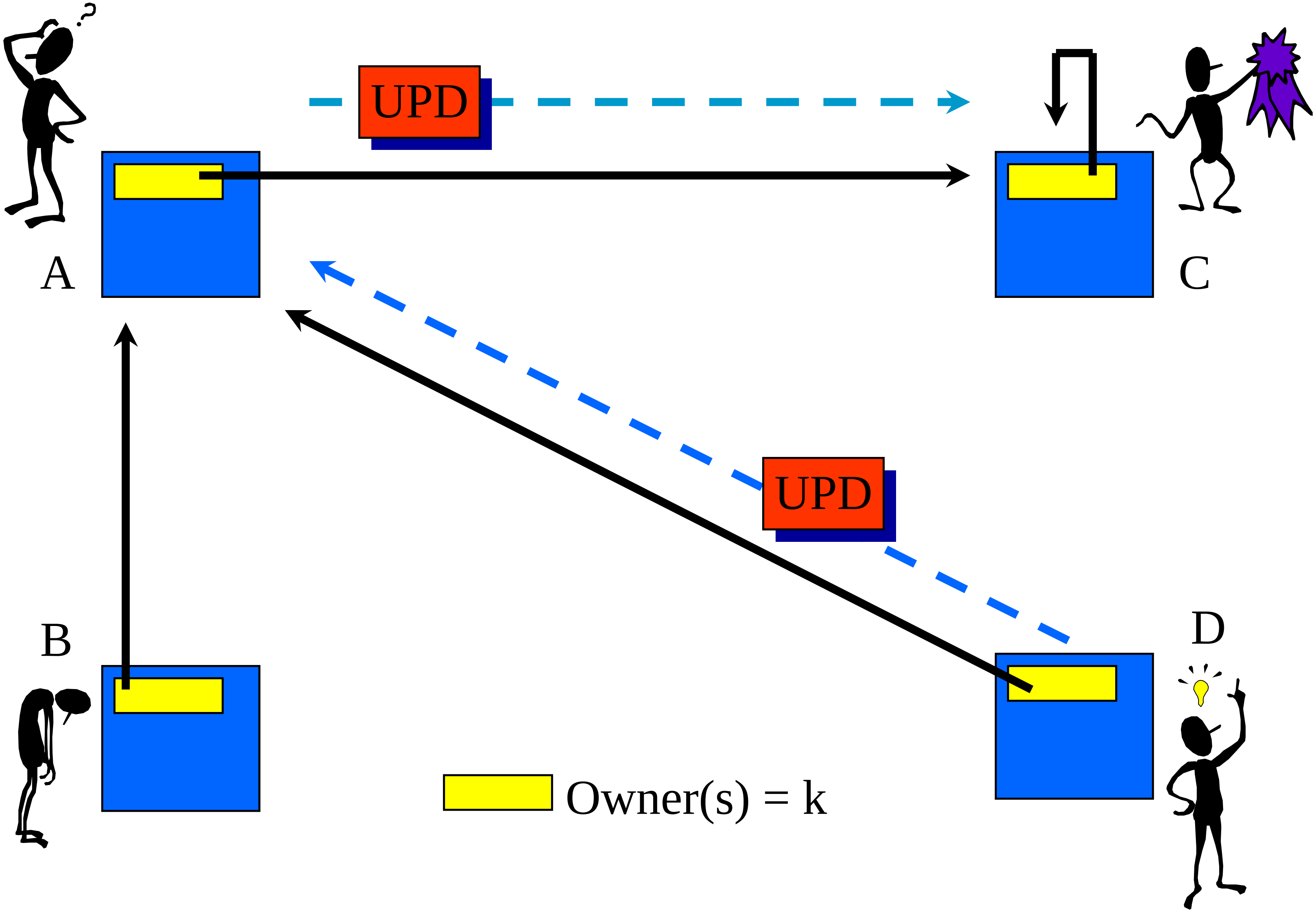
Basic Protocol (update)



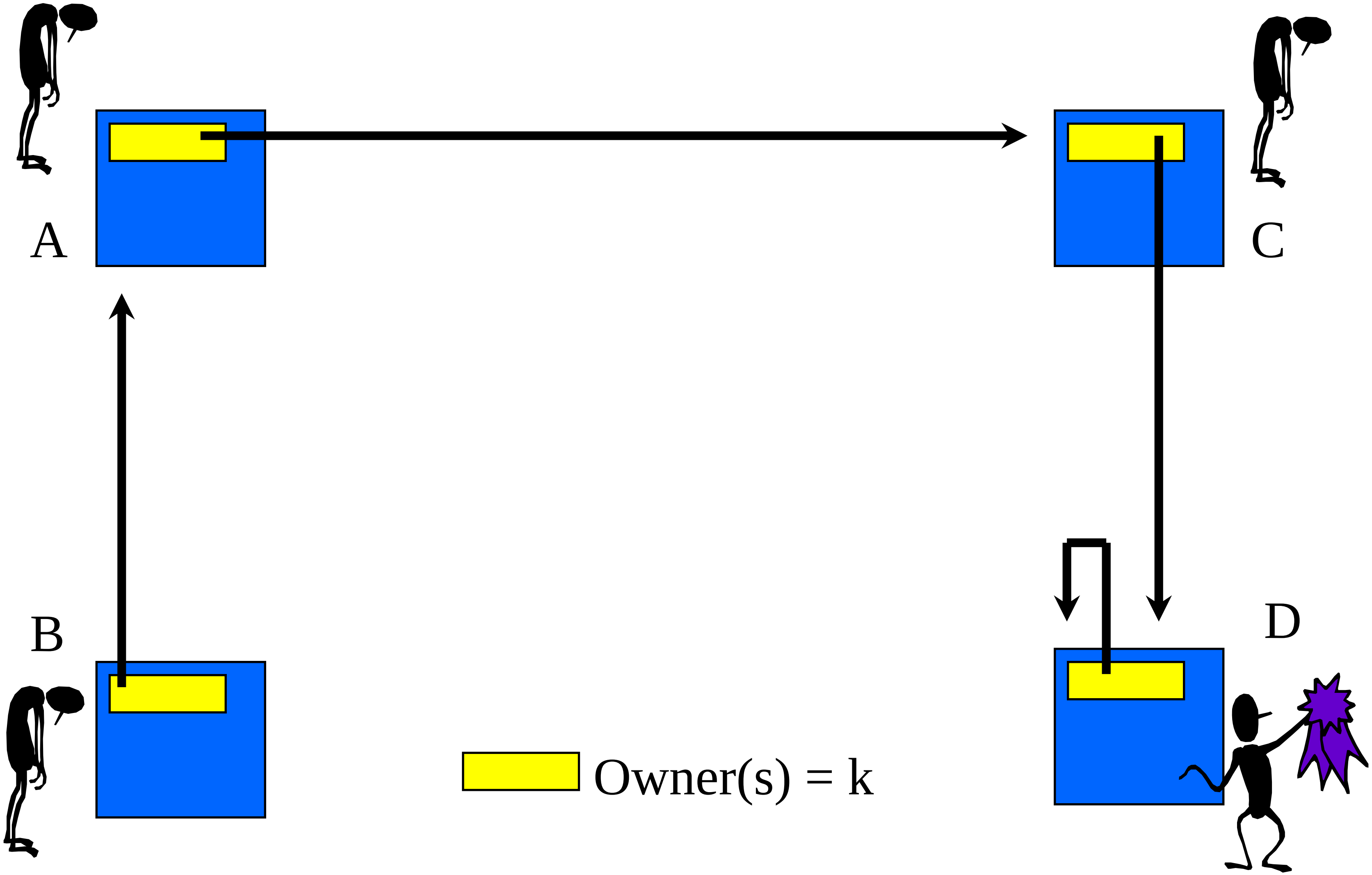
Basic Protocol (token change)



Basic Protocol (update)



Basic Protocol (token change)



A

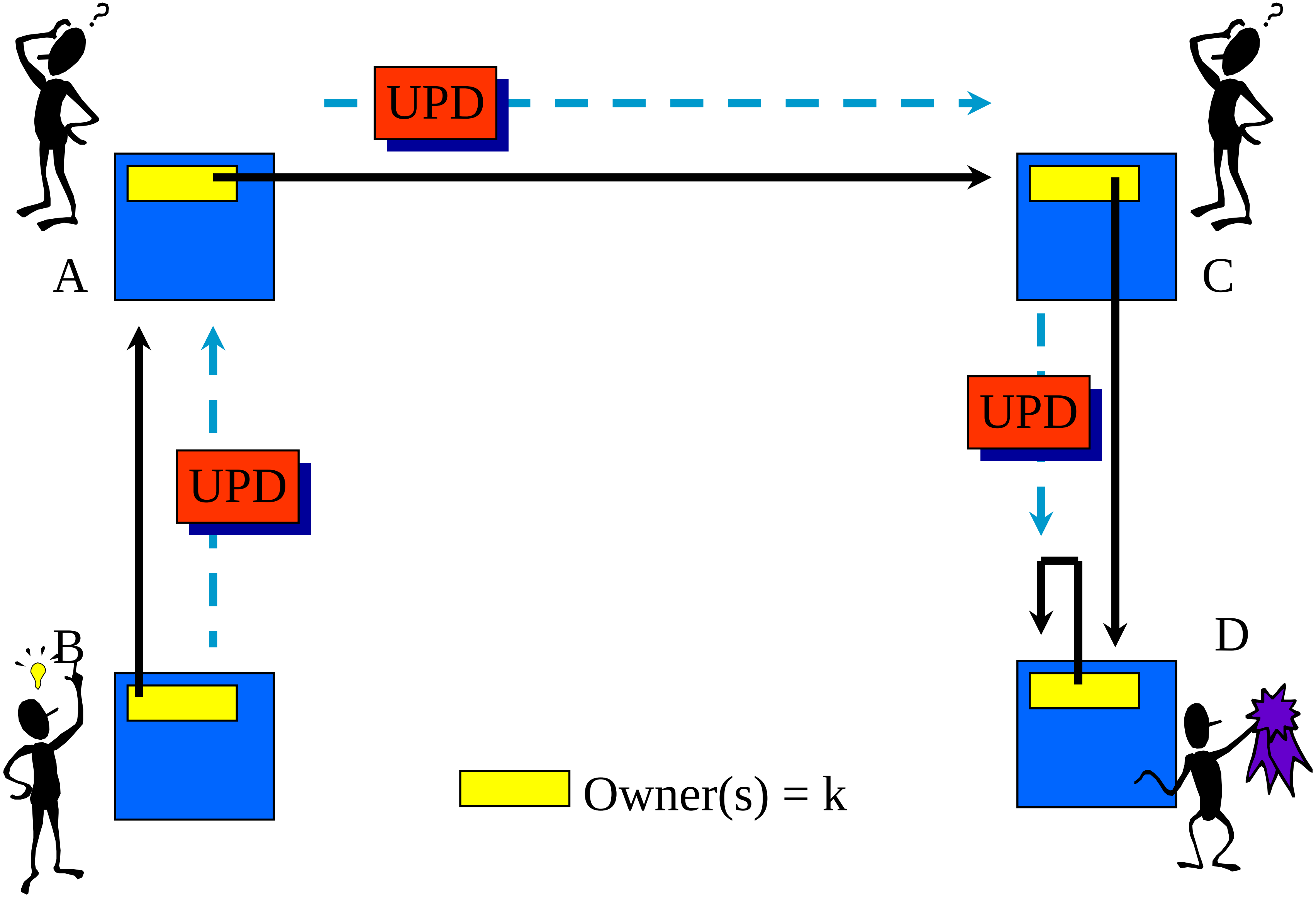
C

B

D

🟨 Owner(s) = k

Basic Protocol (update)



Basic Protocol (algorithms)

Requesting the master copy (s)

```
IF Owner(s) = s THEN
    master copy already in s
ELSE
    SEND(request) to Owner(s)
    RECEIVE(Token)
    Owner(s) = s
END (*IF*)
```

Receiving a request (q)

```
Receive (request(s))
IF Owner(q) = q THEN
    Owner(q) = s
    SEND(Token) to s
ELSE
    SEND(request(s)) to Owner(q)
END (*IF*)
```

Failures

If communication failures occur, the token may disappear while in transit (message is lost).

