

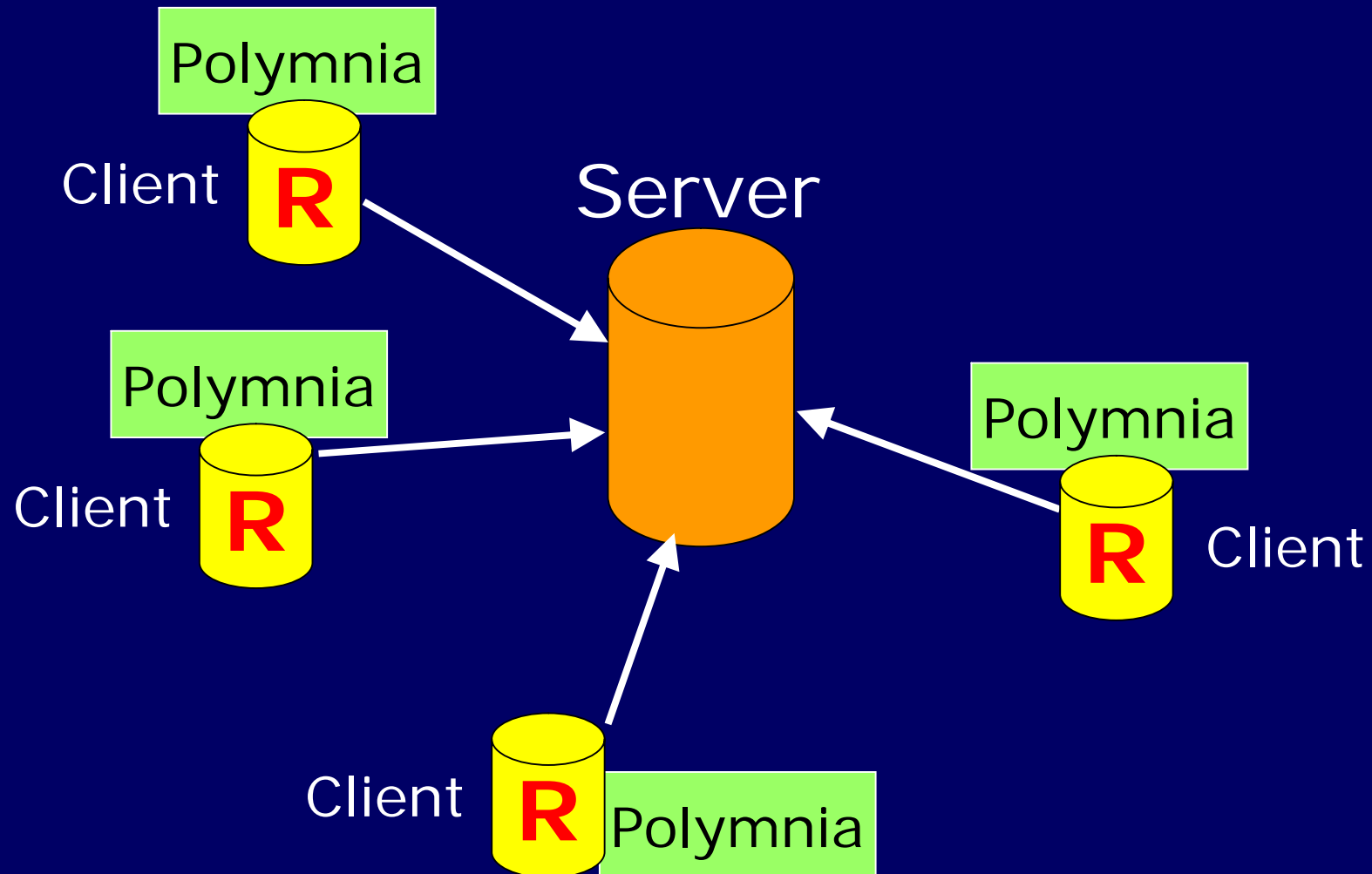
1. My project Polymnia

- Replication on PostgreSQL 7.3
- Asynchronous Multi-Master-Replication with a Primary Copy
- 2-layer protocol
- With User-Interaction (only as much as really needed)

The project itself

- Administration Software for the Music School Association of the Austrian Federal State Burgenland
- Distributed Database Application (each school runs at least one node of the network) with overlapping data
- Nodes don't have constant access to the net

The System - Overview



The concept

- Users are only allowed to change their own secondary copy.
- Changes are propagated from and to the server via a synchronization procedure, called automatically or by the user
- This procedure consists of two layers

