Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Data Synchronization Across Many Concurrent Peers

Cristian-Viktor ARDELEAN

Master's thesis in the Elite Graduate Program: Software Engineering

Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Data Synchronization Across Many Concurrent Peers

Matriculation number:	1276933
Started:	4. June 2014
Finished:	4. December 2014
First assessor:	Prof. Dr. Alexander Knapp
Second assessor:	Prof. Dr. Bernhard Bauer
Supervisors:	Dipl.-Inf. Univ. Ralf S. Engelschall
	Dipl.-Inf. Univ. Peter Huber

SOFTWARE ENGINEERING
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

I, hereby certify that this thesis has been written by me, that it is the record of work carried out by me and that I have not used anything else but the indicated sources and tools.

Augsburg, den 2. Dezember 2014 Cristian-Viktor ARDELEAN

ACKNOWLEDGEMENTS

This master thesis has been carried out as part of the Elite Software Engineering master degree program at the University of Augsburg, Technical University of Munich and Ludwig Maximilians University Munich in collaboration with msg systems ag, since spring 2014. A number of people deserve thanks for their support and help. It is therefore my greatest pleasure to express my gratitude to them all in this acknowledgement.

First and foremost, I would like to express my sincere gratitude to my industrial supervisors Dipl.-Inf Univ. Ralf S. Engelschall and Dipl.-Inf. Univ. Peter Huber who supported me throughout my thesis with their patience, motivation, enthusiasm and immense knowledge. I am deeply grateful to them for the long discussions that helped me sort out the technical details of my work. One simply could not wish for a better or friendlier supervisors.

My sincere thanks also goes to my university supervisor Prof. Dr. Alexander Knapp for his continuous support and feedback. I am very grateful for the time he spent listening to my ideas and for all the assistance accorded.

Special thanks goes also to my fiancée Raluca Marchis, and to my friends Andrei Mituca and Bogdan Lile for their invaluable input during the review phase of the thesis.

My deepest gratitude goes to my family for their unflagging love and unconditional support throughout my life and my studies.

Finally, I appreciate the financial support from the German Academic Exchange Service (DAAD) that funded my studies here in Germany.

Abstract

CONTEXT The era of mobile devices has changed the way to build web applications. User experience has become an important factor for the success of an application. This led to a switch from Thin-Clients to Rich-Clients with the big advantages that the data is now replicated onto the client and the ability to redraw any part of the User Interface (UI) without requiring a server roundtrip to retrieve HTML. The software architectures faced an impact: a separate data model is now required on the client.

MOTIVATION This gained advantage of Rich-Client applications raised the difficulty of having fresh data on all cache levels: server database, server in-core model, network protocol, client in-core model, client cache, client UI presentation model. To achieve this, fresh data must be sent to all connected peers. HTTP, the foundation of the Web, is a client-initiated request/response based protocol, which means that there is no way to initiate the sending of messages from the server to the client without having the client create a request first. In modern web applications this is a common restrain, which led to different workarounds (long polling and deferred responses using AJAX). Real-time web applications is the buzzword that started to make its way to the Web.

The key aspect behind this is a bidirectional connection between network peers. In this way every peer gets notified immediately when data has changed on other peers or new data is available. With such a platform in place, smart “lock-less” web applications can be built, raising user-collaboration at a new level and facilitating new domain specific use-cases. For instance, users will be able to edit at the same time a form in two instances of a web application. With real-time peer communication the form data can be synchronized faster between peers, so that there is no need to lock the whole form for a single user. A smart real-time web application would assume the possibility to work offline using a local cache. At the moment it reconnects to the other peers, it needs to resynchronize its data model.

CHALLENGE The first challenge for synchronizing data models between peers is to know at data modeling time what data needs to be kept in sync at all. A solution must be found to attach sync information to the heterogenic data models of the peers. To synchronize the data across peers we need to deal with different data types that need different merge semantics in case of a modification conflict. A helpful feature would be also data model snapshotting/versioning for handling undo/redo operations.

For detecting possible conflicts between peers the data model needs to be stateful: to know which peer is editing the data and which is just reading it. After knowing what action (read/write) each peer is performing, all obsolete caches must be invalidated. In case of write-write conflicts on the same peace of data, a conflict resolution strategy must be in place and resolve the conflict.

The data latency issue over the network must be also concealed, giving the users a real-time user experience. The data access module of the peers must know if the application is currently running in connected or disconnected mode. With this information it will know which data store to use when retrieving and updating data. All this must

work over many network protocols (WebSockets, XMLRPC, WAMP) and for different network topologies (Client/Server, Hub-and-Spoke, Peer-to-Peer).

APPROACH We resolve the above challenges with the following approach: First, we need to check partial conceptual solutions for attaching sync information to technology-dependent data models. After that an overlay protocol must be defined to know the actions performed by each peer on each piece of data. For data synchronization, different strategies must be analyzed (Event Passing, Three-Way Merging, Differential Synchronization, Dual-Shadow Method, Guaranteed Delivery Method, etc).

Data model snapshotting/versioning approaches need to be found and compared next for undo and redo operations. Special synchronization cases and scenarios must be analyzed and solved. After conceptual solutions are found, technical feasibilities must be evaluated to support the theoretical aspects.

Contents

List of Figures

1 Introduction

Data is nowadays more distributed then ever. The main reason is the incredible spread of mobile devices in the past years. eMarketer expects 4.55 billion people worldwide to use a mobile phone in 2014. It is become more and more common that people own at least 3 network devices: a laptop/PC, mobile phone and tablet.

Ray Ozzie who held the positions of Chief Technical Officer and Chief Software Architect at Microsoft between 2005 and 2010 made the following statement first at the Technology Alliance Luncheon in Seattle:

> "So, moving forward, again I believe that the world some number of years from now in terms of how we consume IT is really shifting from a machine-centric viewpoint to what we refer to as three screens and a cloud: the phone, the PC, and the TV ultimately, and how we deliver value to them."

But what means exactly interacting with more than one device? The main purpose of software applications is data manipulation. Having many devices, means that the information first needs to be replicated to access it from each device. To replicate it, a persistent and failure safe information source is needed. In most network topologies this role is taken by the server. In Ray Ozzies' statement we have the cloud, that is essentially a network of servers. The most widespread network topology is Client-Server.

Looking back at our example, clients are the "three screens" (phone,PC,TV), but ultimately also all network devices on which a piece of software runs that relies on a server to perform some operations. The Server is in a simple case a network connected machine that responds to client request and has access to a database where all data is safely persisted. There are also more complex infrastructure architectures for the server part that add more guarantees, but we will not cover that here.

Interesting to analyze is in which places we have data. In the following picture the blue dots represent data: On the server part data resides in the database, in the servers business model, interaction layer and optionally in a server cache. On the client, data buckets can be found in its interaction layer, business model, presentation model and optionally in its client cache.

It can be observed that the data buckets size reduces from the server to the clients. In the database we have 100% of the data and in the clients presentation logic only 10% of it, which is currently needed in the user interface.

Looking at figure 1.1 it is clear that all the common data of all buckets need to be kept in sync. The challenge we face is to achieve that during system runtime when every client adds, deletes and modifies data. The data must be propagated fast between clients and data conflicts must be resolved automatically if possible. Receiving information such as: "Does somebody intend to modify this field? Is somebody currently

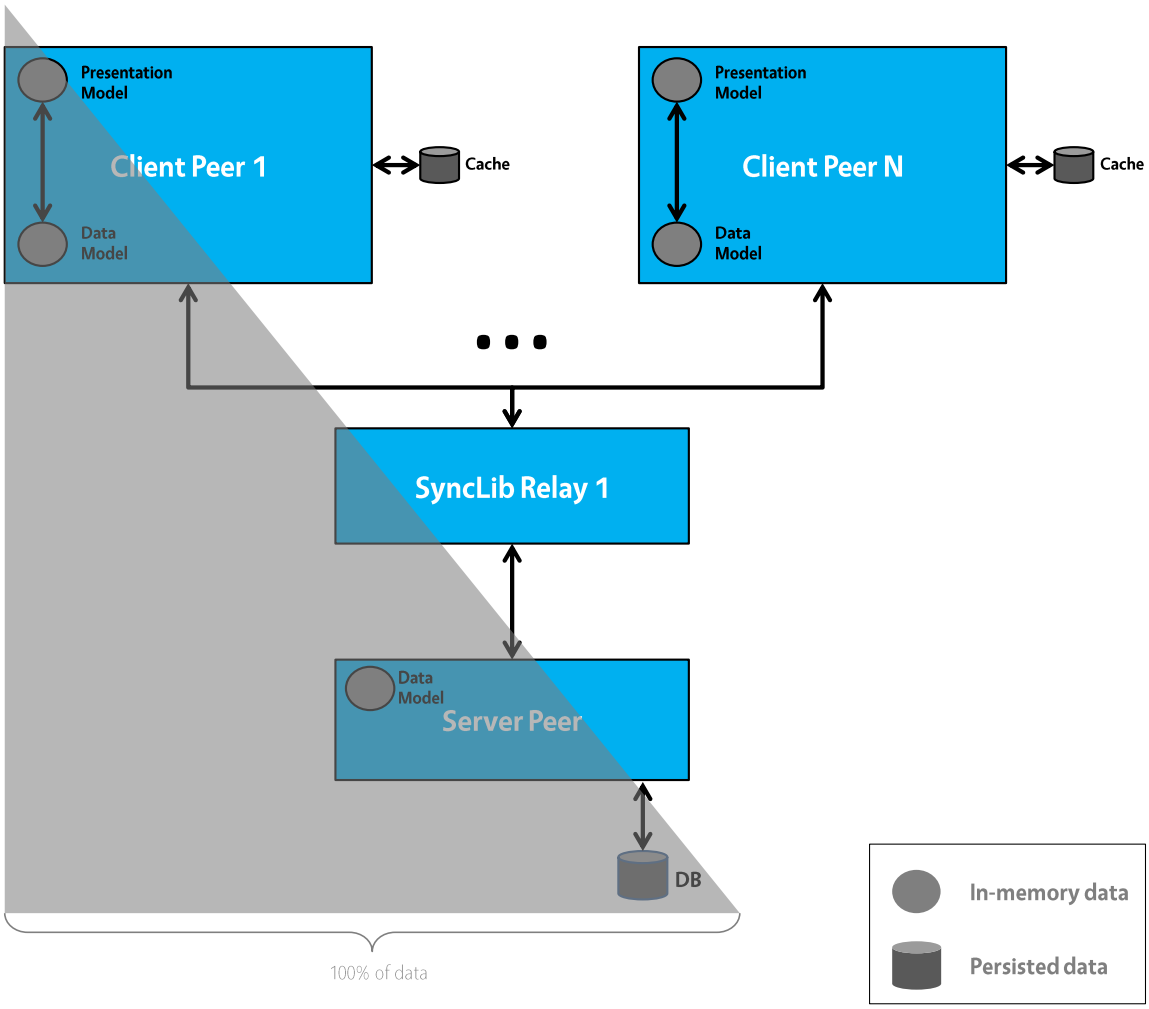


Figure 1.1: Data buckets system overview

modifying this field? Did the server receive my edits? Did all peers receive my edits?”, would definitely bring real-time client collaboration a step further.

Another aspect is that clients can go offline and online. This means that at every moment a client can disconnect from the network (connection loss, battery loss, etc.), but also new clients can join (new connection). Exciting and at the same time challenging is that clients should be able to edit their data also during the time they are offline, and when connection to the server is reestablished, synchronization and possible conflicts need to be resolved automatically.

2 Real-time Rich Client Communication

2.1 Motivation

The majority of web applications from the past years are implementing the Client-Server architecture. This means that a user interacts with the client side software, loads its data from the server and then operates on it. At the same time a lot of users can connect to the server and operate on their own data. What if many clients need to work with the same data set? This means that data on the server must be shared between users.

An example can be an accounting information system where many accountants have the right to access the sales list. Each of them can read and edit every item from the sales table. What happens if two or more accountants want to edit the same item? The classic and usual solution is to lock that item for the first user who accesses it. Once locked it can't be accessed by others. When the user finishes his work on that item the lock is freed and an other user can lock it again. In this way an item can be accessed just by one user at a given time.

We have this situation mainly in applications where collaboration plays a central part. In the example above collaboration between accountants is achieved, but with some downsides. Users can't really collaborate in real-time. Imagine that an accountant opens a sales item and leaves in a break or holiday without closing it. For that period of time nobody can edit that sales item. This is certainly undesired behavior. It would be great if two or more users could edit the same item at the same time, but with the locking solution this is not possible.

Our motivation is to push the current state a bit further and create new kind of web applications, where collaboration between users is brought to a different level. Imagine an user interface where users can edit data at the same time and receive value updates from other users in real-time. Similar solutions exist for text editing applications like Google Docs. Many users can write in the same text file at the same time and in case of conflicts (when many users edit the same text area), merging is made automatically. Additionally each user can see the cursor of all other collaborators.

The grand vision is to achieve this for business information systems, in order to make real-time collaboration between users possible.