□ First, the loss of the token must be detected

□ Second, the token must be regenerated

□ Third, after the regeneration, there must be only one token in the system (only one master copy)

To do this, logical clocks are used:

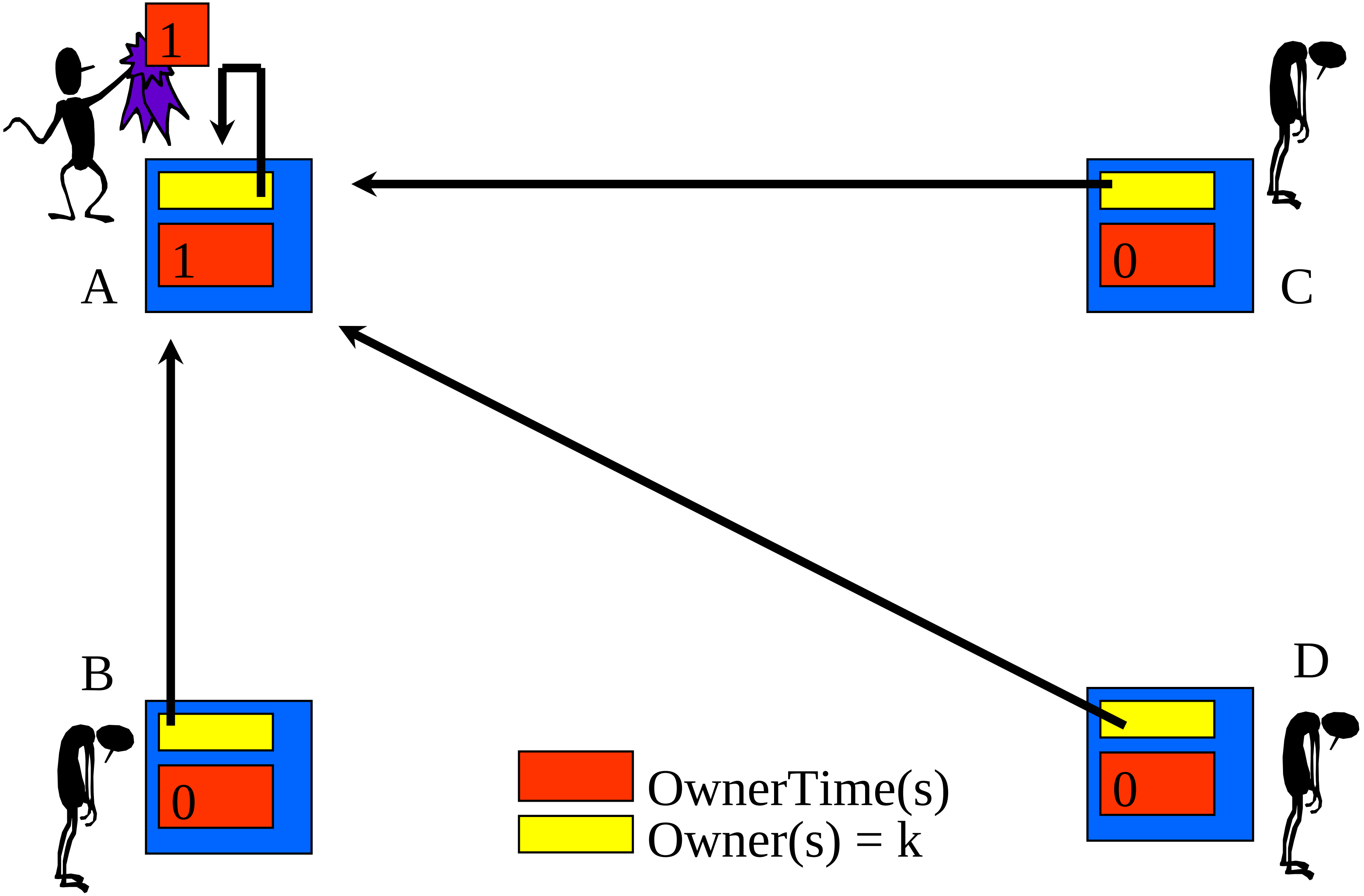
□ OwnerTime(s) is a logical clock associated with the token, it indicates when site s sent or received the token

□ TokenState(s) is the state of the shared resource (values associated with the token itself)

Token Loss Protocol

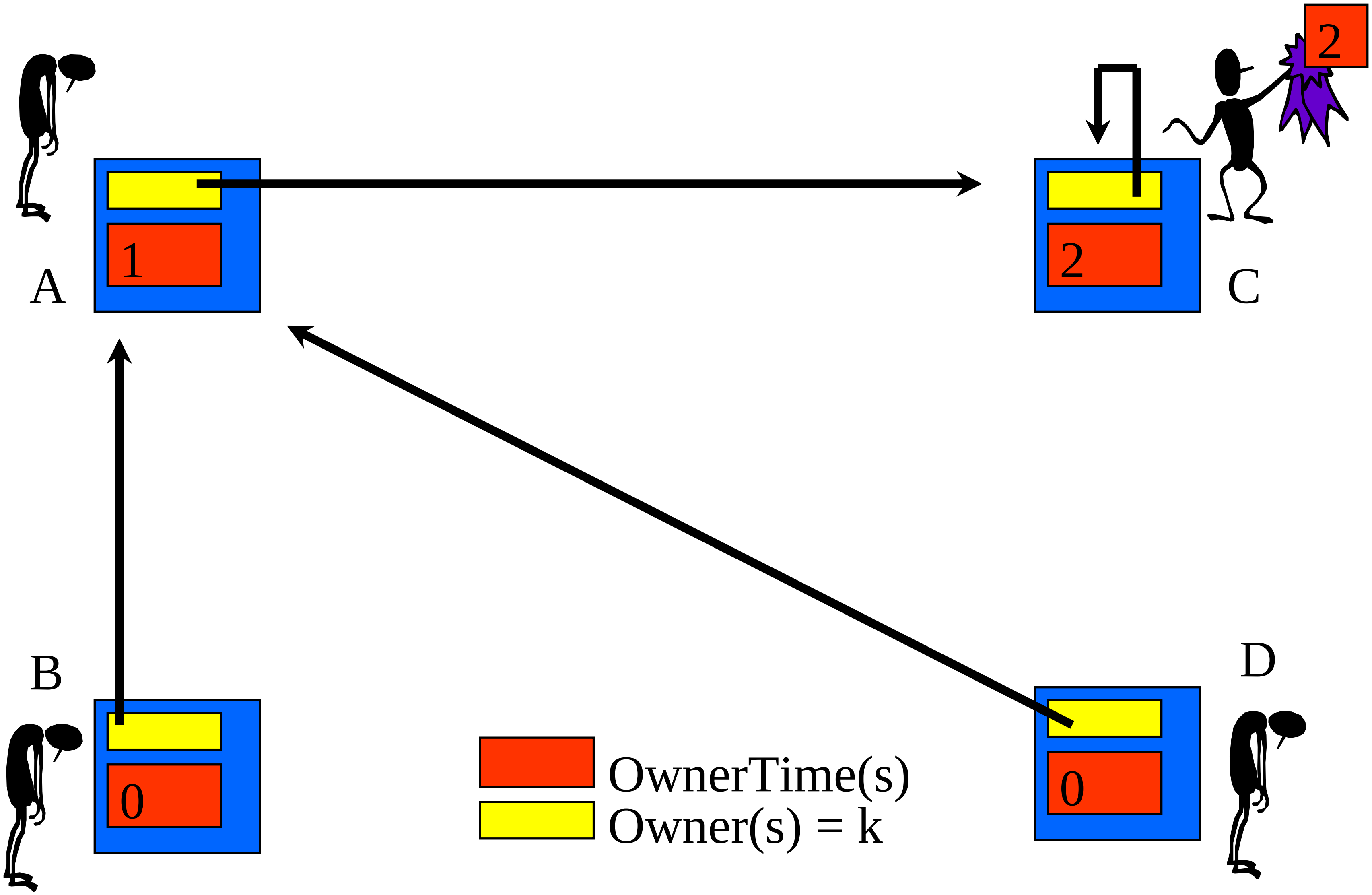
- □ Assume bounded delay (if a message does not arrive after time t, it has been lost). Sites do not fail
- □ When a site sends the token, it sends along its own OwnerTime
- □ When a site receives the token, it sets its OwnerTime to a value greater than that received with the token
- □ From here, it follows that the values of the OwnerTime variables along the chain of pointers must increase
- □ If, along the chain of pointers, there is a pair of values that is not increasing, the token has been lost between these two sites and must be regenerated