The lower layer I

- Consist of two phases
 - **Update Phase**: propagates the changes from the server to the client by solving upcoming conflicts
 - **Commit Phase**: propagates the changes from the client and the database merging during the Update Phase to the server – client needs exclusive right to write on the server during this phase

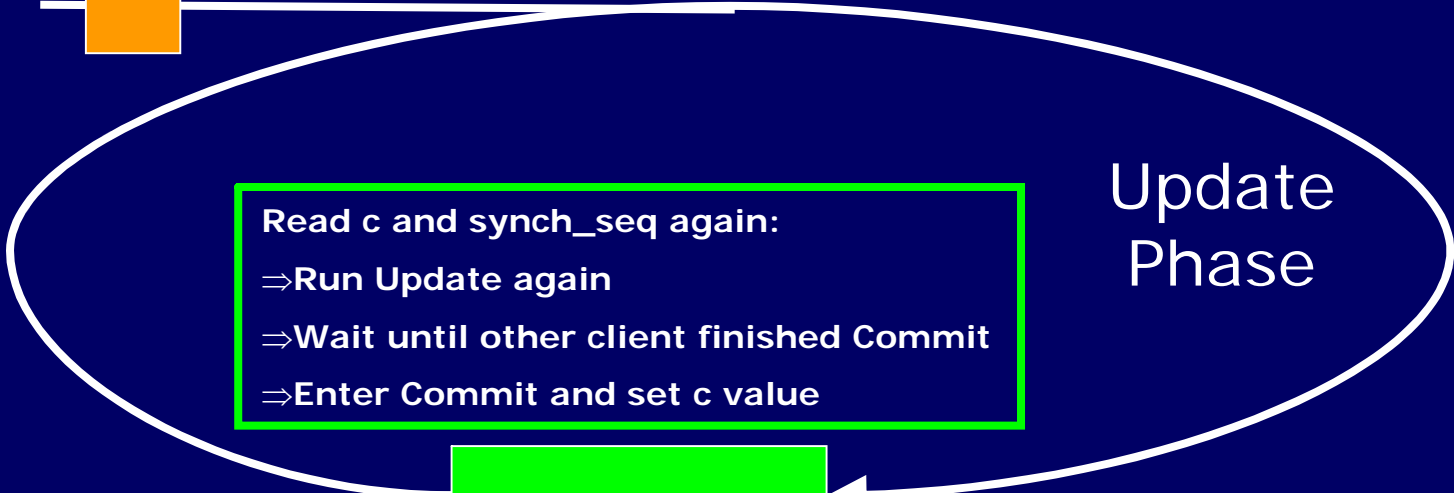
The lower layer II

- In each synchronisation process the clients are getting a unique number – the **synchid**
- When tuples are inserted or updated in the local database, they are always marked with the last **synchid** got from the server in order to mark them for propagation in the synchronisation procedure

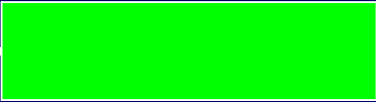
The lower layer III

- The server provides a flag `c` protected by a Semaphore `sem` indicating whether there is a client currently in the **Commit-Phase** or not.

Read c;
Read synch_seq;



Commit Phase



Reset c value;
Commit Transactions on Server Side;
Commit Transaction on Client Side;