2.2 Use cases in practice

Next, some use cases are presented to make the features of this new sort of web applications more clear.

2.2.1 Flight assistant

A big flight company ordered a web application for its flight assistants where they can manage from mobile devices traveler information and airplane logistics. The twist is that they must be able to operate even during flight, when they have no internet connection. Once they land and reestablish connection to the server, all the data must be synchronized between them and possible conflicts must be resolved. This is the most difficult step because the amount of data that is out of sync can be very large due to the long time being offline. Conflicts can arise if other users of the flight company edit the same data, while the flight assistant is in offline mode.

In this use case the alternation between online and offline mode with continuous data editing and synchronizing is the key concept.

2.2.2 Car sales vendor

A customer wants to order a car online and therefore he enters the companies web site. The ordering process requires to fill out a long form with all the car configurations and options desired. Having difficulties to fill out some fields, he calls a sales vendor for assistance. The assistant logs in on the clients data on the server and edits the problematic fields for him. The data typed in by the vendor is synchronized to the server and then from the server sent to the users device in the form of an update. In this way a sales vendor can collaborate in real-time with its customers taking advantage of the new, synchronization enabled web applications.

In addition to this, the customer can observe which fields of its form are selected or currently edited. If the vendor selects a field with the intention to edit it, the customers field turn yellow as a warning not to edit it. If the vendor types already in the field, then it turns red in the customers UI, signaling that editing it will very certainly cause a conflict. Moreover when the customer modifies a field a symbol appears next to it illustrating if the field value is in transfer, has been received by the server, has been received by some interested users, or by all of them.

In this use case the real-time data synchronization between clients is the core concept that elevates online collaboration.

2.3 Rich Client

When analyzing the software that runs on clients we can differentiate between thin clients, rich clients and fat clients. We defined the client in the introduction as a network connected device that runs a piece of software that relies on a server to perform some operations.

Thin clients are concerned just with the presentation of content that they receive from the server. They don’t do any computation, everything is handled by the server. A good example are the “click and load” web sites, where the browser renders just some masks received from the server. With every click in the user interface a new

mask may be rendered and therefore the browser needs to load it first and then it can display it.

An advantage of thin clients is that they need few hardware resources and no software deployment. The software is deployed centralized on the server and each client receives its user interface masks through the network. This option is good for devices in public places because they don't need expensive hardware and the software does not need to be deployed on all of them.

Fat Clients on the other hand hold everything from the UI mask to the domain data. Only the database is extern and the client needs a connection to it. They are called fat because beside the UI mask, widget logic and dialog logic they also include domain services and domain data. Fat clients are hard to maintain because they can't be deployed and updated remotely.

Rich Clients, called also Smart Clients, are somewhere in the middle. They contain the UI masks, widget logic and dialog logic, but the domain services and data reside on the server. Web applications that run in browsers are the most popular and widespread example of rich clients, especially since the boom of smartphones. Native mobile applications are very costly because they must be implemented for more platforms, therefore a lot of companies decide to implement their application as a webapp that is platform independent.

Rich clients have all the masks needed and all the logic to switch between them, so that the user experience is better than on thin clients because the masks must not be loaded every time from the server. Another big advantage is that with rich clients offline working mode is possible, because they have their own data model which can be manipulated independently from the server.

In the next chapters when we use the term "client" we will refer to rich clients.

2.4 Network Communication Protocols

In order to be able to make real-time collaborative web applications possible, a library is needed that handles all the data synchronization between peers. A custom protocol is also needed for exchanging synchronization specific messages across peers (see chapter 7).

Before designing such a protocol, we must first analyze what network communication protocols exist nowadays. The synchronization protocol will then run on top of it and therefore does not need to care about message delivery, package losses, message ordering and other low level network communication concerns.

Next, we will go chronologically through the main network communication protocols and present the concepts behind them.

2.4.1 Request/Response

This is the network communication model that sits at the foundation of the World Wide Web in the form of HTTP (Hypertext Transfer Protocol). Its premise is that all devices connected to a network are either clients or servers. Client devices usually contain a browser-enabled environment in which the user interface component of the application is running. Server devices are connected to the main databases and have the responsibility to handle incoming requests, process them and respond with a specific data set.

The communication flow between a client and a server has two parts: a request and a response. A communication cycle is always initiated by the client by sending a request to a server. The server then handles the request and sends back a response. In this model the server cannot send messages to a client without receiving a request first.

This is the main issue with the request/response model. There are many cases when it is desired to be able to send messages from the server to the client, without needing a request beforehand. This unidirectional communication burden has been more and more sensed and amplified with the transition to modern rich web applications. Such situations could be:

- The client wants to receive the temperature form the server, every time it exceeds 30 degrees.
- The client wants its stock ticker to be updated every time the price changes
- The client wants to receive continuously the real-time location of all airplanes in order to display them on a map.
- The client wants to be notified about other users actions (if they received a message, their cursor location, new messages, etc.)

In order to overcome these limitation of HTTP, two workarounds emerged that simulate a full-duplex communication between client and server. It is more a “hack” that wants to achieve something, for what HTTP was never intended. In other words HTTP is not the right tool for the job and therefore “short polling” and “long polling” appear unnatural.

2.4.2 Short Polling

Short polling uses Ajax and the idea behind it is to send at a specific interval a request to the server. When the server receives a request it responds if it has new data. For example the client sends a request every 200ms and the server responds the first 10 reqests with a “NO” because nothing has changed and the 11th request with a “YES” if it has new data available. Figure 2.1 illustrates this. This trick seems to solve the problem, but it comes with some disadvantages. The client still needs to initiate the connection first and if the server has new data right after it sent a response it still needs to wait for a next request from the client and therefore that “real-time” feeling is not achieved. If the polling interval is very short (under 100ms) then this can be solved, but on the other hand the server is now bombarded with requests and it can

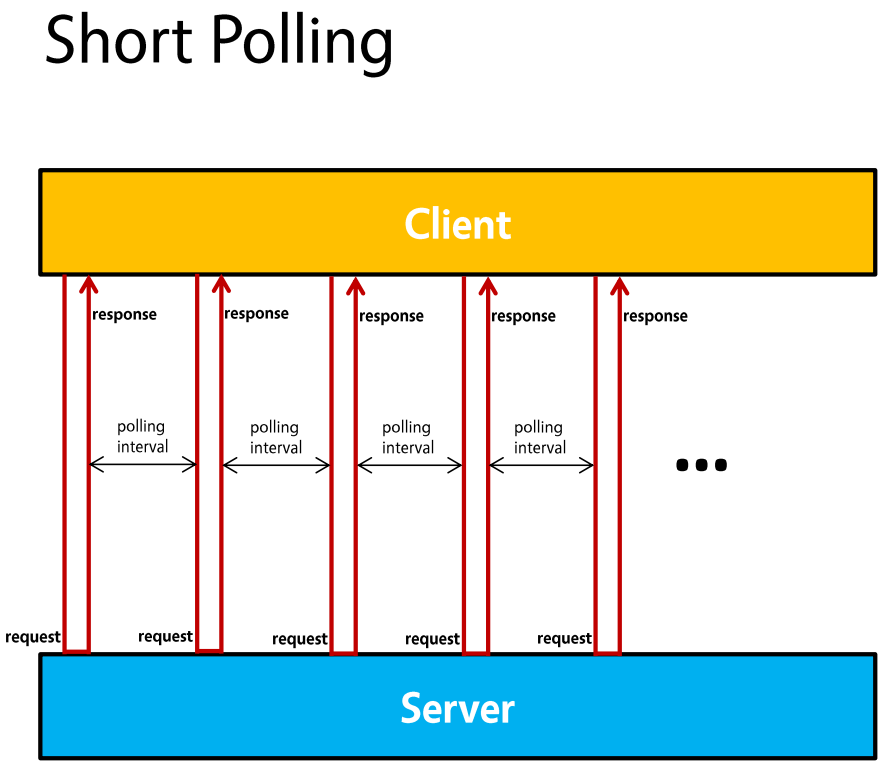


Figure 2.1: Short polling

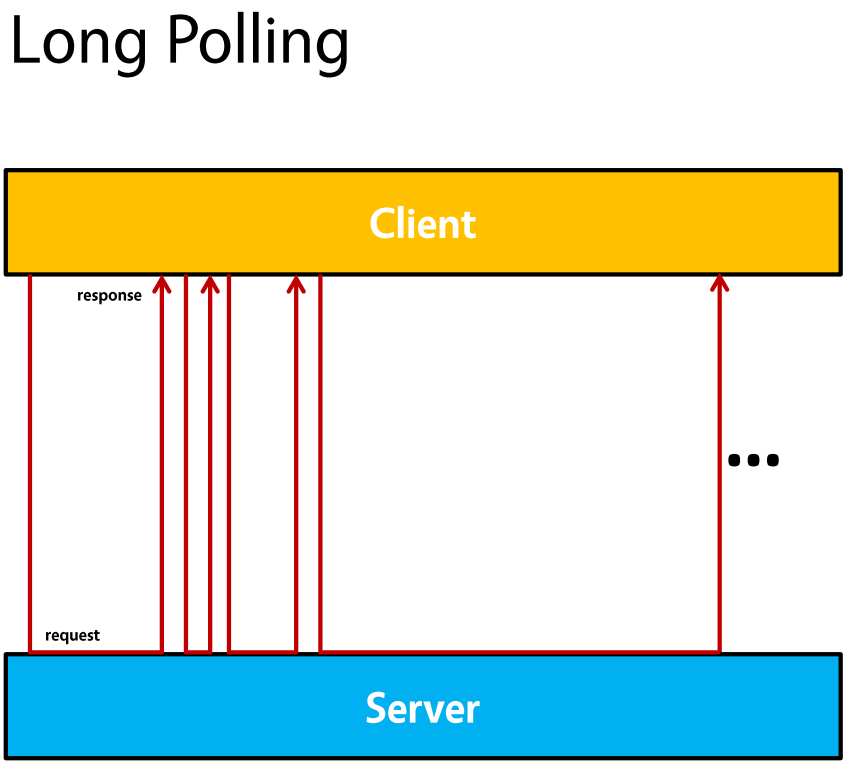


Figure 2.2: Long polling

affect its availability.

These inconvenience has been solved by introducing the long polling concept.

2.4.3 Long Polling

The improvement of long polling relies in responding to a request only when new data is available. After this the client immediately opens up a “long polling” connection to the server and waits for a further response. So basically the server does not replay until it has new data. The client has the false impression that the request is just taking a while to complete. Schematically it can be visualized in figure 2.2.

The good part is that the server is not bombarded anymore with requests and the server can send anytime data to the clients because a connection is always open. However, in this situation the server will have to manage a lot of open connections and each of them requires some resources. So it is possible to reach the maximum number of connections a server can manage, and from that point no other incoming connection will be accepted.

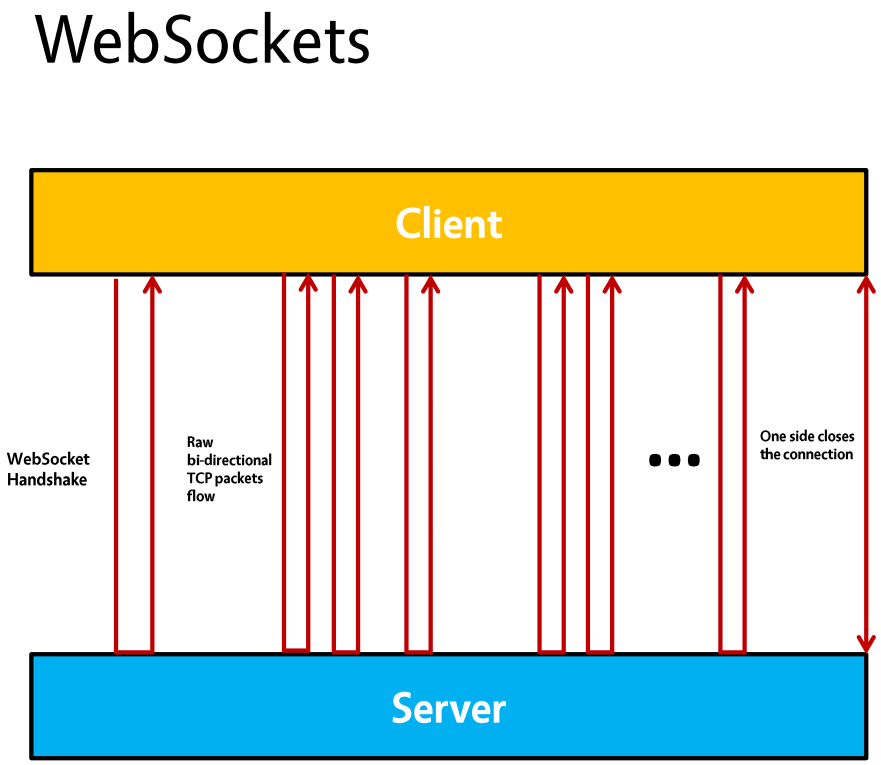


Figure 2.3: WebSockets

2.4.4 WebSockets

To overcome all this difficulties the WebSocket protocol has been standardized in 2001 by IETF (Internet Engineering Task Force) as [2]. WebSockets enable bidirectional TCP communication between client and server facilitating real-time communication. The connection is persistent, which means that once opened, any number of messages can be exchanged without requiring a new connection. Before establishing a connection the client needs to send a WebSocket handshake request to the server in order to receive afterwards a WebSocket handshake response with details on how to open the TCP connection.