Token Loss Protocol



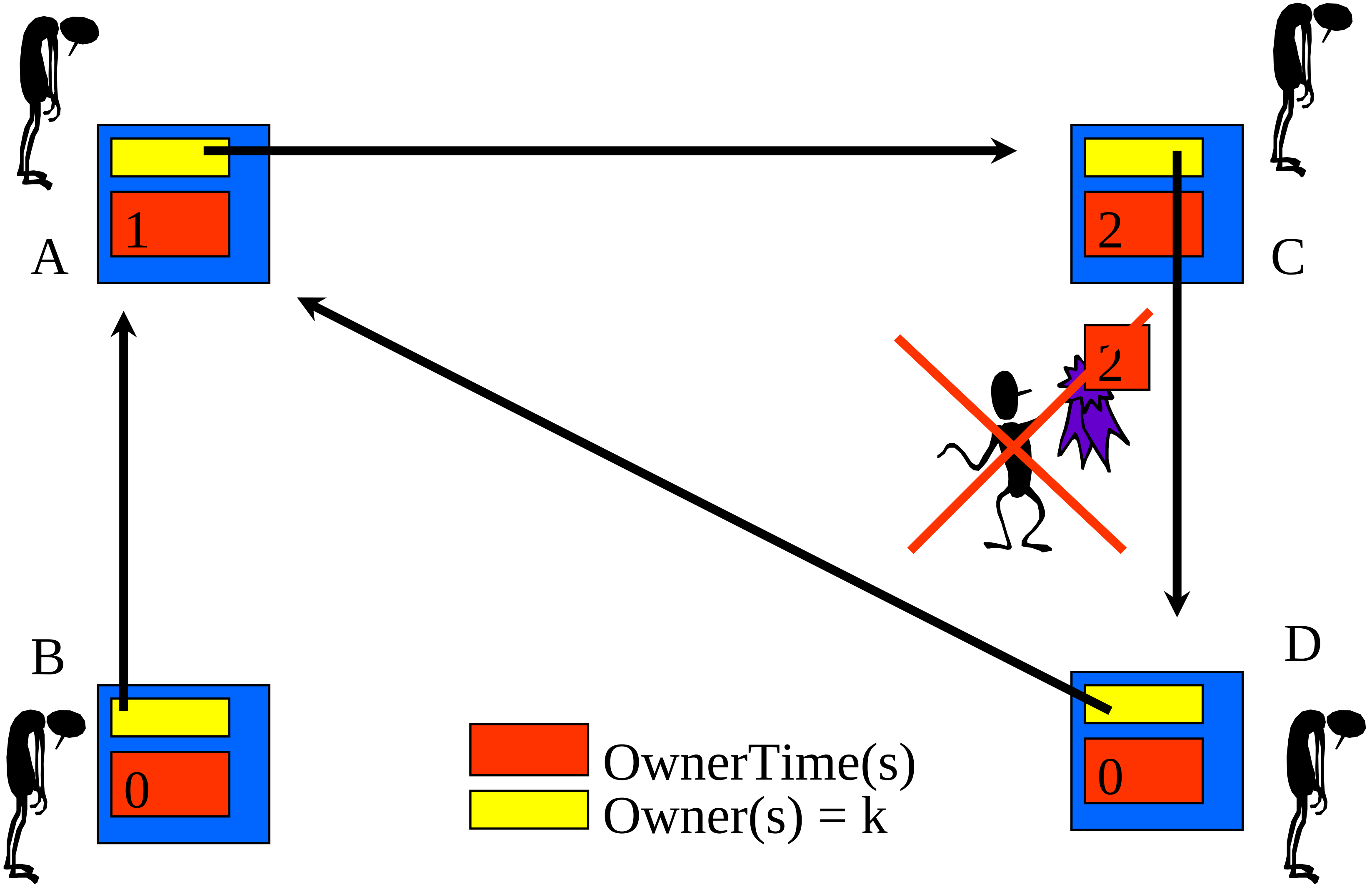
🟥 OwnerTime(s)
🟨 Owner(s) = k

Detecting Token Loss

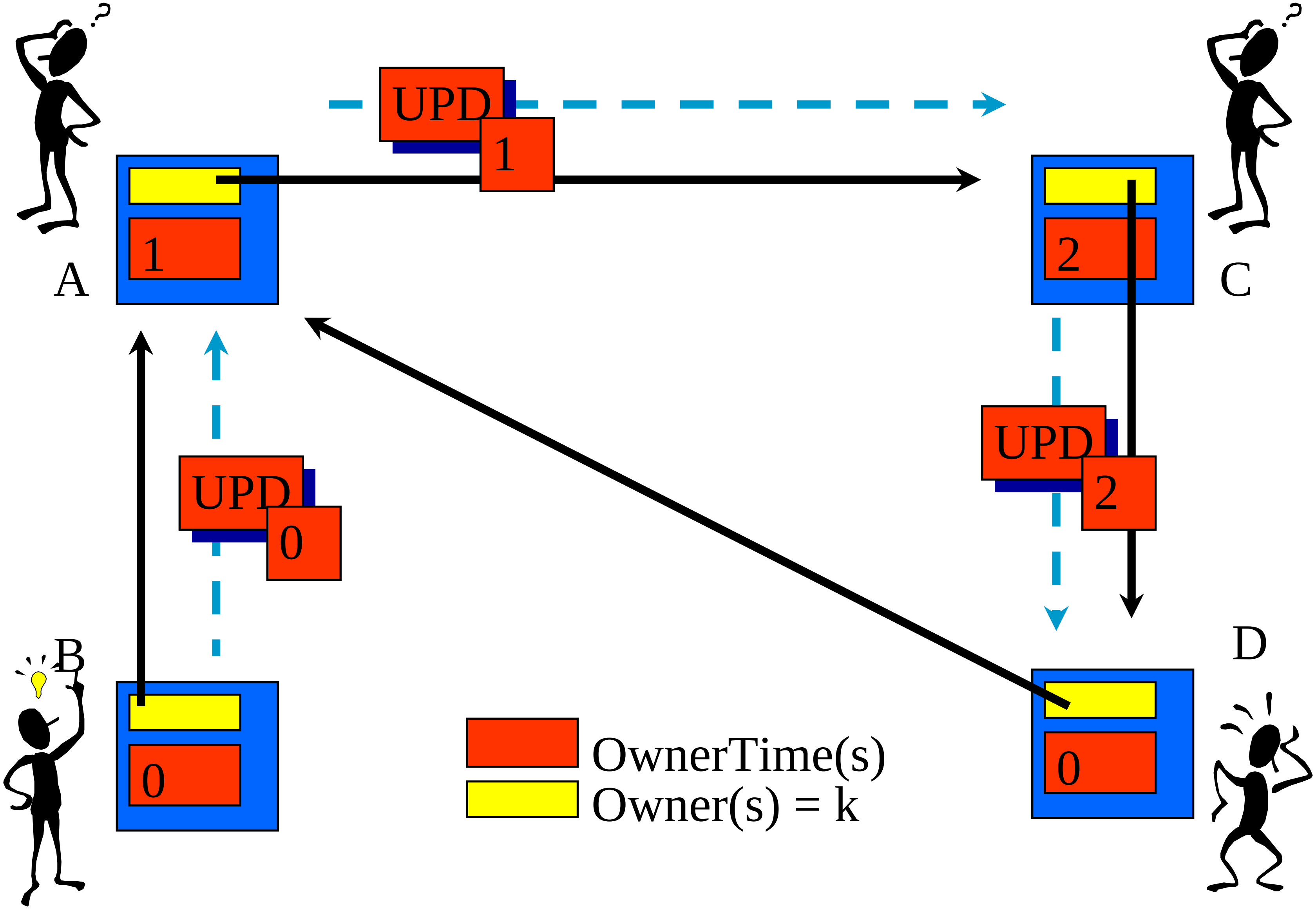


🟥 OwnerTime(s)
🟨 Owner(s) = k

Detecting Token Loss

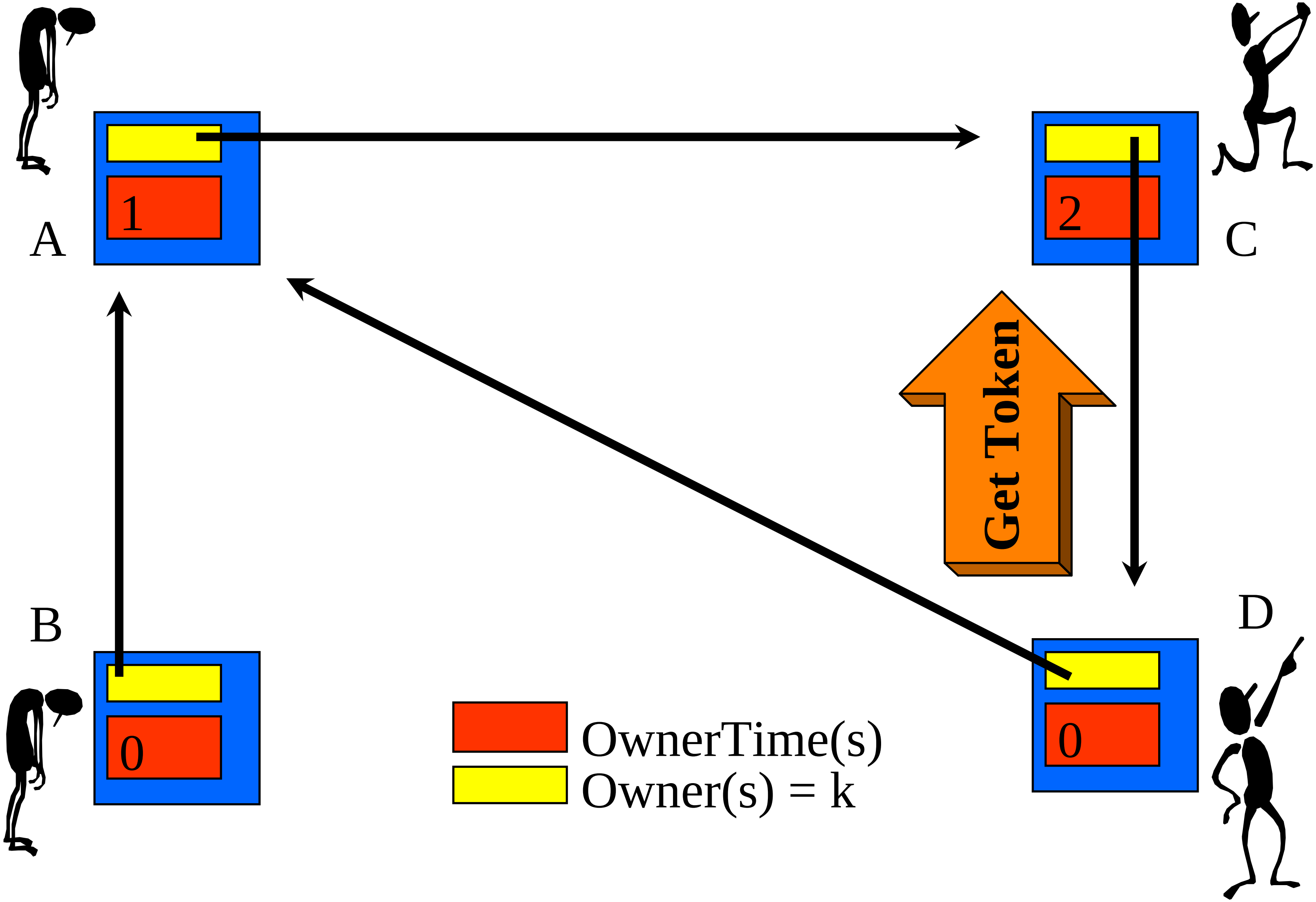


Detecting Token Loss



🟥 OwnerTime(s)
🟨 Owner(s) = k

Regenerating the Token



A

1

C

2

Get Token

B

0

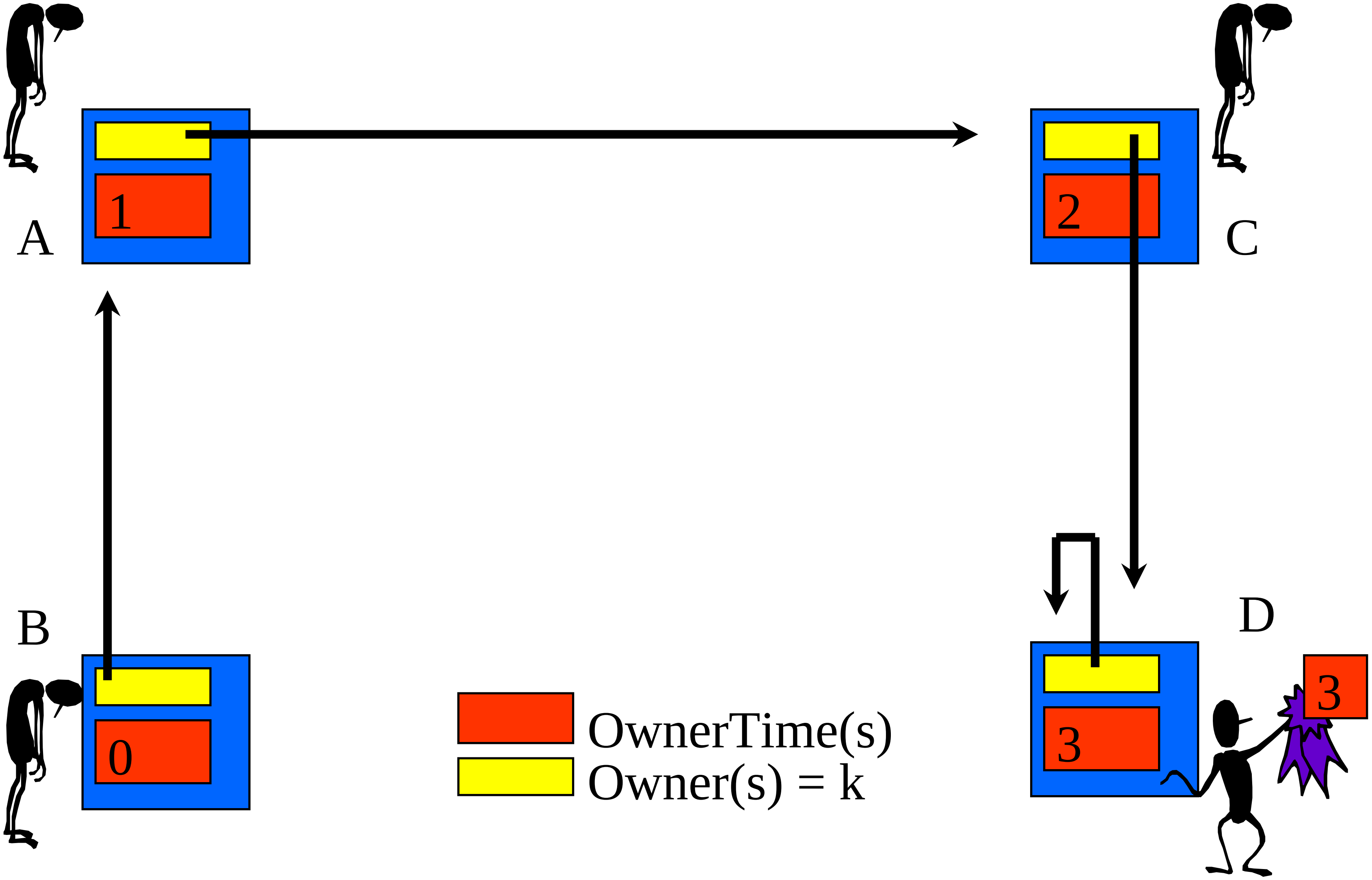
D

0

🟥 OwnerTime(s)