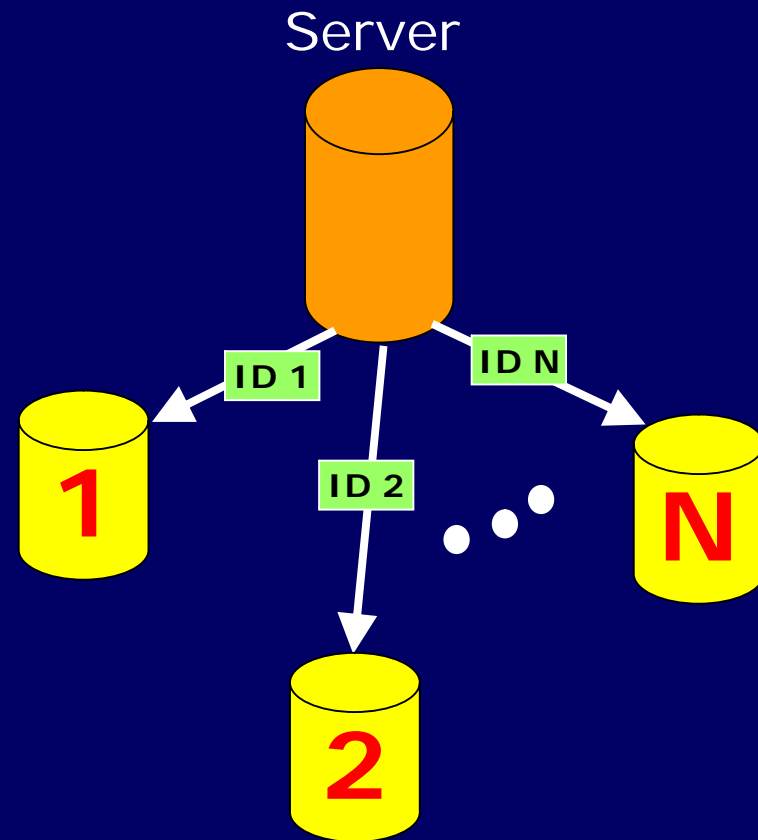


The upper layer - Overview

- Is based on the lower layer
- Handles the conflicts
- Includes User-Interaction

Prerequisites

- Each Table has its own sequencer. Ticket is default value for the primary key of the relation
- Keys have to be unique over the system.



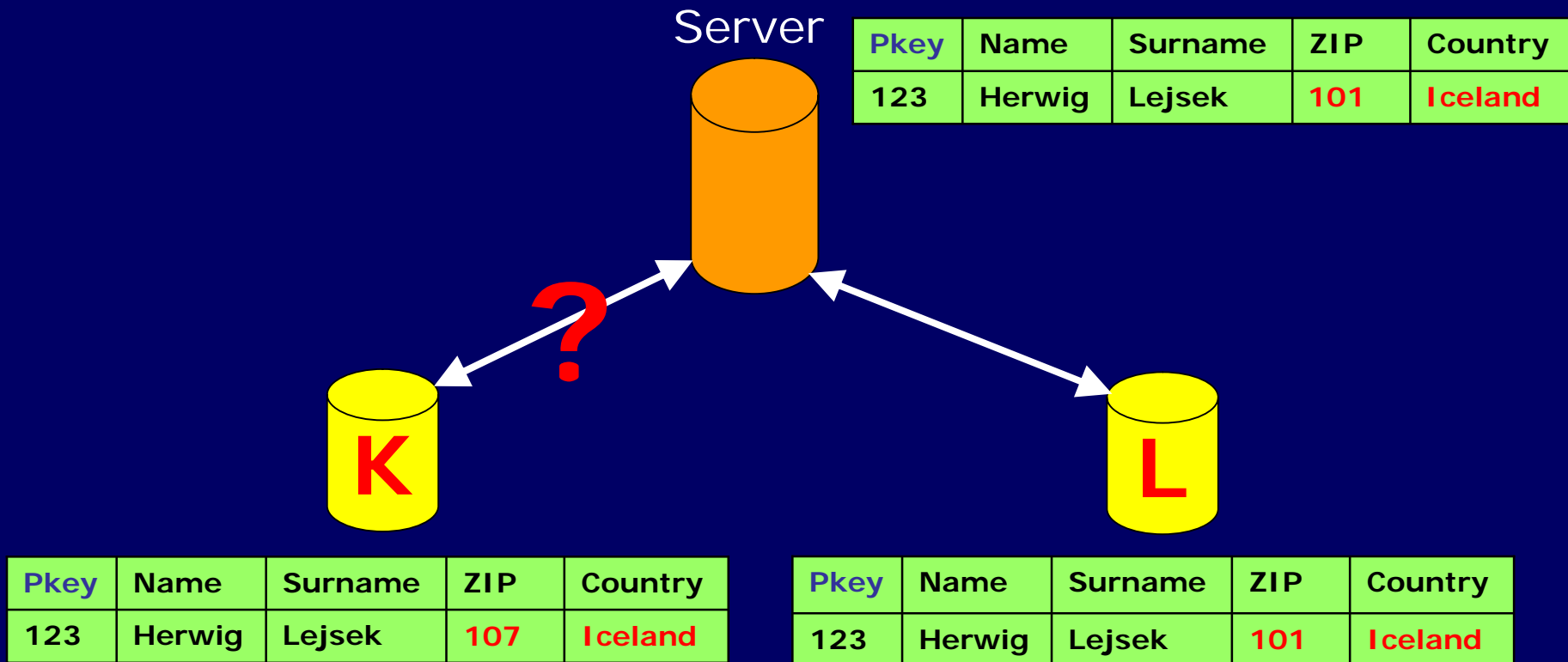
Initialization of a new Client

- Get a clientID from the Server
- Update the Primary Key of all Records in the Database with
 $pkey += clientID * MAX_TUPLES$
- Update all sequencers with $MAX(pkey)$
- SynchronID (of all records) has to be set to the lowest possible value