Using WebSockets, the server can send immediately notifications to the client when new data is available, without needing to wait first for a request. In this way the threshold of tolerable delay in client/server interaction is substantially lowered. Since 2014 all browsers, except Opera Mini, support the WebSocket protocol. Figure 2.3 illustrates the communication flow.

2.5 WebRTC

WebRTC (Web Realtime Communications) is designed to enable browser-to-browser audio calling, video chat and data sharing without requiring any internal or external plugins. It was stared in May 2011 by Google as an open source project for browser-to-browser real-time communication (RTC), and a standardized implementation is now available in Google Chrome, Mozilla Firefox and Opera. It is part of the HTML5 specification managed by the IETF and W3C.

Popular web services like Skype, Google Hangouts and Facebook use RTC, but need additional plugins or native apps that can require licensing and can be error prone, complex, difficult to maintain and a burden for the user.

The revolutionary idea of WebRTC is to build standardized APIs for RTC directly into the web browsers. This results into the highest performance and the lowest latency possible. If a client wants to share data with another client, traditionally it needs to send it first to a server, which then forwards it to the recipient. With WebRTC data is transmitted directly from client to client using the WebRTC Data Channel. Brendan Eich, Mozilla CTO and inventor of JavaScript, alleges that:

> "WebRTC is a new front in the long war for an open and unencumbered web"

In consequence, a more and more obvious orientation towards real-time and collaborative web applications can be observed in the past 3 years. The main driving force behind this was the need for modern web applications that are able to consume more data than ever before and exchange it rapidly.

Therefore, looking back we can observe that the motivations behind this master thesis blend in perfectly with the actual context and future vision of web applications.

2.6 Synchronization

Synchronization is the key concept in this thesis, because the main objective is to design a library that observes the data changes and distributes them across the network to the other peers in order to reach a consistent global state as fast as possible. Before presenting the different aspects of the Synchronization Library, the term synchronization should be clarified.

In every software application data is being manipulated. This can be database records, file system informations or simple variables from the main memory. As long as there is just one process that modifies the data, synchronization has no relevance. If more processes can modify the same data set, a scheduling mechanism must be introduced in order to serialize the changes. This is called process synchronization and has the role to avoid the case that two processes modify the same data item at the same time.

It works for applications that run on a single device. Large business applications are usually deployed on different machines that communicate through a network. In this case the data items could be replicated across all network peers and could be

changes separately. So process synchronization has no effect if the same data item is modified simultaneously at physically distributed peers. The mechanism that keeps data consistent across a network is called data synchronization. In the next chapters we will refer to data synchronization by presenting a solution that could lay the base of real-time, collaborative business information systems.

3 SyncLib Data Meta Model

In order to present all the concepts behind the Synchronization Library (SyncLib) one of the best methods is to visualize them in a meta model diagram. The advantages of a diagram are that all the key information fit on one page, similar concepts can pe grouped with swim-lanes together, and all the relationships between them can be identified.

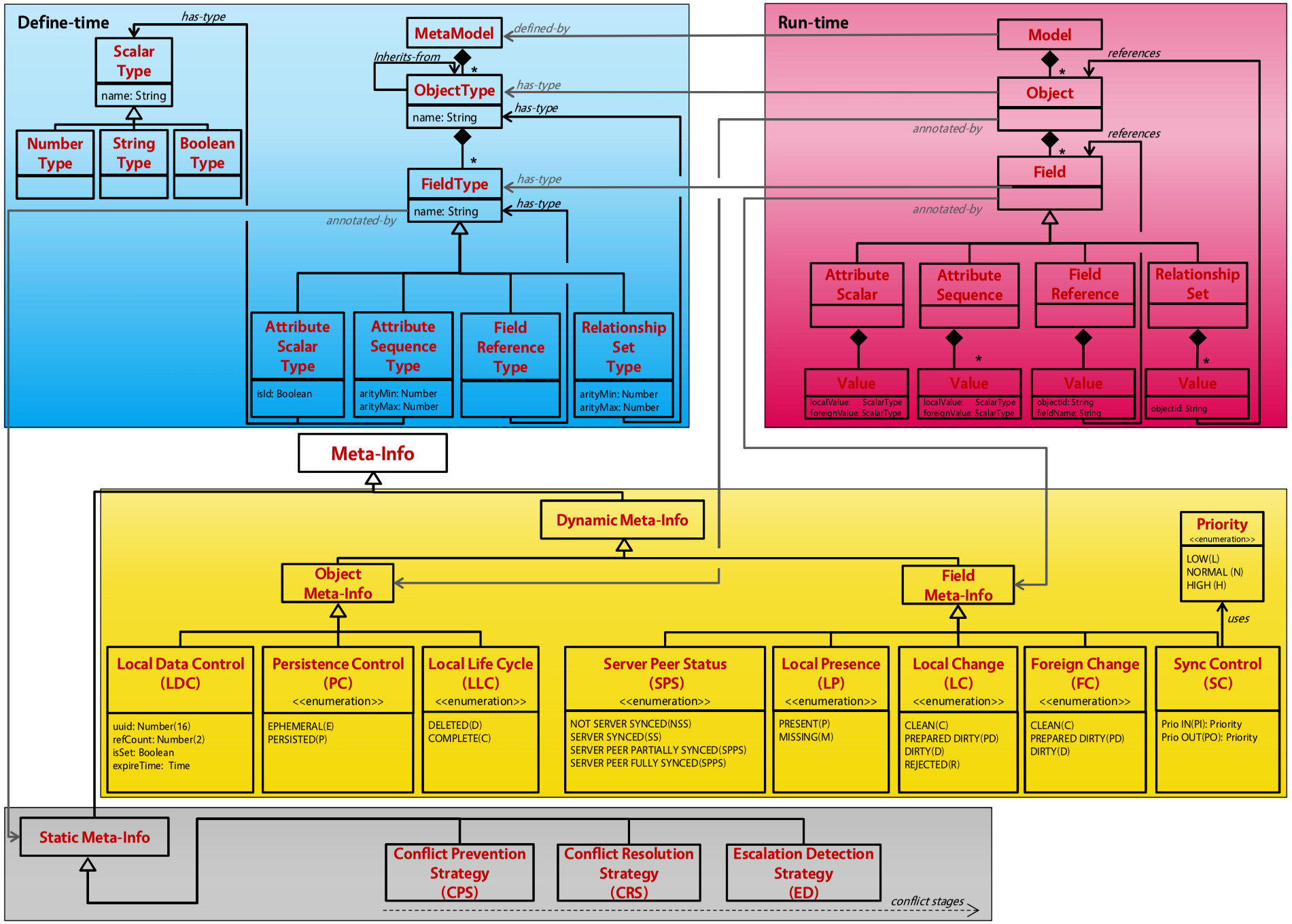
On the first page the actual diagram is presented, where UML (Unified Modeling Language) is used as a way to visualize the design of the library. The second figure represents the meta model taxonomy where each concept, present in the diagram, is explained briefly.

3.1 Data Components

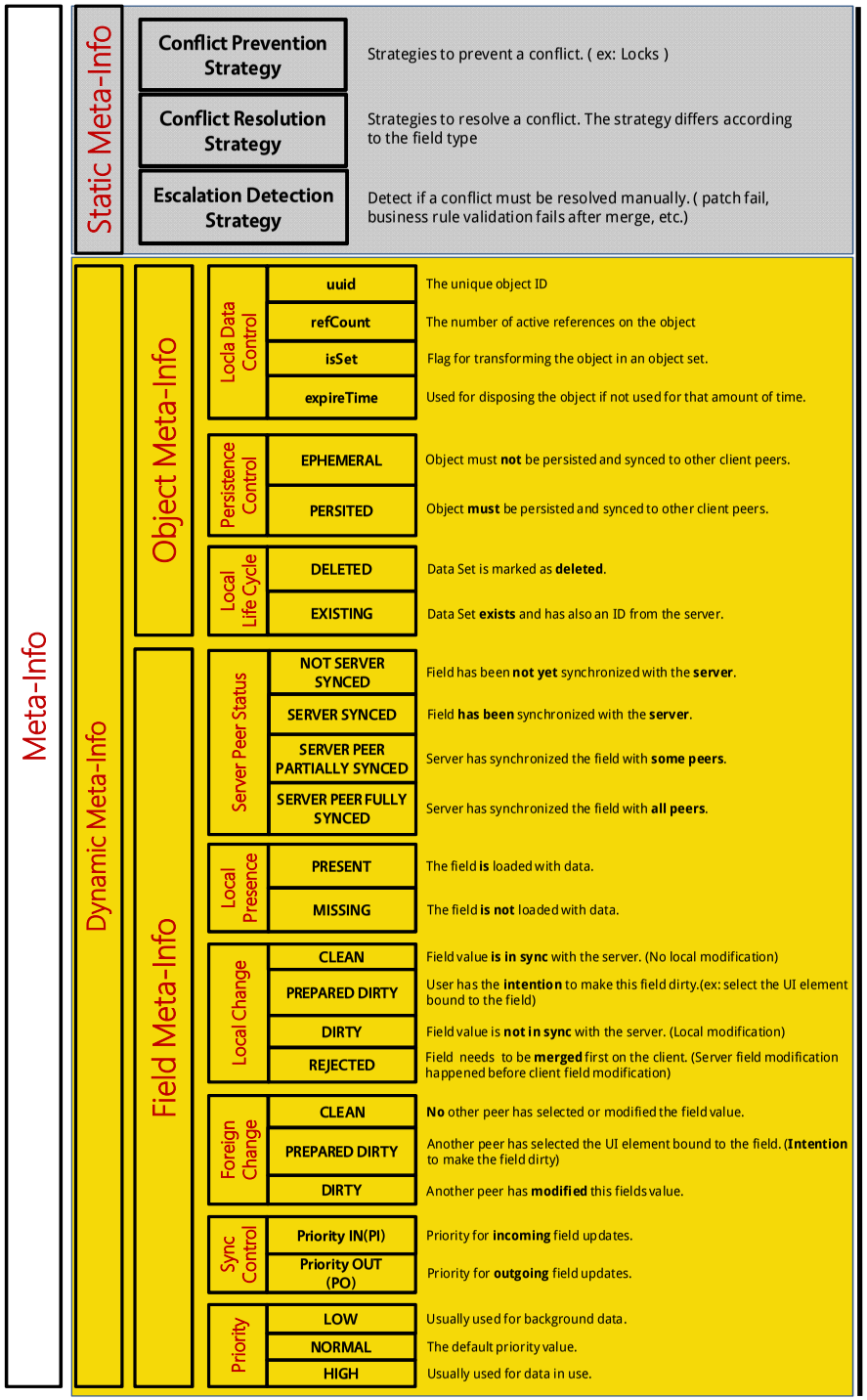
The diagram is named “Data Meta Model”, because its main purpose is to present all the components that are concerned with data. This is the core part of the library, because the most important aspects of synchronization are the business data, the definition of necessary data structures, and meta information that is important for signaling different events between peers.

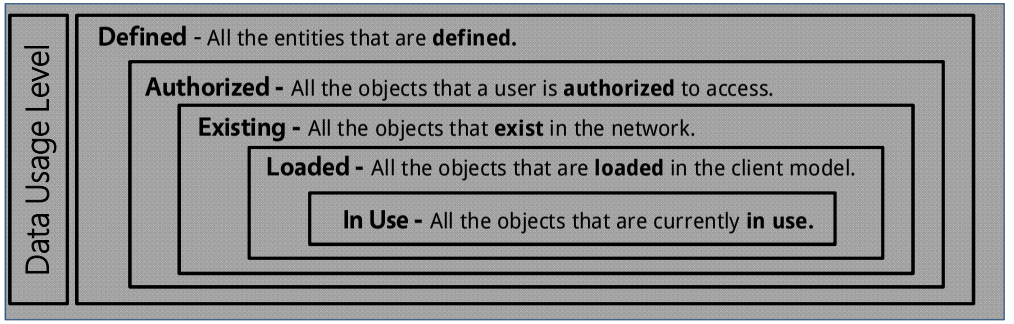
The different colored swim lanes, group data components with similar roles together. Everything in the blue box describes how the type system is defined. In other words, the blue components hold the structure for the actual business data. The red box components are runtime instances of the already defined object types. A close analogy is the Class - Object semantic in OOP (Object Oriented Programming). A class holds the definition, and the objects are instances of that definition.