🟨 Owner(s) = k

Token Recovered



🟥 OwnerTime(s)

🟨 Owner(s) = k

Token Loss (algorithm 1)

```
Request (s)
IF Owner(s) = s THEN
    already master copy
ELSE
    SEND(request(s),OwnerTime(s)) to Owner(s)
    Receive(Token,TTime) on Timeout(ReqDelay) ABORT
    Owner(s) = s
    OwnerTime(s) = TTime + 1
    TokenState = Token
END (*IF*)
```

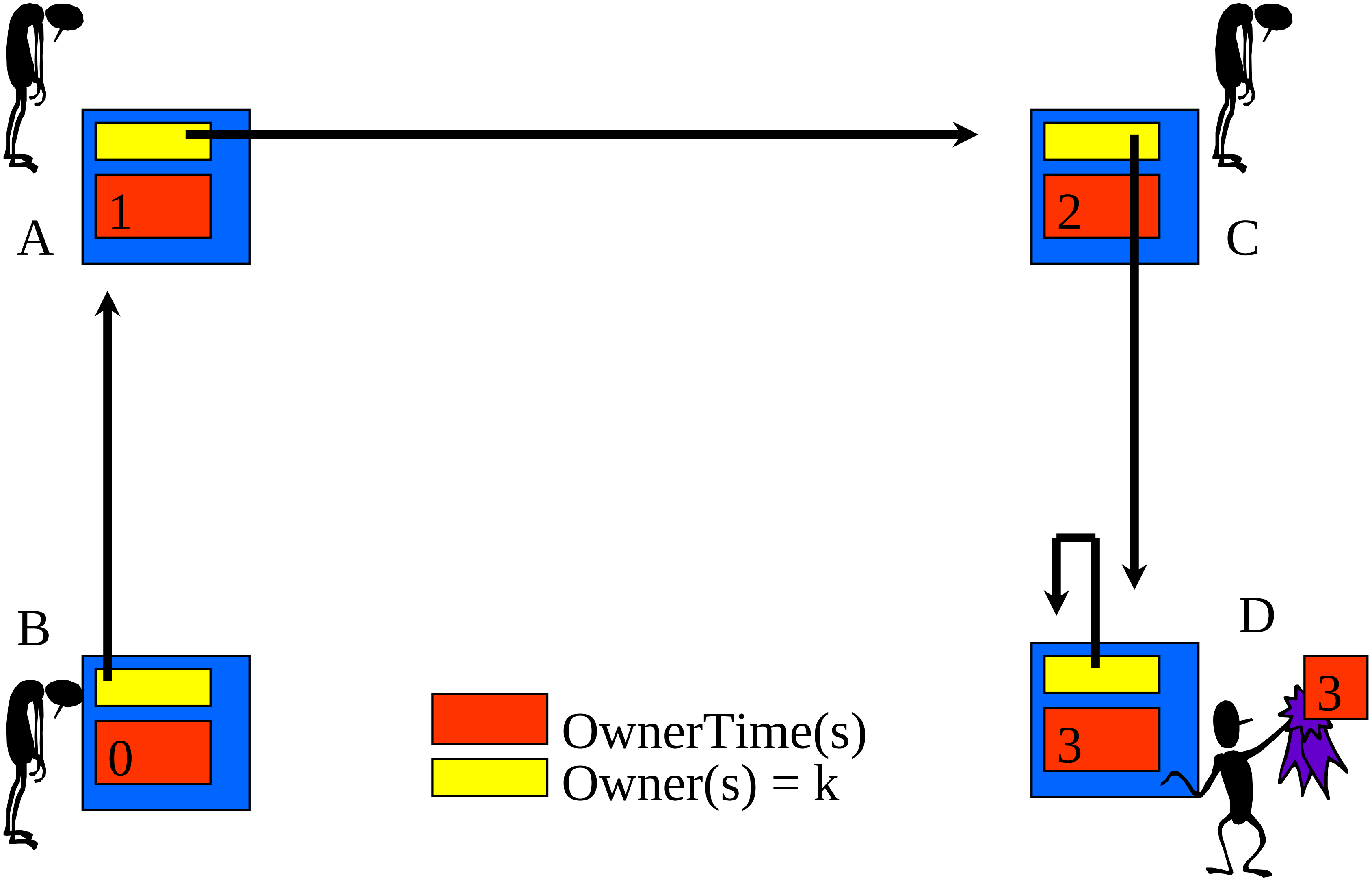
Token Loss (algorithm 2)

```
Receive (request(s),timestamp) FROM p
IF timestamp > OwnerTime(q) THEN (* TOKEN IS LOST *)
    SEND(GetToken) TO p
    Receive(Token,TTime) FROM p ON Timeout ABORT
    Owner(q) = q
    OwnerTime(q) = TTime + 1
    TokenState = Token
END (*IF*)
IF Owner(q) <> q THEN
     SEND(request(s),timestamp) TO Owner(q)
ELSE
    Owner(q) = s
    SEND(Token, OwnerTime(q)) TO s
END (*IF*)
```

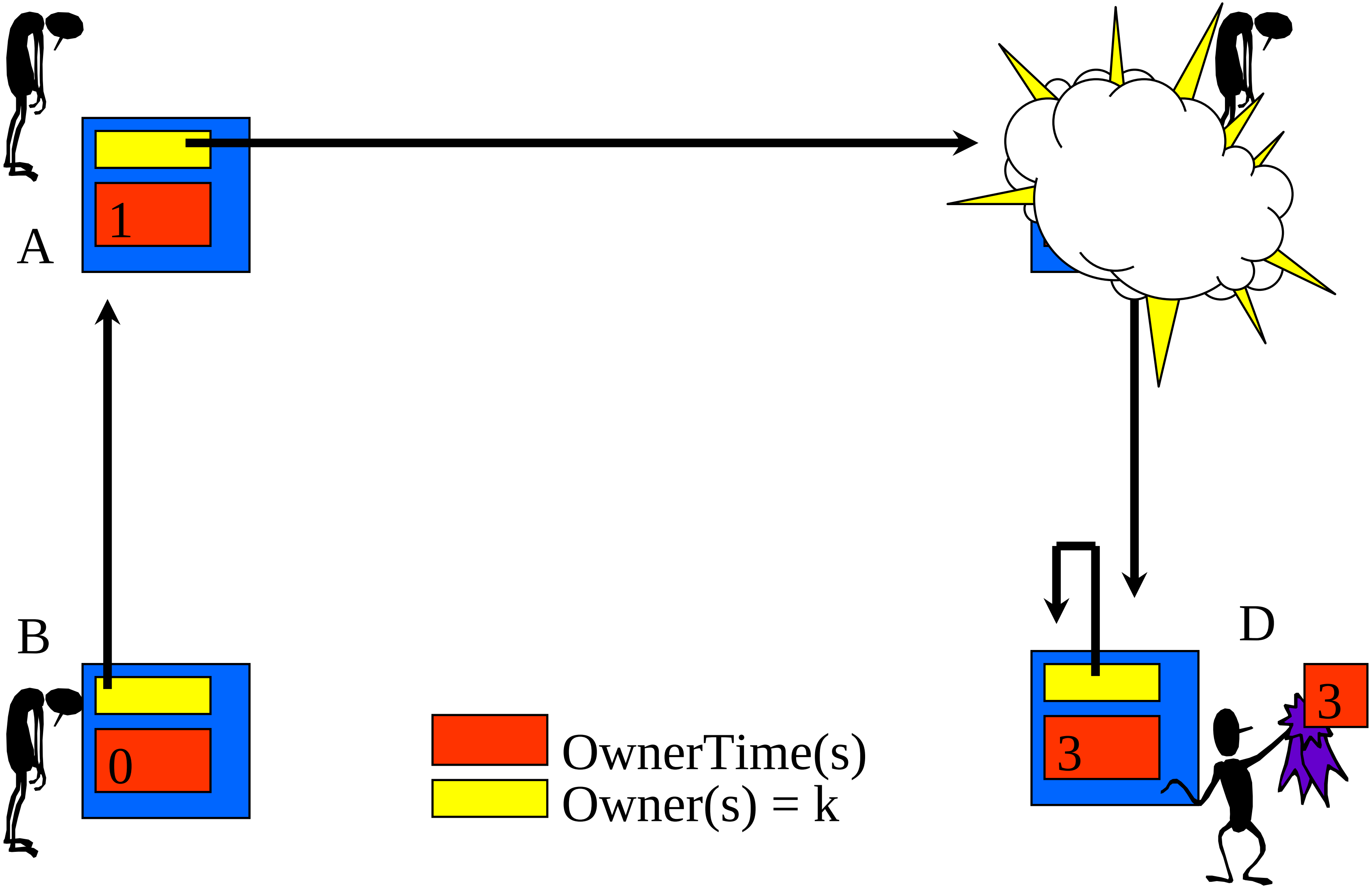
Site Failures

- Sites failures interrupt the chain of pointers (and may also result in the token being lost, if the failed site had the token)
- In this case, the previous algorithm ABORTs the protocol
- Instead of aborting, and to tolerate site failures, a broadcast algorithm can be used to ask everybody and find out what has happened in the system
- Two "states" are used
  - TokenReceived: the site has received the token
  - TokenLoss: a site determines that somewhere in the system there are p,q such that Owner(p) = q and OwnerTime(p) > OwnerTime(q)