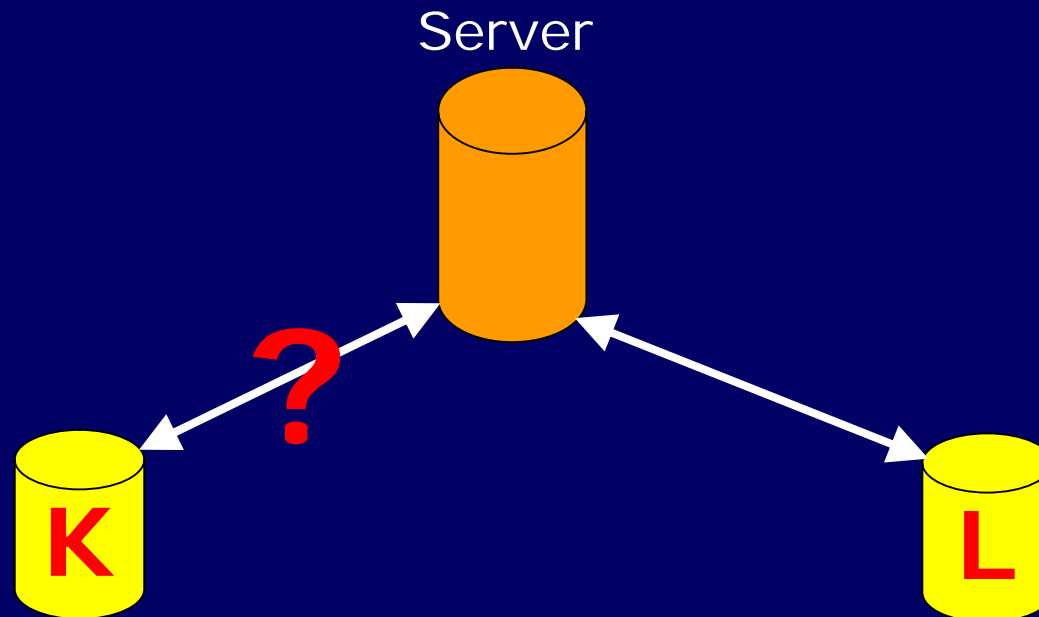
Conflict types

- Double-Update Conflict
- Double-Insert Conflict
- Unique Conflict

Double Update



Double Insert



Pkey	Name	Surname	ZIP	Country
245	Herwig	Lejsek	107	Iceland

Pkey	Name	Surname	ZIP	Country
376	Herwig	Lejsek	101	Iceland

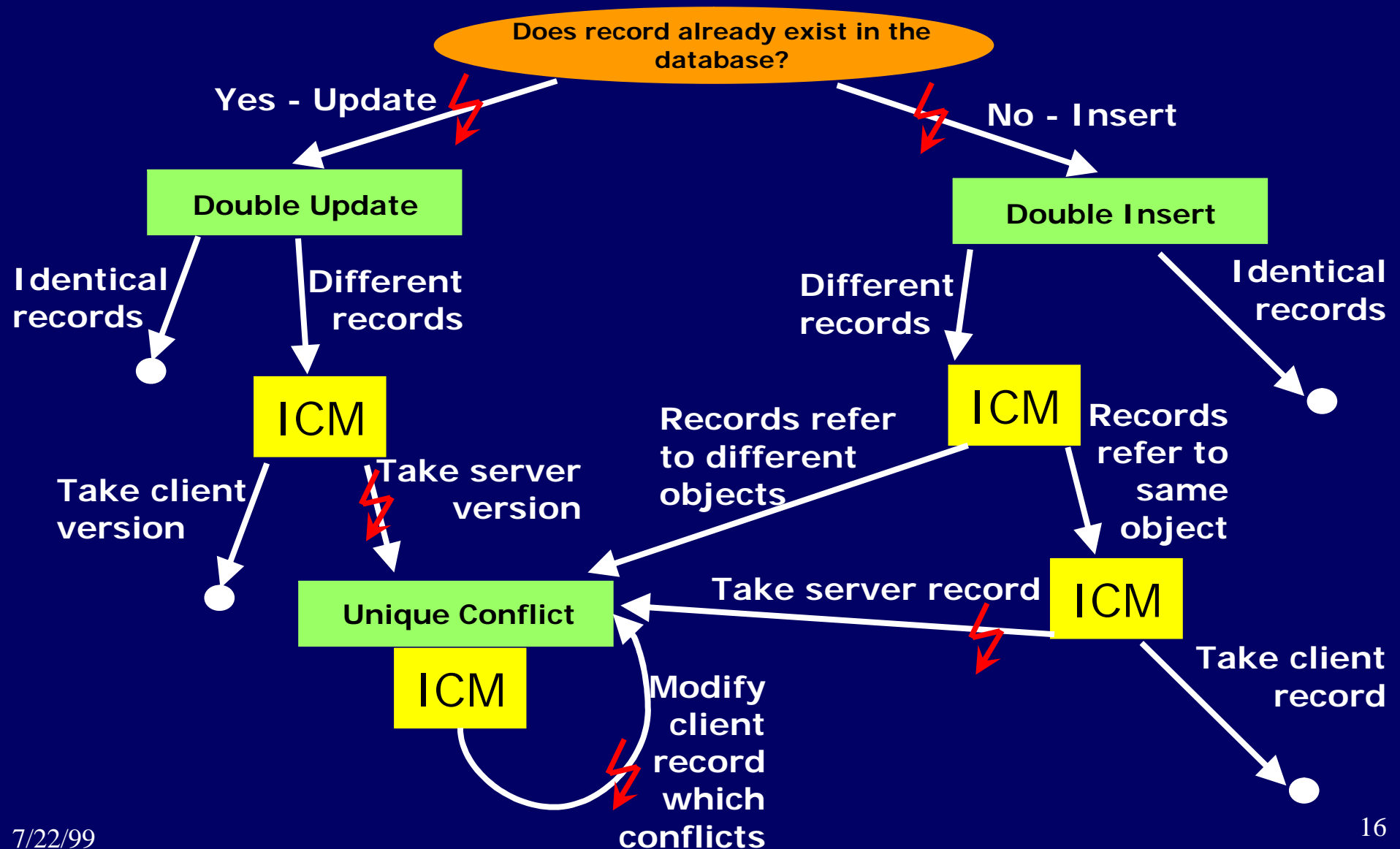
Unique Conflict

- Are these two persons or only one?

Name	Surname	Birth	City	Car
Herwig	Lejsek	13.Sept.1979	Vienna	Austin

Name	Surname	Birth	City	Car
Herwig	Lejsek	13.Sept.1979	Vienna	Ford

The Update Phase



Between Update & Commit Phase

- After the Update Phase the context of primary copy is a subset of the client's secondary copy, except records changed through conflict management.

The Commit Phase

- Pretty like the Update Phase but without conflict handling (are all solved at this state)
- Main problem: cyclic dependencies of changes

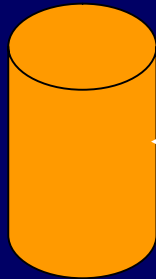
Failures during Commit Phase

- Unique Failures might occur when inserting or updating records to the server due to records which should be overwritten during the Commit phase later on
- Handling: Skip the tuple and requeue it behind all other records waiting for propagation to the server

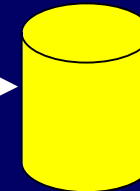
Cyclic permutations

- Users always can be mean:

Server



Client



Pkey	Name	Username
123	Alice	xyz
124	Bob	abc

Pkey	Name	Username
123	Alice	abc
124	Bob	xyz

Detection & handling of cycles

- By counting the number of consecutive requeues compared to the number of records waiting in the queue
- If a cycle is detected the Unique Constraint causing this problem has to be abrogated for the next two records in the queue.

Deletions

- Are generally problematic in a highly distributed system with lots of users accessing the same data
- concept of storing the relations to „global active objects“ for each user
- These relations are marked as active for a certain time. Users can „delete“ them by simply deactivate them.

The Activation Concept

- All records in the database (not only in the table) have to have a global ID.
- All active records are referenced in an own table „**active**“, which indicates if records are currently active
- Activations and Deactivations also have to be propagated

The Activation Concept II

- Deletions on the table „**active**“ are the only deletions performed during the synchronisation.
- Deletions from „**active**“ is only done on the client after finishing the Update phase. It gets all deactivations since the last synchronisation from the server and executes them on its own replica.

Further Work

- Work regarding activation is still in progress.
- Restricted Activation (for a limited time, limited user group)
- History Mode (looking back, how the state of the system was some time ago)