SyncLib Data **MetaModel**



SyncLib Data MetaModel **Taxonomy**





Scalar Type	The abstract type for scalar entities.
Number Type	Type of fields that hold a numeric value
String Type	Type of fields that hold a string value
Boolean Type	Type of fields that hold a boolean value
Data Model	Represents **all** the data structures defined at compile time
EntityType	Represents **a** data structure. (entity name, field names and field types)
FieldType	The abstract type of an entity field.
Attribute Scalar Type	The Attribute Scalar Type can be a Number Type, a String Type or a Boolean Type. It can hold business data or an object ID.
Attribute Sequence Type	The Attribute Sequence Type represents an array of Number Types, String Types or a Boolean Types. It has defined a minimum and a maximum number of objects.
Field Reference Type	A Field Reference Type represents a reference to another object field. It is useful for defining entities that have the role to wrap fields from many objects together in the same object without duplicating the value (Field Set).
Relationship Set Type	A Relationship Set Type represents a set of references to other objects. That is why it has as type the Entity Type. It is useful to model relationships to other objects. (One organisation – Many Employees)

Object Graph	Contains **all** instances of the entities at runtime and the relationships between them.
Object	Represents **a** concrete instance of an entity at runtime.
Field	Is the concrete instantiation of a Field Type at runtime and contains at least one value assigned at runtime and is nested in an object.
Value localValue: ScalarType foreignValue: ScalarType	If a field is an Attribute Scalar or Attribute Sequence, it holds one or respectively many values. A value is composed of a local value for data introduced by the user and a foreign value for data received from other peers.
Value id: String	In case a field is a field reference or a relationship set, its value are object or field reference IDs.
Attribute Scalar	Is a concrete field of the Attribute Scalar Type
Attribute Sequence name: String	Is a concrete field of the Attribute Sequence Type. Because it is a sequence it contains one or many values.
Field Reference name: String	Is a concrete field of the Field Reference Type that contains at least "arityMin" and at most "arityMax" field references.
Relationship Set name: String arityMin: Number arityMax: Number	Is a concrete field of the Relationship Type that contains at least "arityMin" and at most "arityMax" object references.

3.1.1 Meta Model. Model

The **meta model** contains all defined object types with all the relationships between them. It is usually represented visually in projects, and named Software Architecture Information Viewpoint.

The **model** is the implementation of the meta model, and is dynamic during runtime, because new objects can be created, and existing ones can be disposed. An analogy can be made between the meta model and the schema definition of a database. The model would then represent, in the database context, all the concrete table data.

3.1.2 Object Type

Object Types are the basic modeling units that compose the meta model. They are inferred from the business use cases. A name must be specified to define an object type, and for each field a field type and a field name.

```
1 SyncLib.define("OrganisationUnit", {
2     id:   "string",
3     name: "string"
4 });
```

Listing 3.1: Define an object type

3.1.3 Object

Once an object type is defined, we can create many objects from it as follows:

```
1 var xtOrgUnitObject = SyncLib.create("OrganisationUnit",
2                       {
3                           id:   "xt",
4                           name: "msg Applied Technology Research"
5                       });
```

Listing 3.2: Create an object from object type

The object is the central unit of the SyncLib. All the synchronization operations are made on objects. In order to manage all the active objects in a system a reference counter for each object is needed.

The reference counter is incremented every time the object is referenced, and decremented when a reference on it is released. If no references exist on an object, it will be disposed.

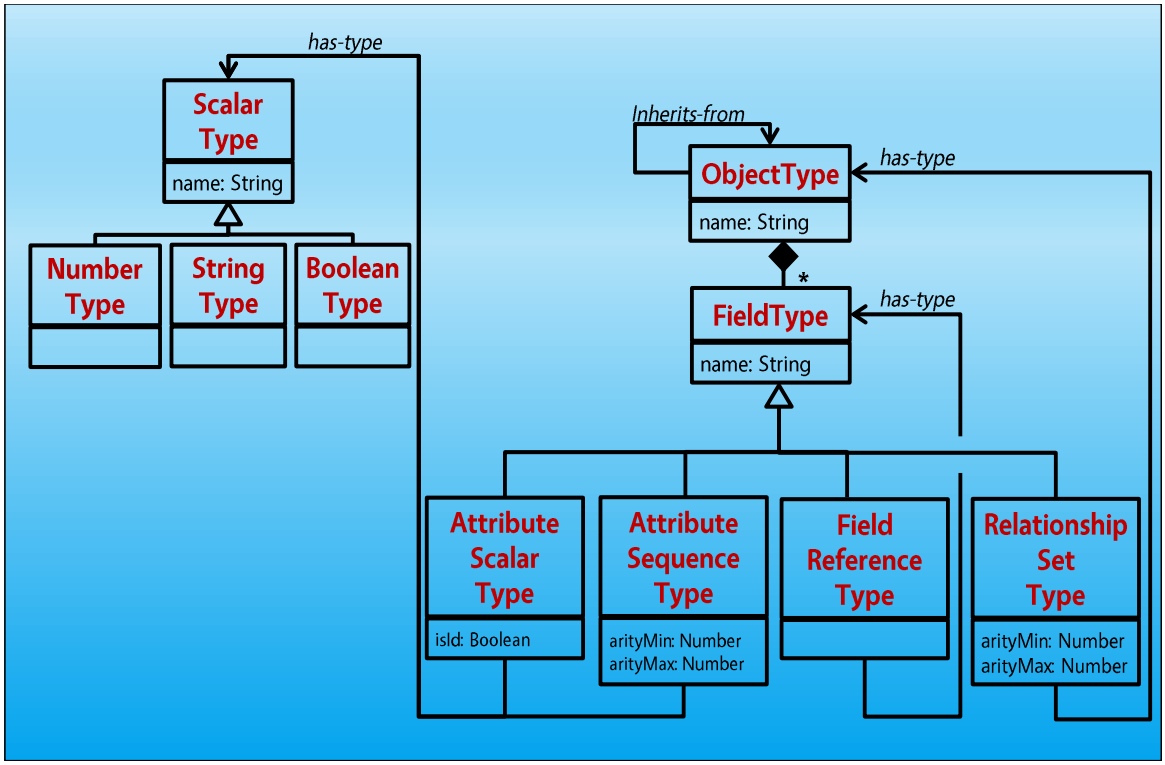


Figure 3.1: Definition View

3.1.4 Field Types

The **field types** are contained by an object type, and must have defined a name, and a type. A field type is, as we can see in the diagram, derived from four concrete types: attribute scalar type, attribute sequence type, field reference type, and relationship set type.

Attribute Scalar Type

The attribute scalar type refers to fields that hold a single value of a primitive type (number, string or boolean). This is why the attribute scalar type has as type, the abstract **scalar type**. It can be a **number type**, **string type** or **boolean type**. An attribute scalar type field can sometimes hold an id, necessary for the SyncLib, fact modeled by the boolean “id” attribute.

Attribute Sequence Type

The attribute sequence type is needed, whenever an array or set of values are required. The minimum and maximum number of elements can be specified using the “arityMin”

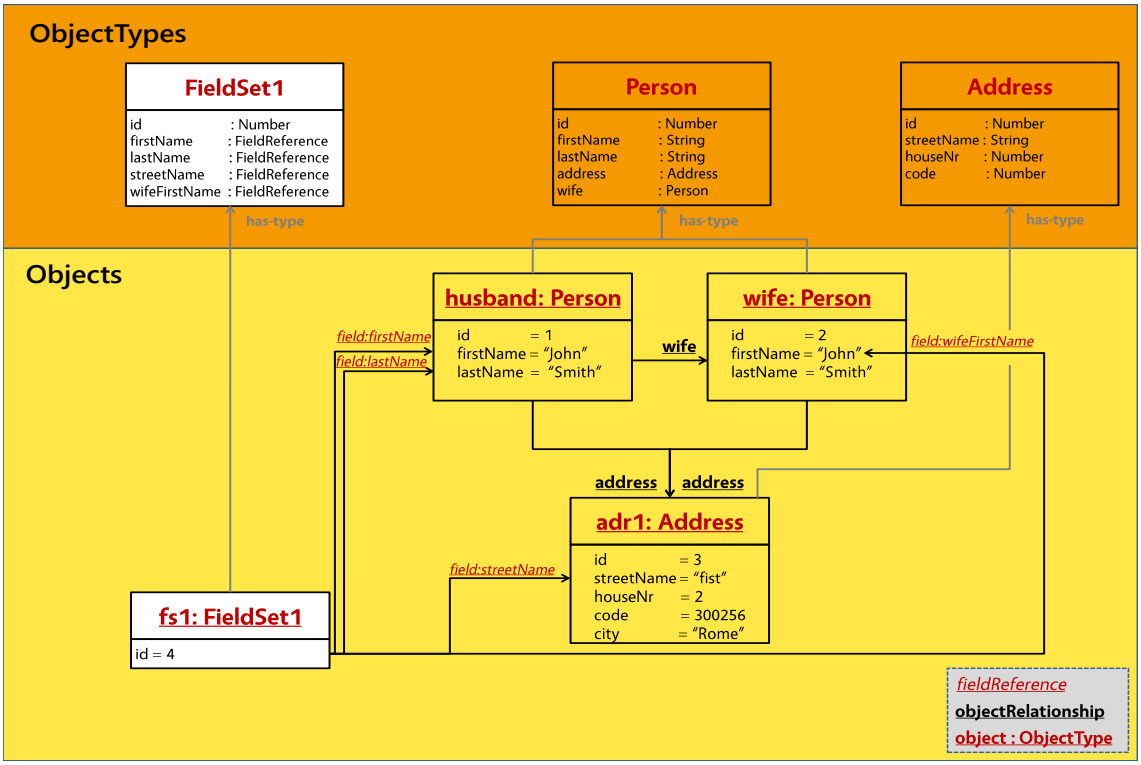


Figure 3.2: Field Reference example

and "arityMax" attributes. Also, the attribute sequence has scalar types, meaning that the array or set values can be numbers, strings or booleans.

Field Reference Type

The field reference type is used for holding instances to other fields. The motivation for this is the case when a set of fields from different objects need to be grouped together for synchronization. To achieve this, an object is needed, that holds references to all the fields. In this case, that object represents the concept of a "field set". It has a field reference type defined for each field.

In figure 3.2 "fs1" is an object created from the "FieldSet1" object type. It has an id, and 4 field references: "field:firstName", "field:lastName", "field:streetName", and "field:wifeFirstName". Conceptually, this object represents a field set, because it groups different fields together. When the object is synchronized, then all fields will be transmitted together.

The main advantage of the field reference is that values do not need to be duplicated in order to group them together. The field reference is conceptually pointing directly to a field, but technically it is composed of an "objectId" and a "fieldName". Theoretically, the field references are a "fancy" feature, but in practice it is most probable to use

object sets (see 5.3) instead, due to the technical complexity.

Relationship Set Type

The relationship set type has the role to model the one-to-many relationships between objects. If an object has many references to other objects of the same type, a relationship set can be formed to group them together. Arities can be defined, in order to specify more exactly the set size.

Listings 3.3 - 3.7 illustrate different possibilities to define set size constraints.

```
em.define("Person", {
    id:        "String",
    firstName: "String",
    orgUnits:  "OrgUnit*"
});
```

Listing 3.3: Relationship Set with arity “any”

```
em.define("Person", {
    id:        "String",
    firstName: "String",
    orgUnits:  "OrgUnit+"
});
```

Listing 3.4: Relationship Set with arity “at least one”

```
em.define("Person", {
    id:        "String",
    firstName: "String",
    orgUnits:  "OrgUnit{3,}"
});
```

Listing 3.5: Relationship Set with arity “at least 3”

```
em.define("Person", {
    id:        "String",
    firstName: "String",
    orgUnits:  "OrgUnit{0,11}"
});
```

Listing 3.6: Relationship Set with arity “at most 11”

```
em.define("Person", {
    id:        "String",
    firstName: "String",
    orgUnits:  "OrgUnit{3,11}"
});
```

Listing 3.7: Relationship Set with arity “at least 3 and at most 11”

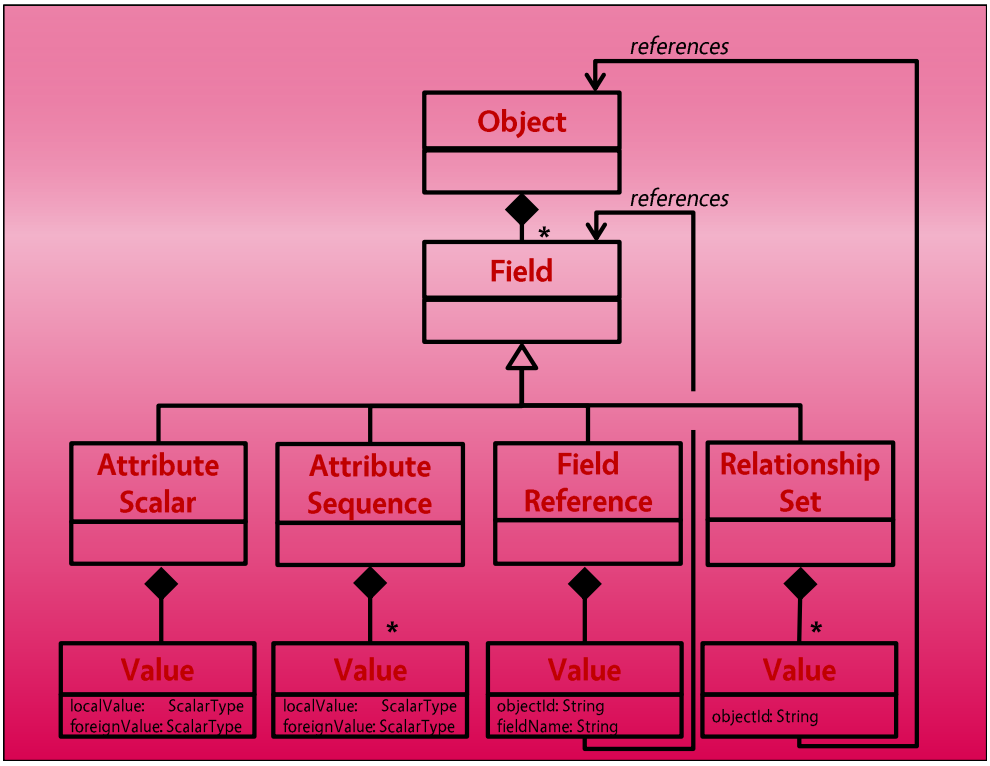


Figure 3.3: Run-time View

3.1.5 Field

The field is the actual instance of a field type at runtime. In the field type the name and type are defined, and the field adds, the value corresponding to the specified type. As we can see in figure 3.3, symmetrically to the previously presented definition view, the field component is derived from attribute scalar, attribute sequence, field reference and relationship set, which are the instances of their corresponding types.