Chain Loss due to Site Failure

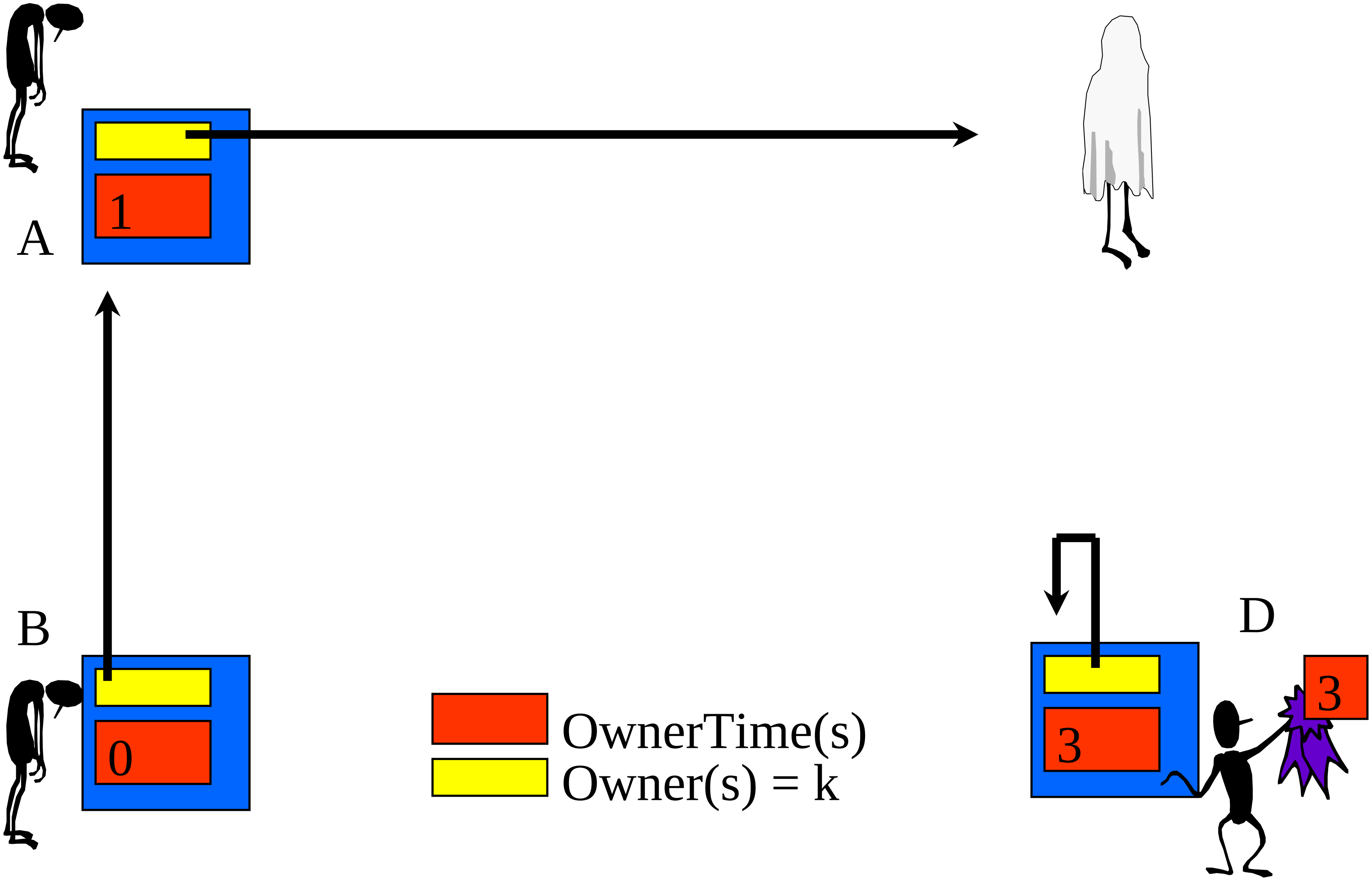


Chain Loss due to Site Failure



🟥 OwnerTime(s)
🟨 Owner(s) = k

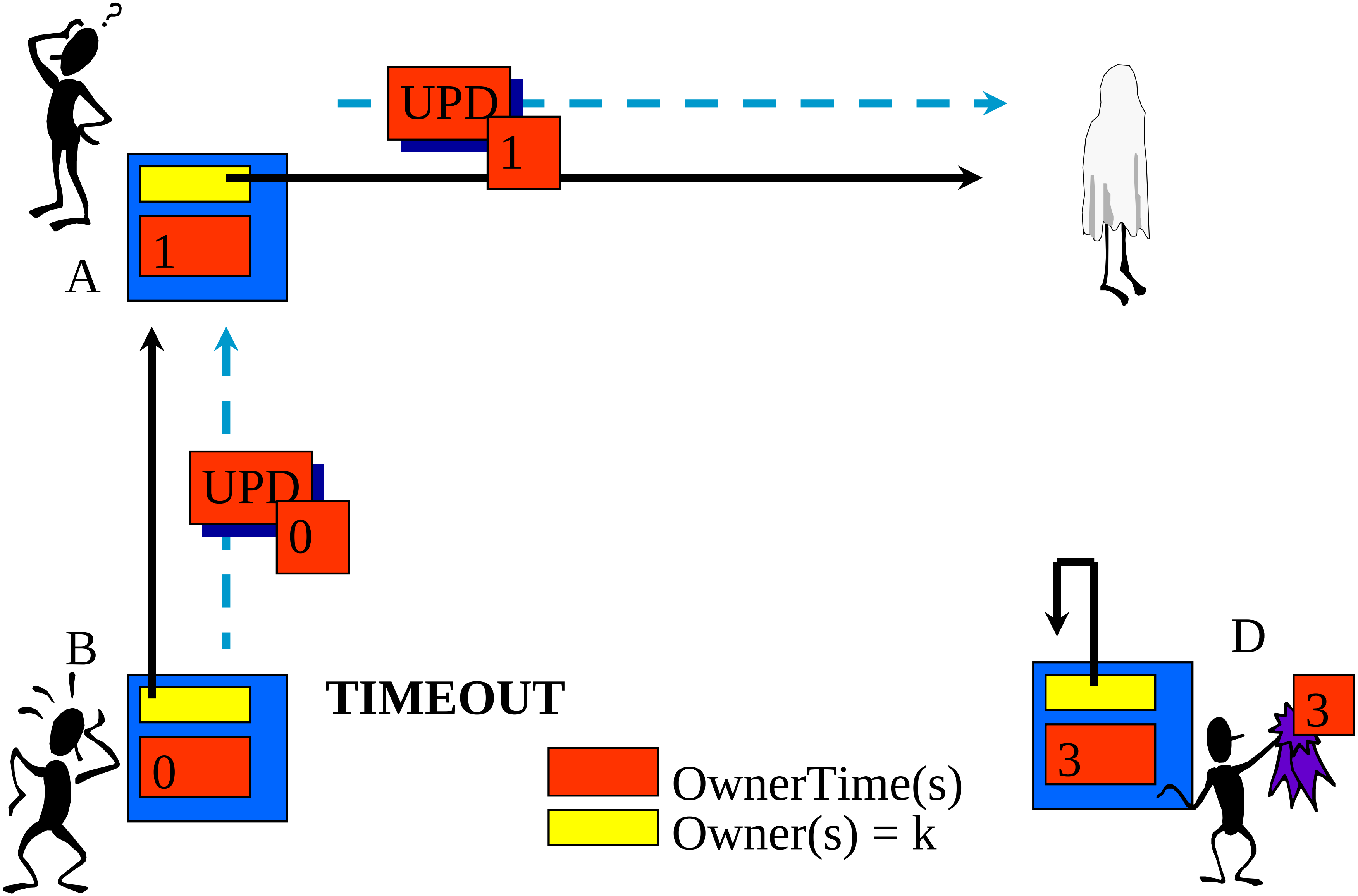
Chain Loss due to Site Failure



🟥 OwnerTime(s)

🟨 Owner(s) = k

Chain Loss due to Site Failure

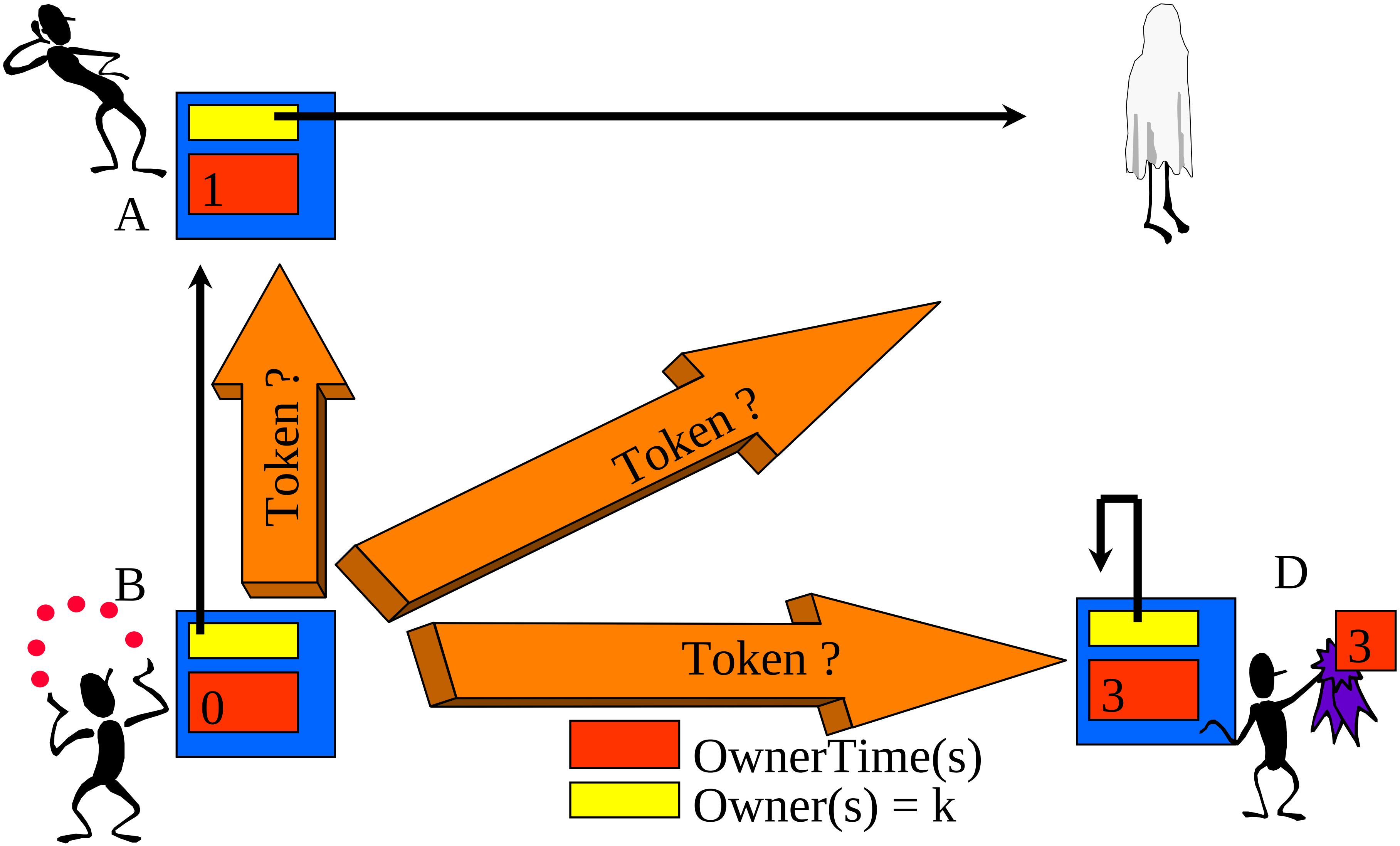


TIMEOUT

🟥 OwnerTime(s)