By looking at these four field categories, we can notice that attribute sequence and relationship set can contain none or many values. This results from the fact that they are collections that have arities defined in their type definition.

Another aspect to note is that two types of values exist: some that have a “localValue” and a “foreignValue”, and others that have just an id. Attribute scalar and sequence fields hold real data. Their values can have two sources: local user and other peers. Therefore two value attributes are needed. The “localValue” is updated by changes made by the local client through the UI, and the “foreignValue” is used for data updates received over the network from other peers. The “foreignValue” is merged at specific times with the “localValue”. More about this process is described in section 6.6.

Field reference and relationship set does not hold real data. Instead they hold references to other fields or respectively objects. Therefore, they got just an id attribute for the references.

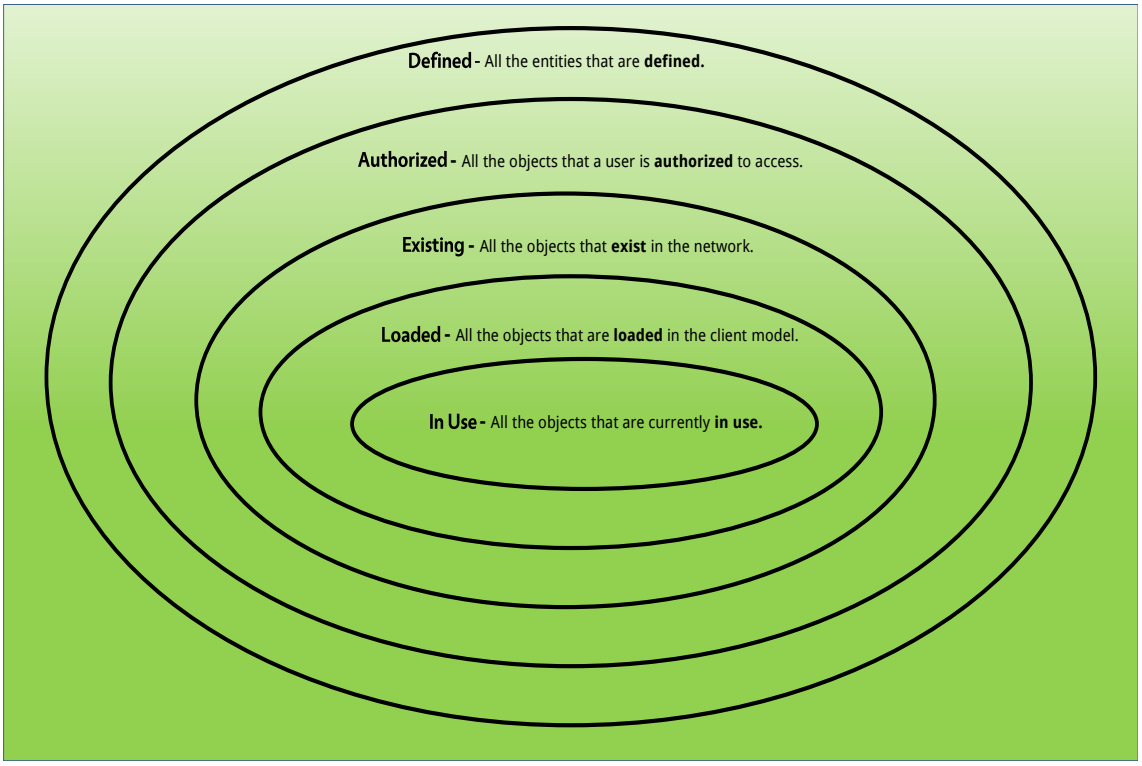


Figure 3.4: Data usage levels

3.1.6 Data Usage Levels

Synchronizing data between a client and a server implies to identify what part of the whole model needs to be synchronized. Therefore separate levels were identified that, group data together based on different aspects. The data usage levels are hierarchical as the Venn Diagram in figure 3.4 illustrates.

DEFINED
The meta model is defined both on the client and the server. It contains all the necessary object types for the application, and holds no data, just the definitions.

AUTHORIZED
For security reasons, a specific user can be authorized just on a part of the meta model. Authorization rules can be applied on object types or on objects. A specific user must receive, through synchronization, only the data sets for which it has authorization. This level restricts the possible objects a peer can possess, therefore it's a part of the defined level.

EXISTING
Data can exist embedded in objects on the server peer, but is not yet loaded on the client peer. Synchronization is necessary only for data that exists on both server and client peers. Objects existing just on the server must be included in the synchronization process just after the client loaded them. The existing level

contains all the concrete models. This means all the data present in the whole system. For data synchronization it's too general, because nothing is said about what data each peer possesses, and what they have in common. For example a peer can be authorized to load objects of 20 types, but currently it has created or loaded just objects of 5 object types.

LOADED
A peer has, usually, loaded just a small part of all the existing data. The loaded part needs to be synchronized because it exists on both client and server. Considering the loading process, a data pre-fetching feature can boost the overall performance. For step-by-step user interfaces, it would be of great benefit to be able to tell, to SyncLib, what data needs to be preloaded for the next view. The difference to the existing level is that the scope is reduced to a specific peer.

IN USE
At this stage the user interface (UI) needs to be taken into consideration. It is improbable that all the loaded data is displayed in the UI at once. This means that we can differentiate between data that is currently displayed in the UI and background data that is loaded, but not yet visible.

The importance of this insight is that it can be useful to prioritize data for synchronization. A lower priority can be chosen for the background data, and a higher one for the data currently in use. All the edits made by an user must be replicated in real-time on all other peers. Very fast synchronization of the data currently in use reduces the conflict occurrence probability. It must be also noticed that the data set in use may go into the background after every user interaction.

The most important aspect is to identify the data in use and synchronize it rapidly across the other clients, in order to offer a real-time feeling to the end users, like for example, in a chat application.

3.2 Meta-Information

For the synchronization process, meta-information (meta-info) plays a central role. It covers aspects like: what are the strategies to use for conflict resolution, data modifications, object life cycle, information about the synchronization progress and priority.

A first differentiation looking at the SyncLib Data MetaModel diagram is made between static meta-info and dynamic meta-info.

3.2.1 Static Meta-Info

Static meta-info targets the definition part of the MetaModel, more precisely the field type. It represents rules or suggestions for the SyncLib that are defined once and do not change during the runtime of the application. This means that at the time an object type is defined, its field types can be annotated with static meta-info.

In figure 3.5 a graphical representation can be seen that shows every stage from the

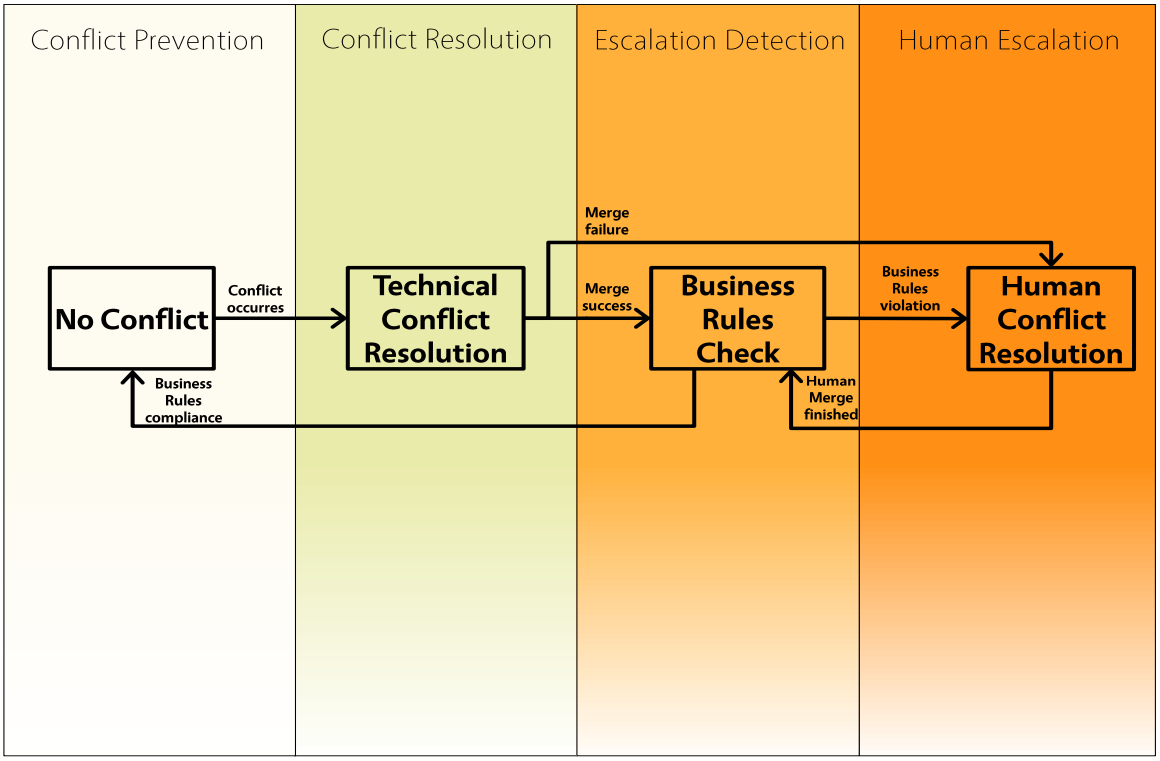


Figure 3.5: Field Reference example

conflict free situation to the point where the human user must intervene.

Conflict Prevention Strategies

Conflict Prevention Strategies describe different methods that reduce the probability of a conflict to occur. One explicit method to achieve this is data locking. Different locks can be used:

Hard Lock
Don't let client A edit a field if other peer is currently editing it.

Soft Lock
Let client A edit a field, but show a warning every time it is going to, or is currently changed by another peer. This concept is implemented using the "Foreign Change" dynamic meta-information. Every field has by default at the beginning a soft lock.

Break Lock
The possibility to unlock a field if it has a hard lock.

Conflict Resolution Strategies

Conflict resolution strategies refer to algorithms used for resolving data conflicts. These algorithm will be presented in detail in a separate chapter.

Escalation Detection

There are cases when the technical conflict resolution fails and a human is needed to resolve the conflict. This means that the human cannot be excluded from the synchronization process. This is the reason why conflict resolution is not executed on the server, but instead on the client.

The escalation detection mechanism checks if a human must intervene in case of a conflict. We identified two such situations:

- **Technical merge patch fails**: In a classic synchronization algorithm when a client modifies locally data, a difference is made between the old and new value. That difference (called patch) is then sent over the network to the other client peers to apply it. If one of the other peers modifies the same piece of data at the same time, most probably the patch received over the network can not be applied and a synchronization conflict occurs that cannot be resolved automatically. In this case the human user must decide between the two data versions or enter a third one.
- **Business Logic validation fails**: It is possible that when two clients edit the same piece of data, both modifications can be merged successfully. Even if no conflict occurred, the merge result can be invalid from the perspective of the application business rules. Let's see the following example:

  1. A text field contains the string: "example" and can have at **most** one digit in it.
  2. Peer A changes the text field into: "1example"
  3. Peer B edits the text field at the same time to: "example2"
  4. The synchronization process merges the two modifications having as a result: "1example2"
  5. Both peers see now in their text field the string "1example2"

  The technical merge was successful, but the result is not valid against the business rules. This is the second case when the human user must resolve the merge manually.