🟨 Owner(s) = k

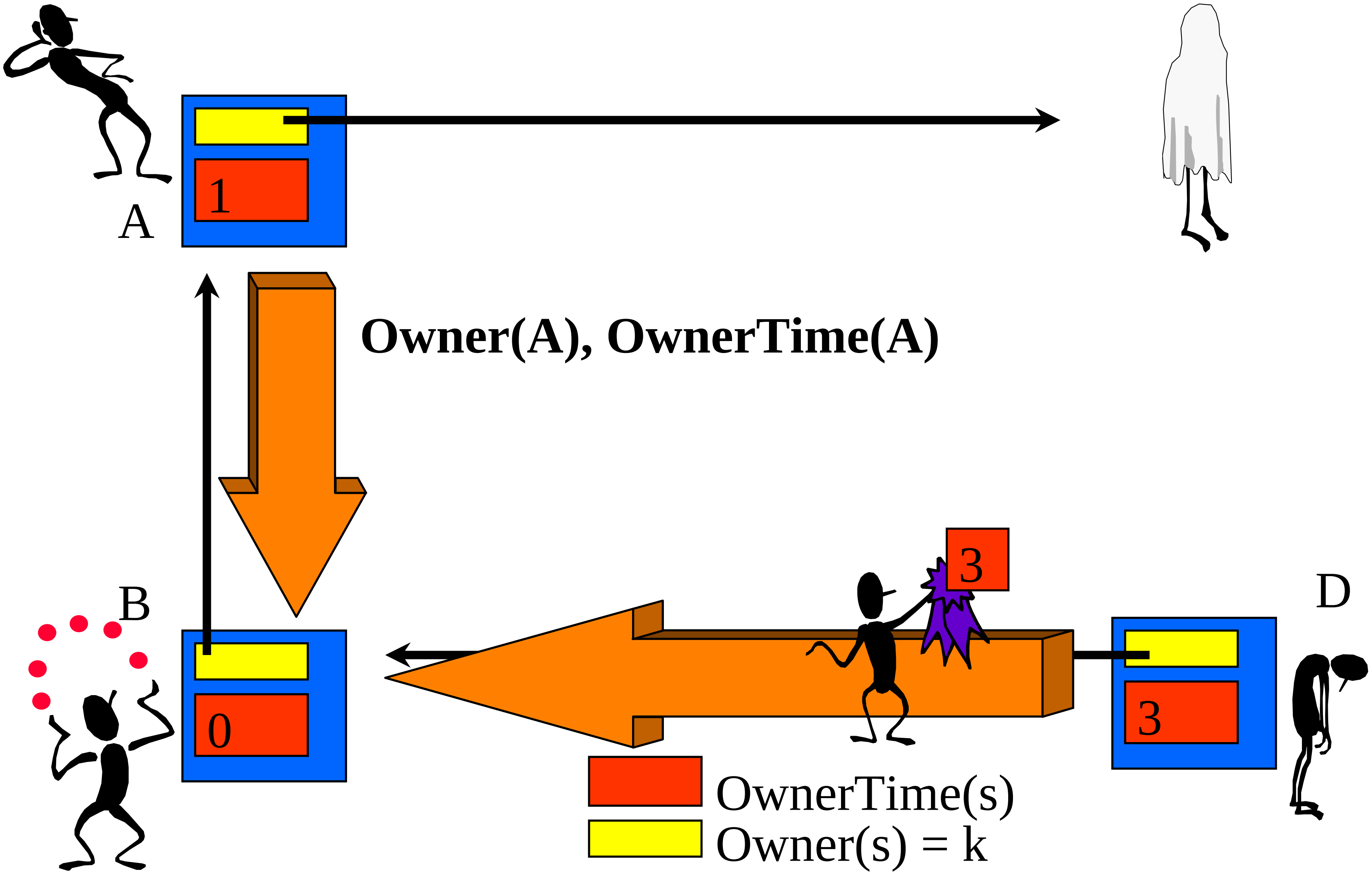
Token Loss due to Site Failure



🟥 OwnerTime(s)

🟨 Owner(s) = k

Token Loss due to Site Failure

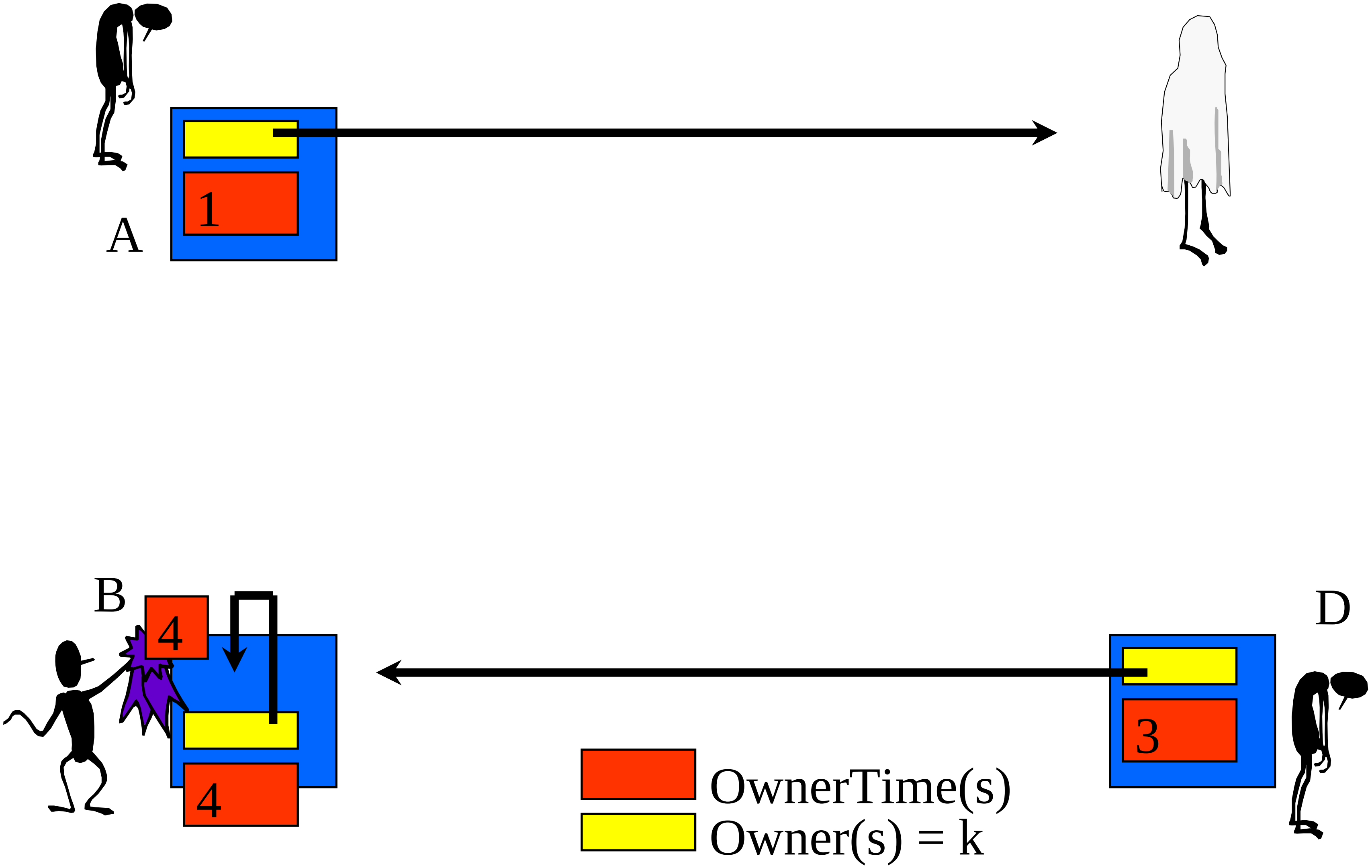


Owner(A), OwnerTime(A)

🟥 OwnerTime(s)