  In conclusion, the SyncLib must escalate the merging to the human user if it can't resolve a conflict automatically, or if the merge was resolved automatically, but it violates the business rules.

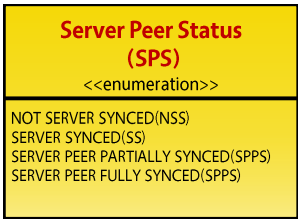


Figure 3.6: Server Peer Status

3.2.2 Dynamic Meta-Info

Dynamic meta-info is exchanged between application peers during runtime. It covers information about the synchronization process. The difference to static meta-info is that the values change while the application runs. The triggers for the changes are mainly user actions. It is also important to notice that the dynamic meta-infos are bound to a field. This means that for each field the SyncLib needs to store data structures for them.

The Data Meta Model diagram presents how meta-infos are persisted. This view differs from how the meta-infos are used programatically, due the possibilities to aggregate them up to the object level. So dynamic meta-info is persisted just for fields and if it's to granular it can be used at object level, but without persistence.

Next, all the dynamic meta-infos are presented along with their aggregation logic at object level for field meta-information.

Server Peer Status

The "Server Peer Status" is a field meta-information that reflects the progress of the synchronization. This is of great interest to the user. When he modifies a text field he will know if his changes are under transmission, have been received by the server or have been already received by the other peers. (Figure 3.6)

NOT SERVER SYNCED
When a field is in the *Not Server Synced* state, it means that it was locally modified and is out of sync with the server. Every time the user changes the value of a field its "Server Peer Status" switches to *Not Server Synced*, fact that can be also visually displayed, letting the user know that his edit is being transmitted to the server. If the client crashes during *Not Server Synced*, it is most probable that his changes will get lost if client side caching is not enabled.

Object level aggregation: Has no object level semantic.

SERVER SYNCED
Server Synced state means that the server got a new field value from the client. Before sending a *Server Synced* notification to the client, the server peer updates its local field value and persists it. This means that when the client peer is in the *Server Synced* state, it can be sure that his changes won't get lost, because the server received and persisted them.

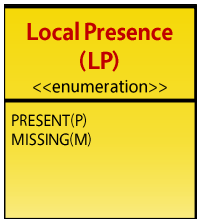


Figure 3.7: Local Presence

Object level aggregation: Has no object level semantic.

SERVER PEER PARTIALLY SYNCED

After the server peer receives a new value from a client peer, it will broadcast it to all interested peers (see 4.3.2). Each peer will then send the server an acknowledge, after receiving the field update. After the server gets the first acknowledge, it signals the client peer to switch the "Server Peer Status" to *Server Peer Partially Synced* state. This informs the user that some peers got his change, but not all of them.

Object level aggregation: Has no object level semantic.

SERVER PEER FULLY SYNCED

After the server peer receives an acknowledge from every peer, it notifies the client peer that all interested peers got his change. Once notified, the "Server Peer Status" on the client switches to *Server Peer Fully Synced*. The user now knows that all interested peers are now in sync with him. This state can be considered the default and initial state, when all peers hold the same data.

Object level aggregation: Has no object level semantic.

Local Presence

The client peer has the possibility to completely load an object from the server, or just specific fields. This means that fully and partially loaded objects can exist. This information is stored using the "Local Presence" field meta-information. (Figure 3.7)

PRESENT

If the "Local Presence" meta-info of a field is in *Present* state, it means that the field value has been loaded from the server.
Object level aggregation: If all fields are fully loaded, then we say that the whole object is fully loaded.

MISSING

The *Missing* state specifies that the field holds no value yet.
Object level aggregation: If at least one field is in *Missing* state, then the object is partially loaded.

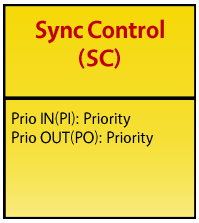


Figure 3.8: Sync Control

Sync Control

As stated in the *In Use* data usage level, any given time a part of the UI is in foreground and the rest in background. Obviously the data currently in use must be synchronized with a higher priority than the background data. That is what "Sync Control" stands for. "Sync Control" is a field meta-information that gives the developer the possibility to set *IN* and *OUT* synchronization priorities for fields. (Figure 3.8)

PRIORITY IN
: *Priority IN* targets the data that is sent from the server to the client. The higher the *Priority IN* of a field is, that faster it will receive updates from the server for that specific field. Update messages with higher priority will overtake those with lower one, being placed at the beginning of a message priority queue.

PRIORITY OUT
: *Priority OUT* targets the data that is sent from the client to the server. The higher *Priority OUT* of a field is, that faster edits of that field will be sent to the server. *Priority IN/OUT* are enumerations and can therefore have three different values: low, normal and high. The default priority value for a field is normal.

Object level aggregation: The SyncLib API provides setting priorities at object level. When the whole object priority is set then all its fields will be assigned to that priority.

Local Change

The modification state of the fields needs to be tracked, in order to know what synchronization actions to perform. "Local Change" is a field meta-information and has as main purpose to observe the user actions and trigger corresponding functionalities in the SyncLib. (Figure 3.9)

CLEAN
: The default "Local Change" state of a field is *Clean*. This means that the client peer field value is in sync with that on the server peer and the user do not even intend to change it.

  Object level aggregation: If all fields are *Clean*, then the whole object is considered *Clean*.

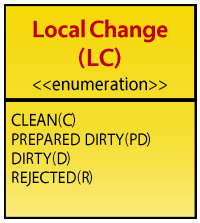


Figure 3.9: Local Change

PREPARED DIRTY

Prepared Dirty signals that the user intends to change the value of the field. This intention can be caught at the UI elements level. For example the user clicks in a text box, or swipes over a slider, or clicks a combo box to open it, etc.

The field value is not yet modified, it is prepared to be modified. This state has importance in conflict prevention, because the "Local Change" will be transmitted to all peers interested for that specific field. Now imagine an UI, where a client is notified when others are intending to modify, or even edit fields. This information is transmitted faster than the actual data and conflicts can be prevented. Users will probably not modify a field, that has a warning that somebody else is going to modify it, or has changed it already. In some UIs the users could switch between fields very fast, therefore a short delay is needed between the UI element selection and the "Local Change" state transition *Prepared Dirty*.

Object level aggregation: If at least a field is in *Prepared Dirty* state and none in *Dirty* state, then the whole object is considered *Prepared Dirty*.

DIRTY

After a field has been changed, the "Local Change" state switches to *Dirty*. This means that the fields local value is not equal with the server peer value. When this state is active the synchronization process starts.

Object level aggregation: If a field is in *Dirty* state, then the whole object is considered *Dirty*.

REJECTED

A *Dirty* status for a "Local Change" of a field implies that the client's new value was sent to the server by SyncLib. In the optimistic case, the server overwrites his field value with the received one and notifies afterwards all other interested client peers. Unfortunately, there exists a scenario where a conflict may occur. If two clients(A and B) modify the same field, then both fields "Local Change" state switches to *Dirty* and both will send their new value to the server peer.

Lets assume that the server peer receives the value from client A first and overwrites his field value with it. When the field value from the client B arrives at the server peer, a conflict occurs, because client B did not receive the current server value. To signal this, the server peer sends a rejected message to client B. This leads to a "Local Change" state transition to *Rejected* at client B. After the reject message, the server peer will eventually send the field update to client B. After receiving the update message, he must choose between his local value, the

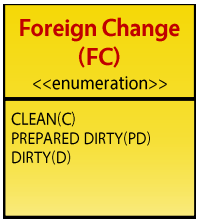


Figure 3.10: Foreign Change

updated value, or enter a third one.

Once the conflict is resolved on client B, it sends his new value to the server and waits for an acknowledge.

Object level aggregation: Has no object level semantic.

Foreign Change

As we discussed, it is important for a client to know in real-time what actions the other clients are doing, to prevent conflicts. Therefore we need to know for every field if somebody else has the intention to change it, has already changed it, or nobody is willing to change it. "Foreign Change" is a field meta-information that mirrors the other client peers actions in real-time. (Figure 3.10)

CLEAN

If no other peer has the intention or modifies the field, then the "Foreign Change" is in the *Clean* state. In this state an user can modify the field with no conflict risk.

Object level aggregation: Has no object level semantic.

PREPARED DIRTY

If at least one peer has the intention to change the field value (clicked in a textfield, selected a combobox etc.) then the "Foreign Change" switches to *Prepared Dirty*. This is more like an attention that another peer may change the value of this field and if the current client changes it as well, a conflict can appear.

Object level aggregation: Has no object level semantic.

DIRTY

The *Dirty* state is active when at least one peer has modified the field. This lets the client know that the field value will be shortly updated with a new value and it is therefore risky to edit it as well because of a conflict.

Object level aggregation: Has no object level semantic.

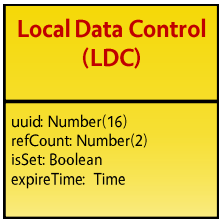


Figure 3.11: Local Data Control

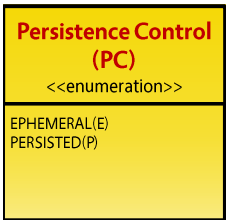


Figure 3.12: Persistence Control

Local Data Control

"Local Data Control" is an object meta-information that is motivated rather technically. These informations are used by the SyncLib to manage objects internally and do not interfere with business data. (Figure 3.11)

UUID
The *UUID* (Universally Unique Identifier) is necessary for the SyncLib to identify objects uniquely. It is generated by the peer that creates the object using a combination of the device MAC address and time stamp.

REFCOUNT
The *refCount* holds the number of existing references on the object. It is necessary for garbage collection. If the refCount is 0 then the object will be deleted automatically and must be afterwards reloaded from the server peer when needed.

ISSET
The *isSet* flag states if the object must be treated as an object set (true), or not (false).

EXPIRETIME
When an object is created its *expireTime* field is set to a default value (e.g.: 10 minutes). If the object is not used for that period of time, then it is automatically deleted. At every object access, the *expireTime* attribute is reseted.

Persistence Control

"Persistence Control" is an object meta-information that enables or disables persistence for an object.(Figure 3.12)

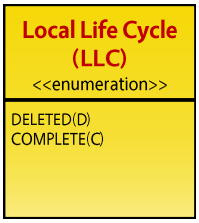


Figure 3.13: Local Life Cycle

EPHEMERAL
The *Ephemeral* state means that the object does not need to be persisted by server peers that have the task to persist data. *Ephemeral* objects are usually private between a client peer and a server peer. (see 5.4)

PERSISTED
The objects that are in the *Persisted* state need to be persisted by server peers having that responsibility. In general these are business objects, that must not get lost.

Local Life Cycle

The "Local Life Cycle" meta-information stores if an object is complete or deleted. (Figure 3.13)

DELETED
When an object is deleted on the client, the values are not immediately discarded, but they are still stored, until the server acknowledges the delete action. Only the "Local Life Cycle" state switches to the *Deleted* state, and a delete message is sent to the server. If the server accepts it, then the whole object is discarded on the client. When an object switches to the *Deleted* state it is not displayed in the UI and the SyncLib will send a delete message to the server.

COMPLETE
When an object is created the "Local Life Cycle" switches automatically to the *Complete* state.

4 Architectural Considerations

In this chapter architectural concepts and solutions will be presented, identifying the main software components and analyzing different ways to combine them.

4.1 Architectural Components

A business information system that uses the synchronization library is composed of many network-connected peers with different roles. In figure 4.1, the most simple architectural context view of a SyncLib-driven software application is shown. One or many client network peers are connected to a server network peer. Following architectural components can be identified:

Rich Client Functionality
: Is the software component that uses the SyncLib and contains application specific code. It runs in a browser on a client device and communicates over the network with a thin server.

SyncLib Client Peer
: Is an instance of the SyncLib library with the client role (see 4.2). It observes the data model for changes, the UI for user actions and holds the escalation detection logic. It needs to send messages to the server peer, every time data has been changed by the user, or new meta-information is available for other peers.

SyncLib Relay
: Plays an important role when more than one server is needed. If this is the case, a first naive solution, would be to connect all the N client peers to all M server peers, resulting N x M connections. The big disadvantage here is the high number of connections, and that all the clients will know about all the servers, what gives no flexibility in removing or adding new peers to the network.

  A better solution is to introduce a machine in the middle with the SyncLib relay software (see 4.3) deployed on it, to transform the N x M relationship into N + M. Having a relay in the middle, separates the clients from the servers and this brings flexibility because they don't have to know about each other any more. The resulting architecture is also known as "Hub/Sandglass" [3] and brings loose coupling in communication and referencing between solution components, by reducing the number of interconnections between peers.

SyncLib Server Peer
: Is an instance of the SyncLib library with the server role. The main differences to the client peers are query execution and authority (see chapter 4.2).

Thin Server Functionality

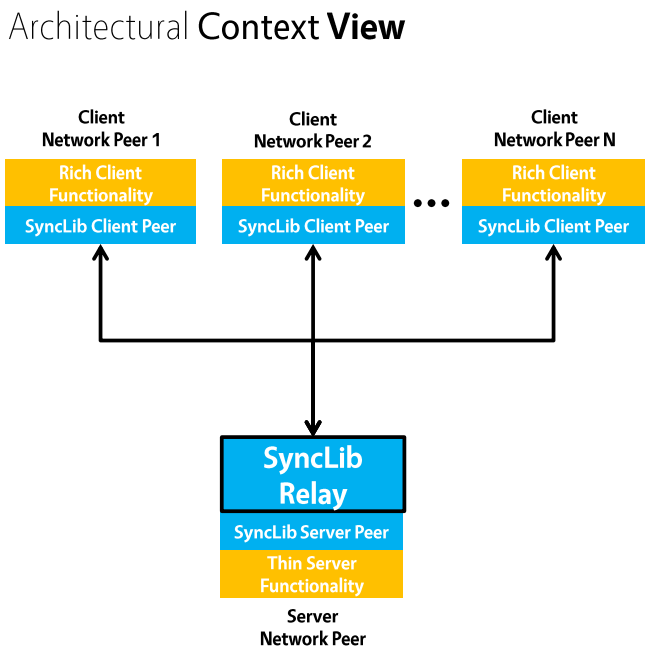


Figure 4.1: One Application - One Server Architecture

Is the software component that uses the SyncLib and implements the server logic. Its main role is to offer application specific business services and to read and write to the database.

4.2 SyncLib Peer Roles

A network peer represents a device that has the possibility, to be connected to a network. In a network of peers, we differentiate between peers that are used mainly for user interaction (client peers) and others that have data processing and usually persistence tasks (server peers). The SyncLib library needs to exist on both client and server network peers, in order to be able to synchronize their data models. An interesting question to answer, is if there is a difference between the SyncLib client component and the SyncLib server component. Is it possible to use the same SyncLib library implementation on both client and server peer, or two different ones are needed?

The table in figure 4.3 contains all the functionalities provided by the library and associates them with the SyncLib components. The highlighted rows show the functionalities where the SyncLib client and server peers distinguish themselves from each other.

Resolve Queries

To resolve queries, a peer is needed, that holds a connection to the database. The SyncLib library forwards query parameter values or an abstract syntax tree to the application service adapter component, depending what strategy is used (see 5.5.2). The application service adapter component receives the query parameters and passes them to the concrete query or receives and parses an query abstract syntax tree and generates from it a an expression in the database specific query language. A server peer has such a responsibility and that's why this represents a point where the client and server SyncLib peers differ.

Human Escalation

Human Escalation means to let the user decide between two values in conflict or enter a third one. This happens after the escalation detection stage, presented in chapter 3.2.1, has a positive result. Because an user interacts with the application through a network client peer, the SyncLib library must have the human escalation module present just in the SyncLib client peer.

Authority

In a big application the SyncLib components (client peer, server peer, relay) are interconnected. Existing data travels from the database to the SyncLib server peer, then to the SyncLib Relay, from there to the SyncLib client and optionally to a client local cache (see Figure 4.2).

The components of the system are: Rich Client Functionality, Thin Server Functionality, three SyncLib instances (SyncLib Server Peer, SyncLib Client Peer 1 and SyncLib Client Peer 2) and two SyncLib Relay instances. Between them, two synchronization processes are present. A first one, between the SyncLib Server Peer and the SyncLib Client Peer 1 and a second one, between the SyncLib Client Peer 1 and the SyncLib Client Peer 2. A data chain is therefore formed, from the

database to the client cache. The SyncLib library is responsible to synchronize the server data model with that of the client and the client data model with the local cache.

The authority concept is important when data needs to be acquired. If many peers are connected to a relay and one of them is requesting data, then the relay must forward the request to an authoritative peer, which is usually a server network peer, because it has access to the database. Figure 4.2 illustrates two synchronization stages. In “Synchronization 1” the SyncLib Client Peer is non-authoritative and the SyncLib Server Peer is authoritative, based on the SyncLib Relay 1 . In the “Synchronization 2” process, the left SyncLib Client Peer is authoritative and the right one is non-authoritative. This is because the cache values are always overwritten by the data model values and not the other way around.

If peer 1 sends a field with a newer tag than peer 2 to the relay, and peer 2 is authoritative and peer 1 is not, then peer 1s message will be rejected by the relay even if its tag is newer.

Data access authorization

In some cases authorization checks need to be done before a SyncLib peer sends data to the network. In business information systems the authorization logic is located usually on the server component, therefore it is common sense to place the authorization handling in a SyncLib server peer.

As shown above, the SyncLib server and client peers have few differences, what means that a single library can be developed containing the common and distinct functionalities. When the library is instantiated, its role can be selected according to the relay it is connected to. In figure 4.2 three SyncLib instances exist: two of them in the client role and one in the server role.

4.3 SyncLib Relay

In the simple case when a server network peer is enough regarding performance and response times, the relay plays not big role. It can deployed on the server machine, together with the SyncLib server peer component and the thin server functionality component (see figure 4.1). The rationale behind its existence is to handle message passing between clients. The logical separation architecture principle is here applied that states, that the components within an application that have different concerns, must be separated.

In figure 4.6 the relay is deployed on a separate machine and becomes a single point of failure. This means that the relay logic must be stateless, in order to not loose data in case of a failure. The relay has the responsibility to route messages, containing data updates and meta-information, between client and server peers. An important insight is that not all client peers are interested on the same data set, because they can have different parts of the data model loaded.

This means that the relay must know for what data set each client is interested at any given time. But this means that he must hold a state. The trick to achieve a stateless behaving relay, that still holds state is to let each client peer tell the relay in what he is

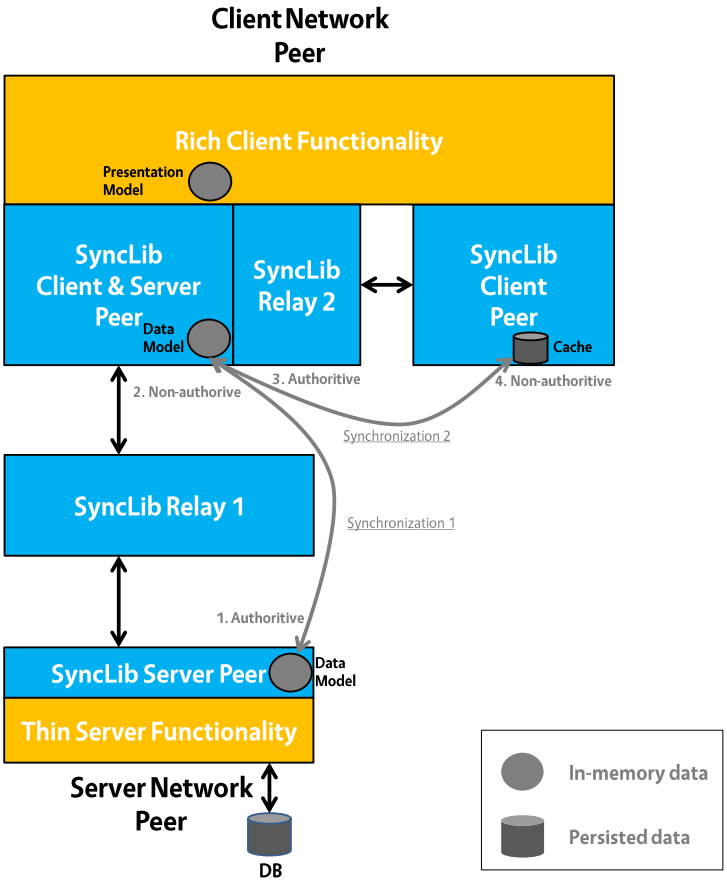


Figure 4.2: Authoritative and Non-authoritative Peers

SyncLib Component **Differences**

Functionality	Client Peer	Server Peer	Relay
Define entities	X	X	-
Create objects	X	X	-
Modify objects	X	X	-
Delete objects	X	X	-
Define queries	X	X	-
Resolve queries		X	-
Detect conflicts	X	X	-
Conflict Prevention	X	X	-
Conflict Resolution	X	X	-
Human Escalation	X	-	-
Meta-info generation	X	X	-
Meta-info aggregation	-	-	X
Message routing	-	-	X
Authority	-	X	-
Data access authorization	-	X	-
Business rules authorization	X	X	-
Data Timestamps/Versioning	X	X	-

Figure 4.3: SyncLib Component Differences

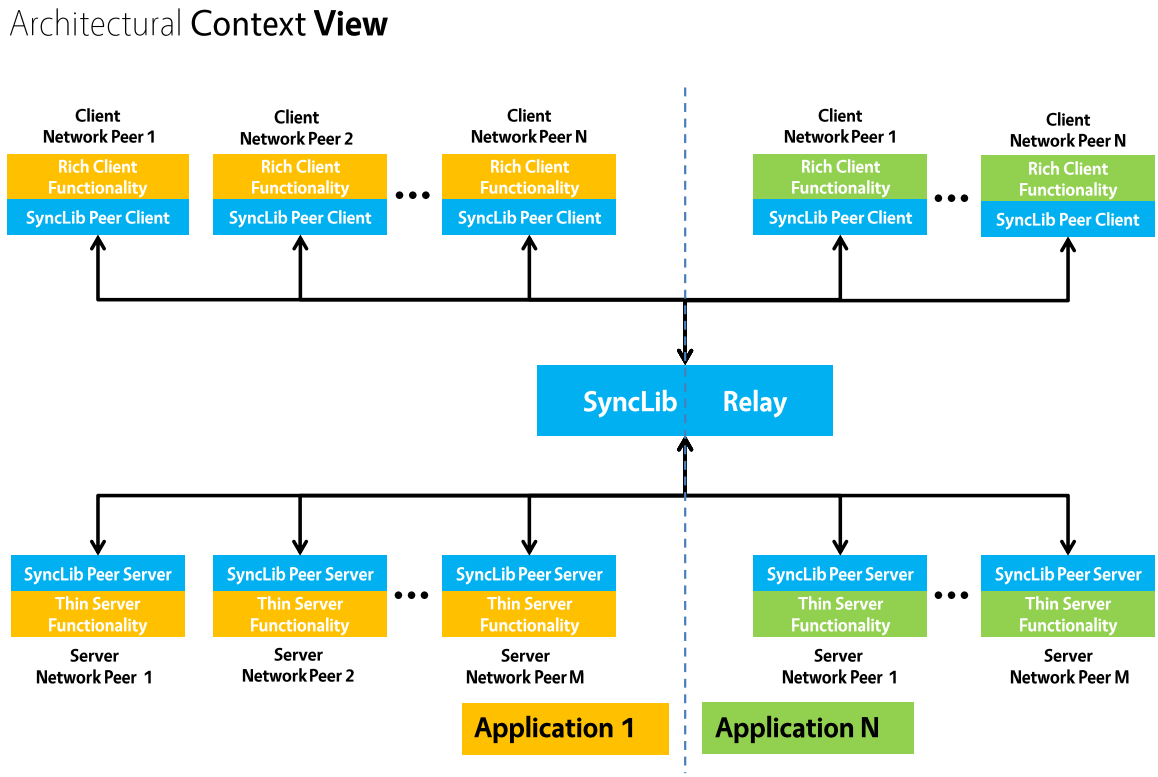


Figure 4.4: Many applications - Many servers Architecture

interested at connection time. In this way, if the relay has a failure and is restarted or replaced, the clients will reconnect and send the relay the data set they are interested in and consequently the relay can reconstruct his lost state.

Besides the “interested table”, the relay must additionally store some information necessary for meta-information processing. As described in chapter 4.3.3, the relay must for example count how many peers acknowledged an update in order to be able to generate server peer status meta-info messages. In case of a relay failure this type of information will get lost and can’t be reconstructed from the clients, because just the relay possesses it. The effect of this is not something crucial for the system functionality. The only thing that can happen is that the client has not the correct field warnings displayed. But after the first field update is sent to the relay it will be correct.

Moreover, the relay does not depend on the data model, because it doesn’t process any data. This means that it is a generic component and it does not need to be configured for each software application separately. In this way different SyncLib-driven applications can use the same relay (see figure 4.4)

4.3.1 Relay Functional View

Figure 4.5 shows the internal architecture of the SyncLib relay. It is designed according to the 3-tier Component Application Reference Architecture [4]. The interaction tier has the responsibility to receive messages from SyncLib peers. The network communication module is a third-party WebSocket library, that is used as a bidirectional transportation mechanism. The received messages are then passed over to the protocol module, that knows how to process the incoming SyncLib messages and how to interact with the SyncLib relay service façade.

The service tier has two main components: 'Routing' and 'Meta-Info Processing'. 'Routing' contains all the logic needed to determine to which peers to forward the messages. To achieve this it uses mainly the 'Interested Data' component. On the other hand, the relay needs to process some incoming meta-information and in some cases create some itself. This is why the 'Meta-Info Processing' component is split in two: input and output. The input part analysis just the meta-info messages received from other peers. The output part decides when to produce and send 'Server Peer Status' messages, according to the received acknowledgments.