🟨 Owner(s) = k

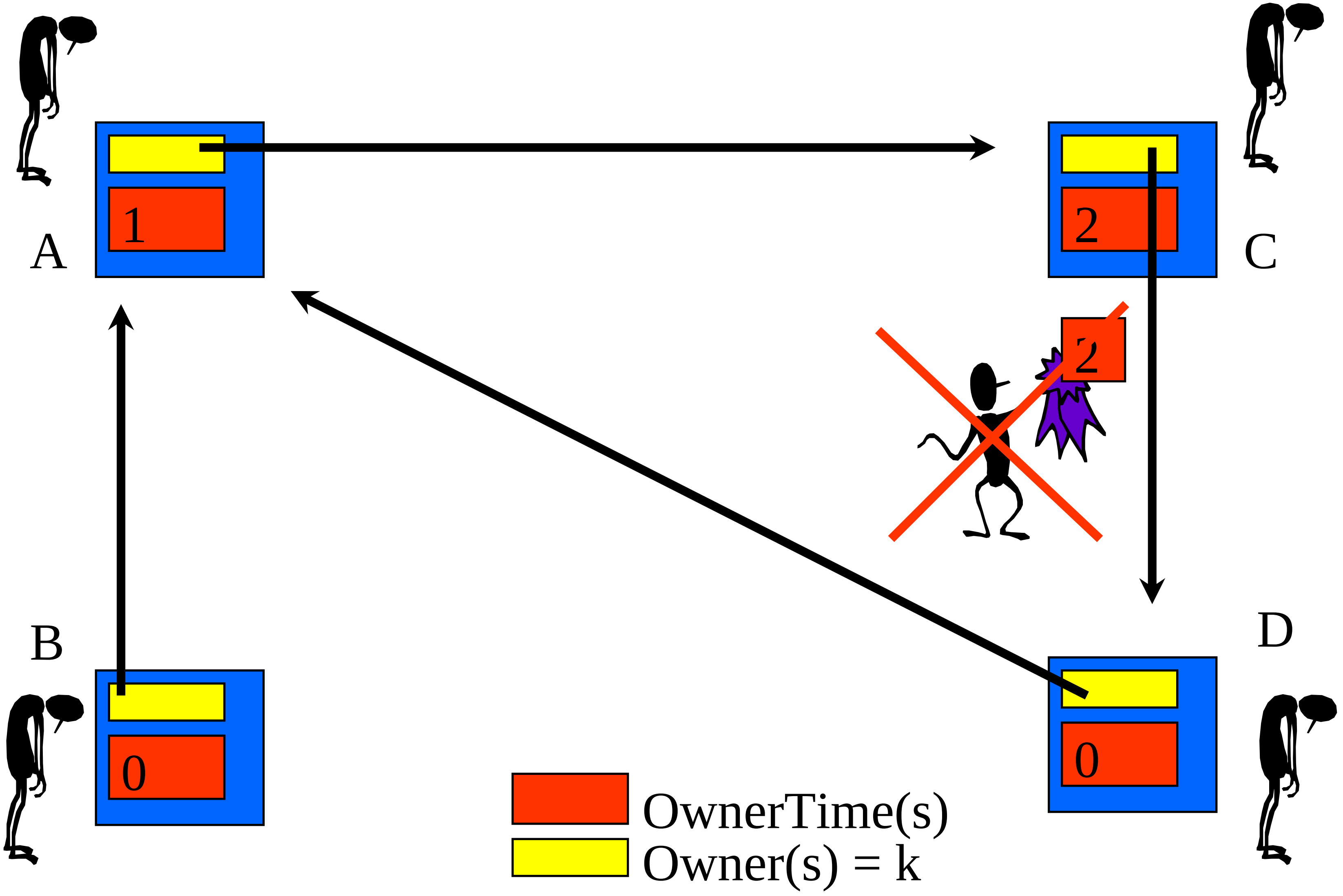
Chain Loss due to Site Failure



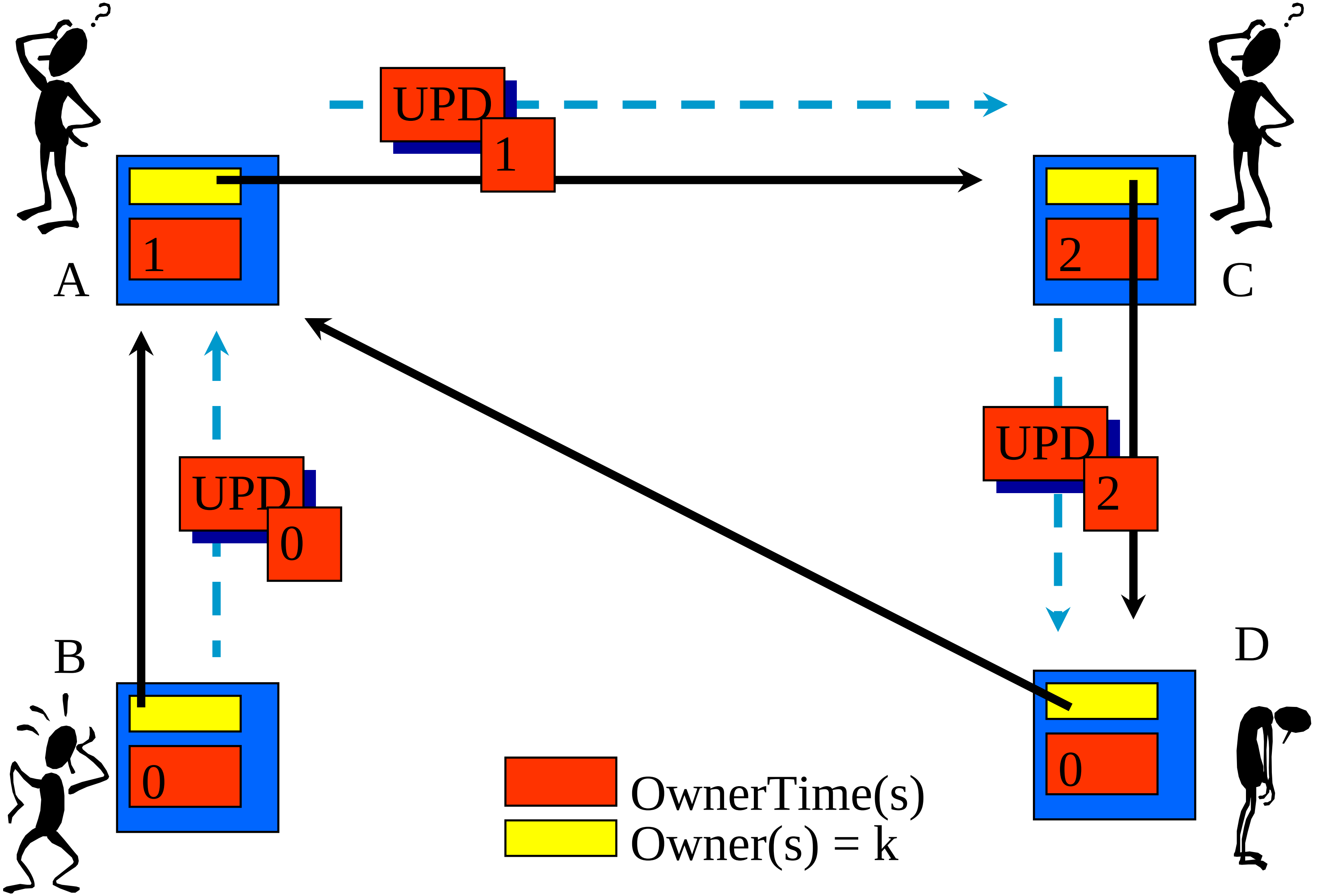
🟥 OwnerTime(s)

🟨 Owner(s) = k

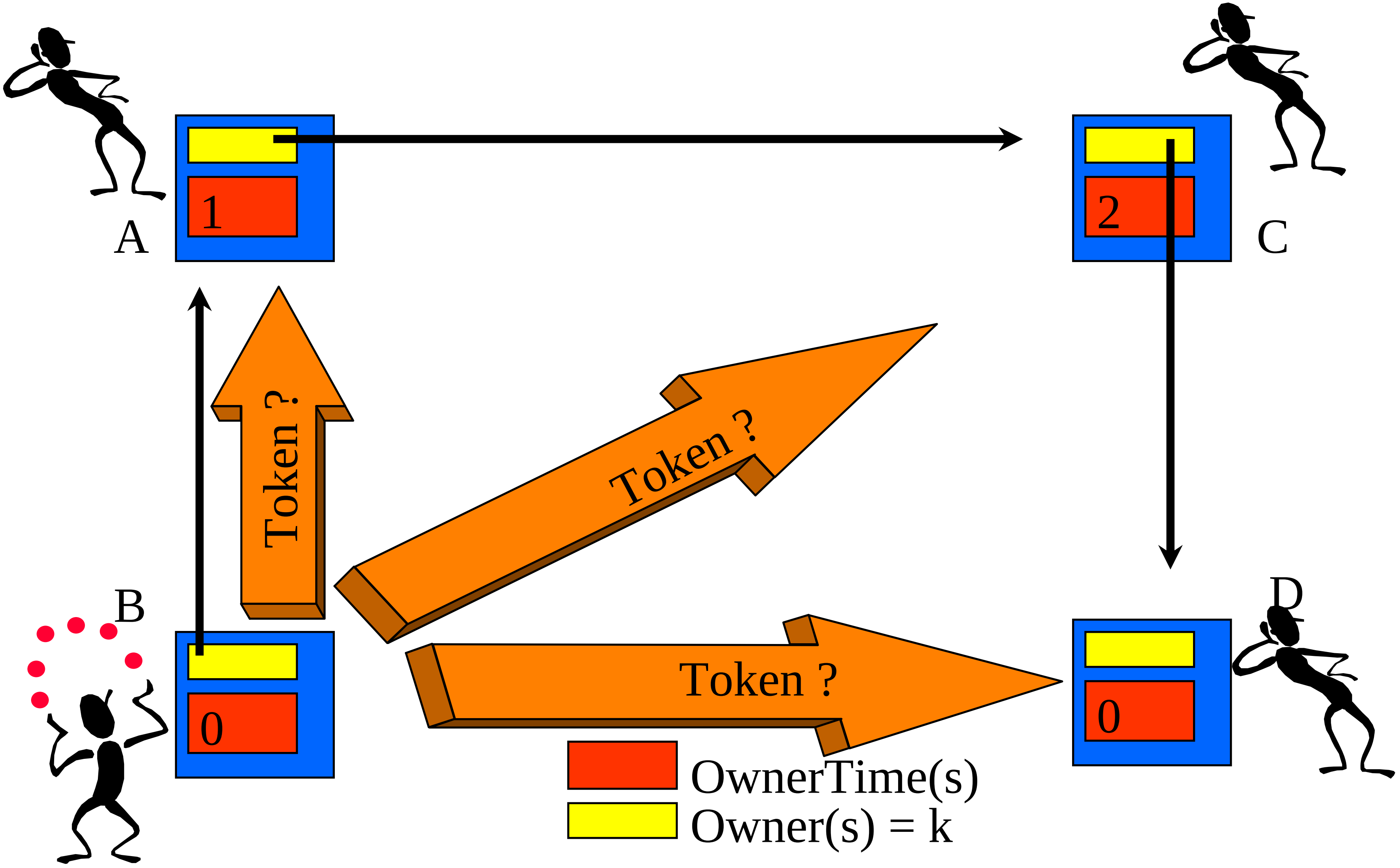
Detecting Token Loss in Others



Detecting Token Loss in Others



Detecting Token Loss in Others



A

1

C

2

B

0

D

0

Token ?

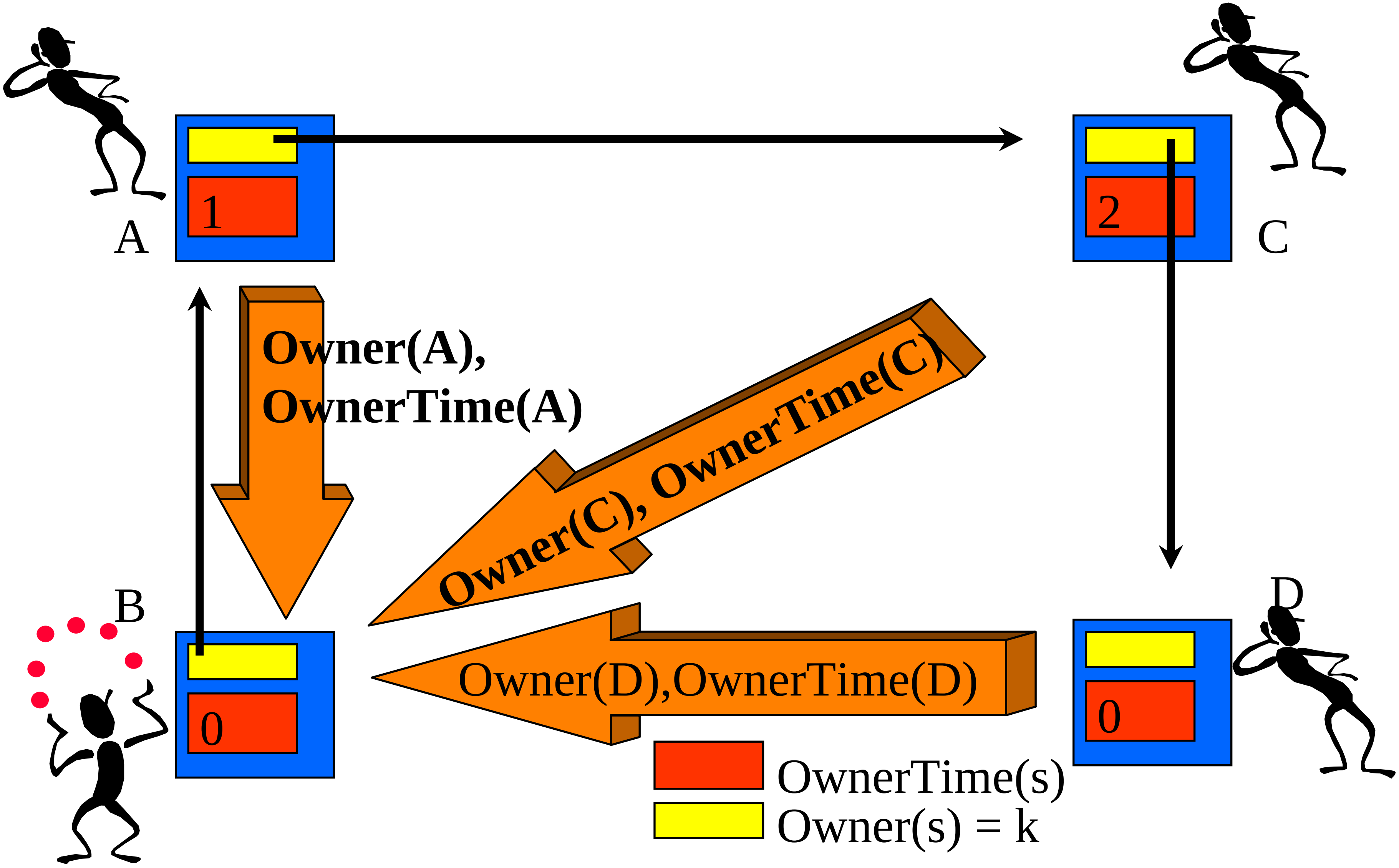
Token ?

Token ?

🟥 OwnerTime(s)

🟨 Owner(s) = k

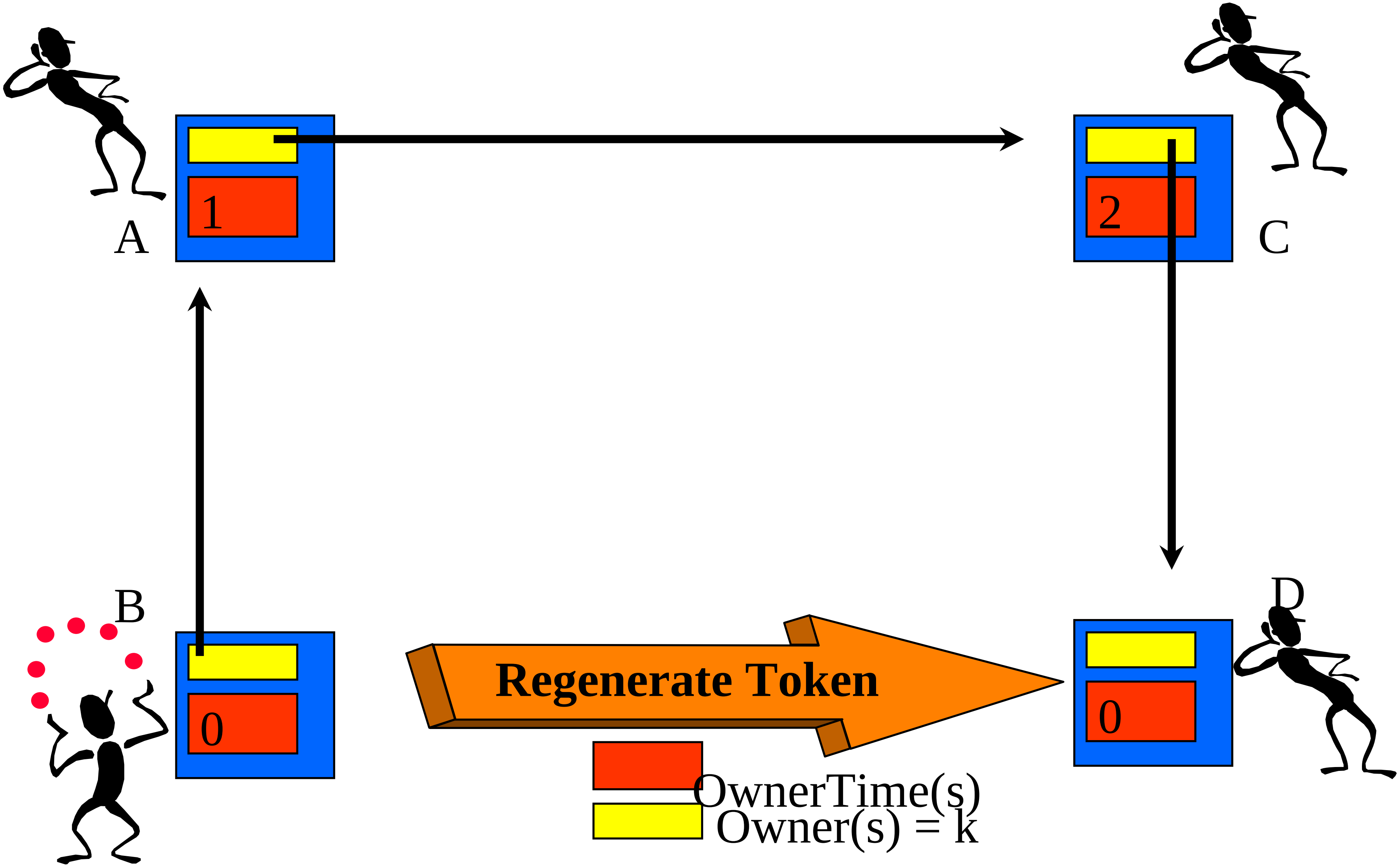
Detecting Token Loss in Others



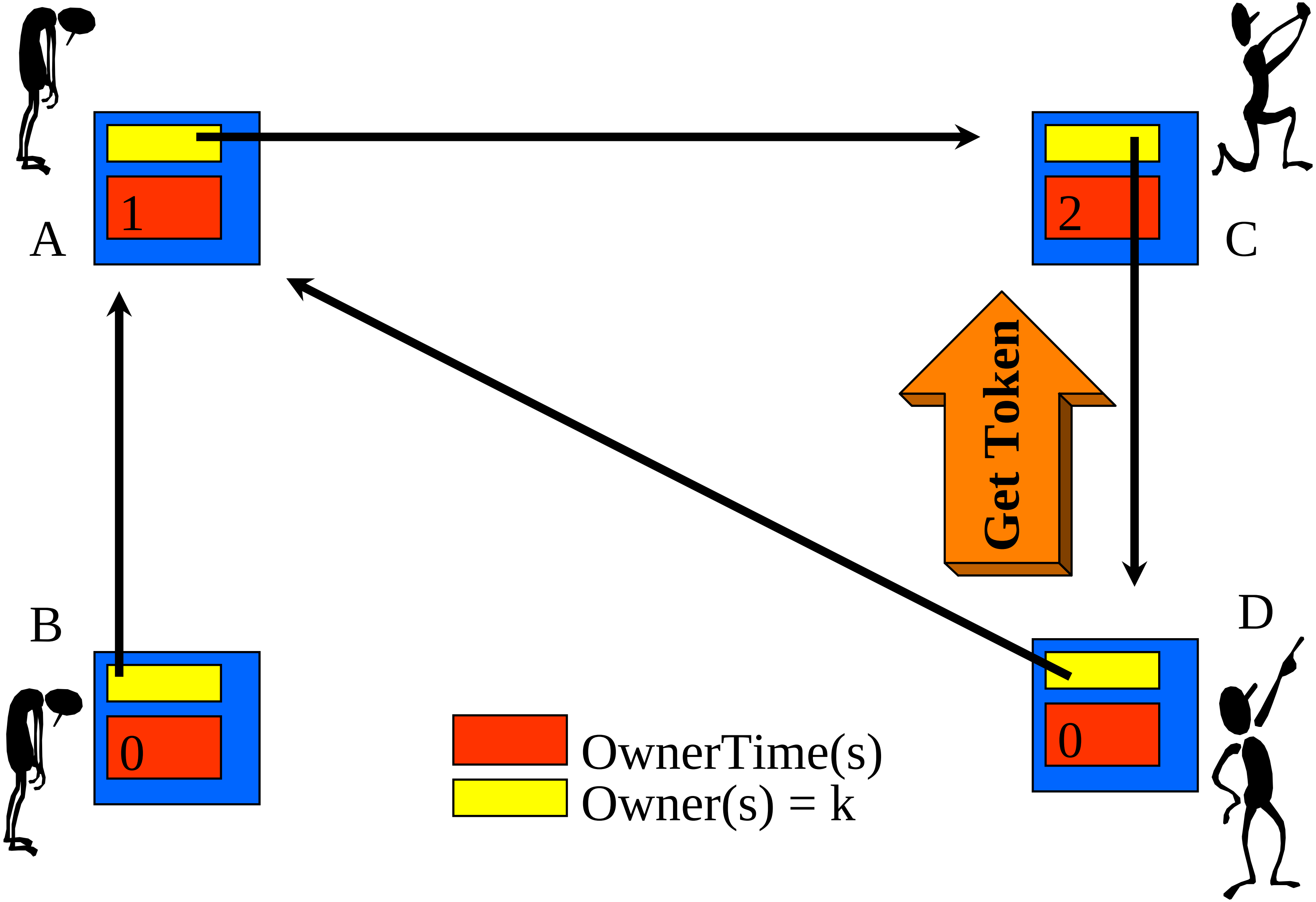
🟥 OwnerTime(s)

🟨 Owner(s) = k

Regenerating Token in Others

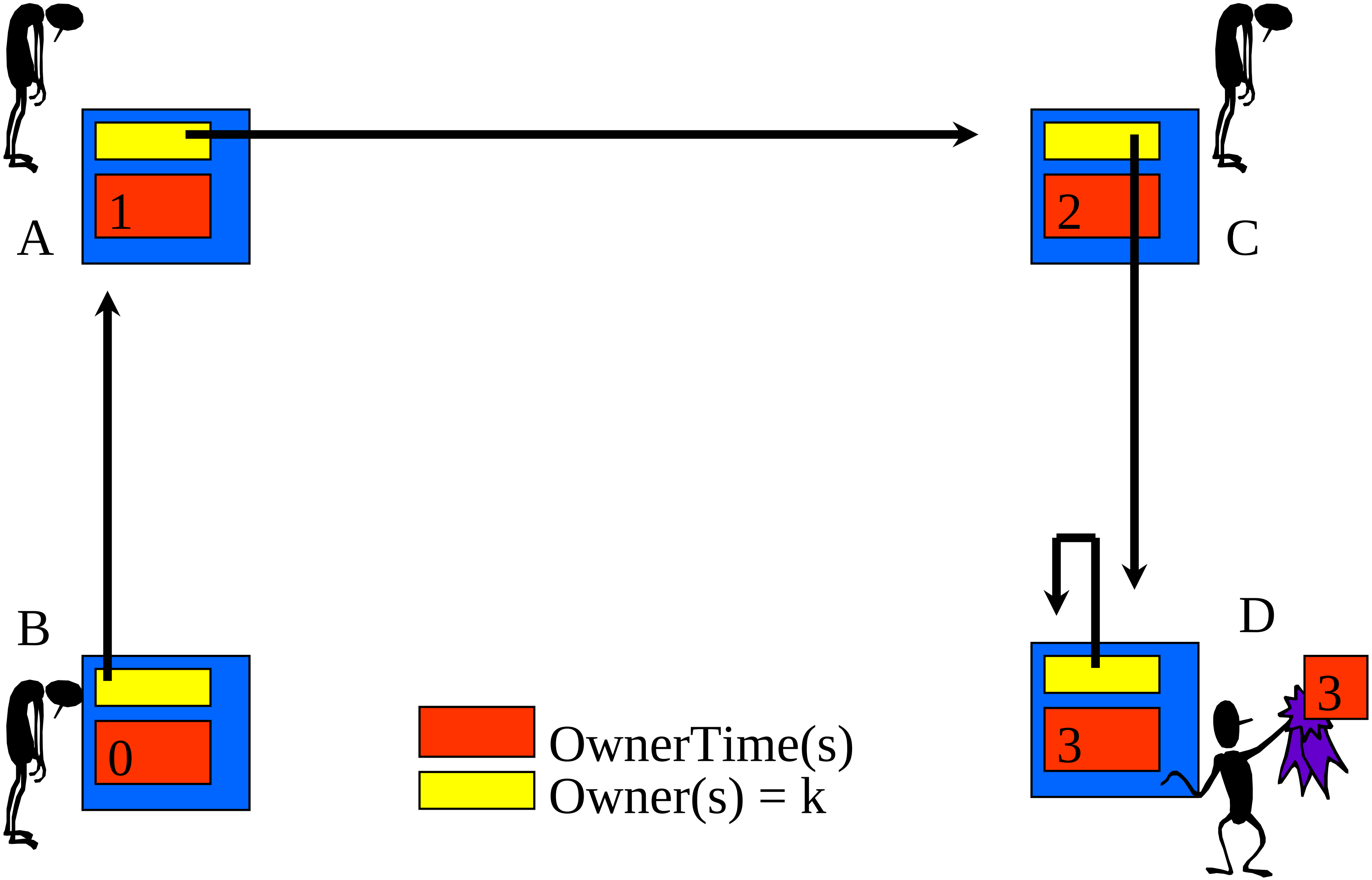


Regenerating the Token



🟥 OwnerTime(s)
🟨 Owner(s) = k

Token Recovered



🟥 OwnerTime(s)

🟨 Owner(s) = k

Broadcast (algorithm)

```
SITE s: SEND (Bcast) TO all sites
COLLECT replies UNTIL TokenReceived OR TokenLoss
IF TokenReceived THEN
    Owner(s) = s
    OwnerTime = TTime + 1
    TokenState = Token
END (*IF*)
IF TokenLoss THEN
    DetectionTime  = OwnerTime(q)
    SEND(Regenerate, DetectionTime, p) TO q
    RESTART
END (*IF*)
```

Broadcast Request (algorithm)

```
Broadcast Request arrives at q from s:
Receive(Bcast)
IF Owner(q) = q THEN
    Owner(q) = s
    SEND(Token,OwnerTime(q)) TO s
ELSE
    SEND(Owner(q),OwnerTime(q)) TO s
END (*IF*)
```

Regenerate Token (algorithm)

A request to regenerate the token arrives at q:

```
Receive(Regenerate, DetectionTime, p)
IF OwnerTime(q) = DetectionTime THEN
    SEND(GetToken) TO p
    Receive(Token,TTime) FROM p ON Timeout ABORT
    Owner(q) = q
    OwnerTime(q) = TTime + 1
    TokenState = Token
END (*IF*)
```

Starvation

- □ Starvation can occur if a request for the token keeps going around the system behind the token but it always arrives after another request
- □ One way to solve this problem is to make a list of all requests, order the requests by timestamp and only grant a request when it is the one with the lowest timestamp in the list
- □ The list can be passed around with the token and each site can keep a local copy of the list that will be merged with that arriving with the token (thereby avoiding that requests get lost in the pointer chase)