In the data tier we find the actual 'state' of the relay. First, an interested table that stores information about what data sets each peer is interested. 'Meta-Info Data' contains information needed for meta-info processing. For example, in order to create different 'Server Peer Status' messages, the number of connected peers and the number of received 'ACK' messages for each update must be known and stored.

4.3.2 Interested List

As noted above, the relay must know in what data set each client network peer is interested. This is important when the relay receives a message from a peer, and needs to know to which peers it needs to forward it. Therefore a data structure similar to a routing table must exist on the relay. A network peer can express its interest on different abstraction levels: object type, object and field. If a client peer is interested on an ObjectType, the relay will forward him all messages that contain updates or meta-information regarding all objects that belong to that ObjectType.

If a client peer is interested on an Object, the relay will forward him all messages that contain updates or meta-information regarding just that object and all its fields.

If a client peer is interested on an Field of an Object, the relay will forward him all messages that contain updates or meta-information regarding just that field of the specified object.

4.3.3 Meta-information message processing

Meta-info messages contain local meta-info change notifications of a client peer, that need to be transmitted to the other interested client peers as well. When such a message is received, the relay forwards it directly as it is, or makes small computations first and then sends it modified to the other interested client peers.

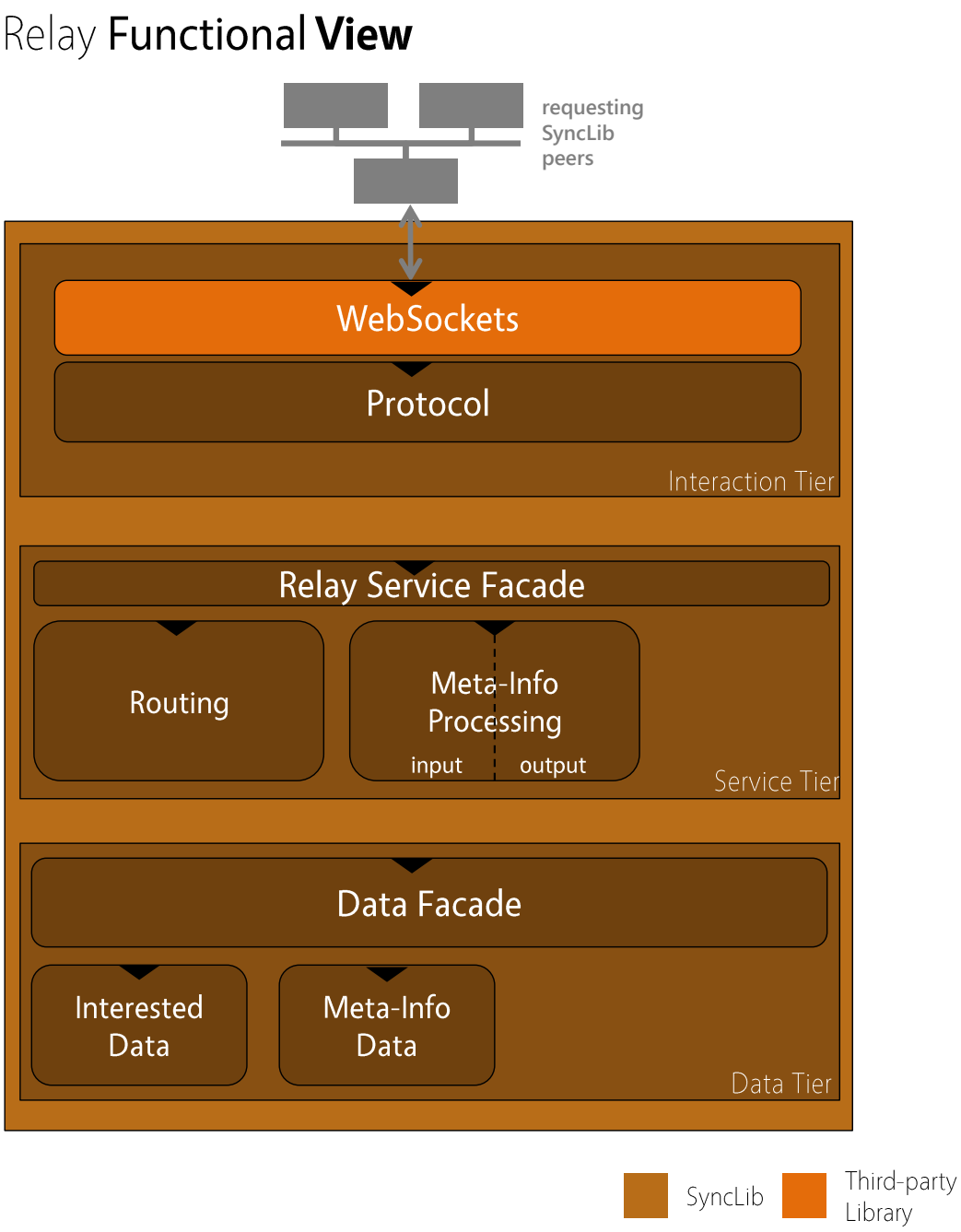


Figure 4.5: Relay Functional View

In other cases the relay generates meta-information messages, according to the number of acknowledges received. Next, we will discuss how the relay handles all the meta-information types.

Server Peer Status
The server peer status messages are always generated and sent by the relay to the client peers after it receives a field update message. The server peer status message can have three values: 'Server Synced', 'Server Peer Partially Synced' and 'Server Peer Fully Synced'.

The 'Not Server Synced' message is not sent, it is set locally by the client peer when the corresponding field is modified.

The 'Server Synced' message is sent to the client peer, after the relay checked if the client peer update does not cause a conflict and a server peer persisted the received value.

The 'Server Peer Partially Synced' message is sent to the client peer after the first peer acknowledged that he got the forwarded field update message from the relay.

'Server Peer Fully Synced' is sent to the client, after all peers acknowledged the forwarded field update.

Local Presence
The Local Presence meta-information is never sent over the network. It is stored just locally on the client peers.

Sync Control
Sync Control has two components: 'Priority In' and 'Priority Out'.

'Priority Out' sets the priority of the messages sent from the client peer to the relay and therefore it is set and stored just on the SyncLib client library.

'Priority In' is set by the client peer and needs to be sent over to the relay because it controls the priority with which the relay sends messages to the client peer.

Local Change
Local Change is always sent from the client peer to the relay. It is transformed by the relay into a 'Foreign Change' message without modifying the value and distributes it to the other interested client peers.

Foreign Change
Foreign Change is always sent from the relay to the client peers after receiving a local change message from a peer.

Local Life Cycle
Local Life Cycle meta-info messages are forwarded to the other peers directly.

4.3.4 Relay Failure

An important topic to handle is what measures can be taken in case the relay has a failure and the system needs to be recovered fast, to ensure availability. A solution

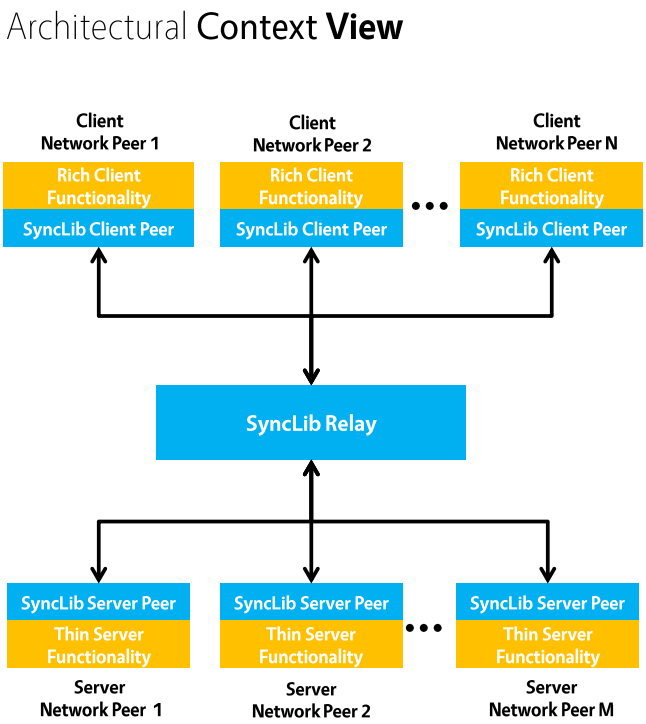


Figure 4.6: One Application - Many Servers Architecture

is to have a second relay in stand-by mode. Once the first relay has a failure, its IP address will be mapped on the second relay and all the peers will reconnect and send the "interest" messages. After that the second relay reconstructs its state and the whole system is recovered.

Another option is to place a load-balancer before the two relays. In this case, if the first relay fails the messages are forwarded automatically to the second one by the load balancer without requiring a reconnect. In this option the two relays must be in the same state. This means that the load-balancer needs to send all messages to both relays.

4.4 SyncLib Application Scalability

If the application needs to scale and more than one server is needed, then the relay needs to be deployed on a different machine that sits between the clients and the servers. The main benefit is that in this way, as stated above, we reduce the architecture complexity to a N+M relationship (see figure 4.6). Following goals could be achieved by having more than one server:

Replication

In this approach each server network peer contains the same application compo-

nents and controls the whole data model. Geographic distribution is a common motivation behind server replication.

The difficulty here is to keep the servers synchronized under each other. Probably a Master-Slave strategy must be used between the server peers, so that in case of inconsistency, all the slave peers take over the master peers data model. Another solution could be implementing a 2PC (Two Phase Commit) mechanism inspired from distributed databases. The disadvantage here is that the 2PC algorithm is synchronous and large delay may occur when writing data to the servers.

Data Partitioning

If the client network peers can be grouped based on the data set they use, then the data partitioning can be achieved. In this case each server holds just a part of the data model and client peers are directed by the relay to the server peer that holds the data set they are interested in.

In this case the relay needs to know additionally how the data model is divided between the servers. This can be done by letting each server send the relay at connection time the information about what data set it holds. In this way the relay can reconstruct its state after a reset.

4.5 SyncLib Library Integration Options

In this chapter we will discuss different possible scenarios, in which the SyncLib library is used. It is important to design the library from start with the thought in mind that it should be able to work along software systems already in use and obviously also with some that were built with the library from scratch.

Single Protocol SyncLib Application

The most simple scenario is the one in which every network peer is designed and built from the start with the SyncLib library. This means that the whole data model is defined using the SyncLib Definition Module, the objects are created through the SyncLib and the whole network communication is done using the SyncLib(SL) components (SL relay, SL client peer and SL server peer).

The functional view of such a network server peer is displayed in figure 4.7. Everything colored blue is application specific, the brown components are modules of the SyncLib library and the orange modules are third party libraries. The interaction tier contains the SyncLib communication module that is connected to all other SyncLib peers. The application service adapter is needed for interacting with the library and linking it to the service façade.

The service tier contains the application specific business logic grouped in service components. The service façade has the role to hide the concrete service components.

The data tier contains components that hold the connection to different data sources (databases, external systems).

The entity components are perpendicular to the three tier and represent the data model definition. An application definition module is needed that uses the Syn-

cLib Definition Module and holds all the code for defining the object types. For completeness, the functional view of a network client peer is shown in figure 4.8.

The architecture is designed following the 3 Tier Component Application Reference Architecture described in [4].

Multi Protocol SyncLib Application

Sometimes a SyncLib peer should be able to communicate with another non-SyncLib peer. The first burden is the network protocol for interconnecting them, because the external peer could not support Web Sockets for example.

A first solution, presented in 4.9, would be to add to the SyncLib a new interaction component that is compatible with the external peer. In the picture besides Web Sockets, a REST communication component is added to enable interconnection with other non-SyncLib peers. From an architectural point of view it looks very good and seems a fair solution. The hidden issue here is that REST has a very vague and permissive specification and it is very hard, if impossible, to write for the SyncLib a generic REST client and server that is compatible with every system.

Even if it would be possible, the external peers will be surely not compatible with the SyncLib messaging protocol.

In conclusion, a multi protocol SyncLib architecture looks good in theory, but is unfeasible in practice.

Add-on SyncLib Application

Because integrating in the SyncLib different network communication protocol components is not a good idea, another option described in figure 4.10 emerged. The SyncLib is designed there as an add-on library. This means that it should be able to be integrated into any existing software system with ease. For communication with non-SyncLib peers a custom, application specific REST component is used and the communication with other SyncLib clients is done through the integrated SyncLib peer.

The challenge in this solution is that the data model is outside the SyncLib. External systems and the local SyncLib peer need to access it. For the SyncLib, data adapters can be written in order to connect it to the application specific data component management mechanism (ex. JPA). Because data can be modified from outside the SyncLib (through the REST component), conflicts can emerge. Therefore a data tagging mechanism must be introduced in order to maintain consistency in the whole system.

4.6 SyncLib Server Peer Functional View

In this section the internal components of a SyncLib peer will be presented. Figure 4.11 shows the internal architecture of two connected SyncLib peers. Both follow the 3-tier Component Application Reference Architecture [4], each tier having different layers. The interaction tier has the responsibility to receive messages from the SyncLib relay. So its first layer is a “Web Socket” component that knows the Web Socket

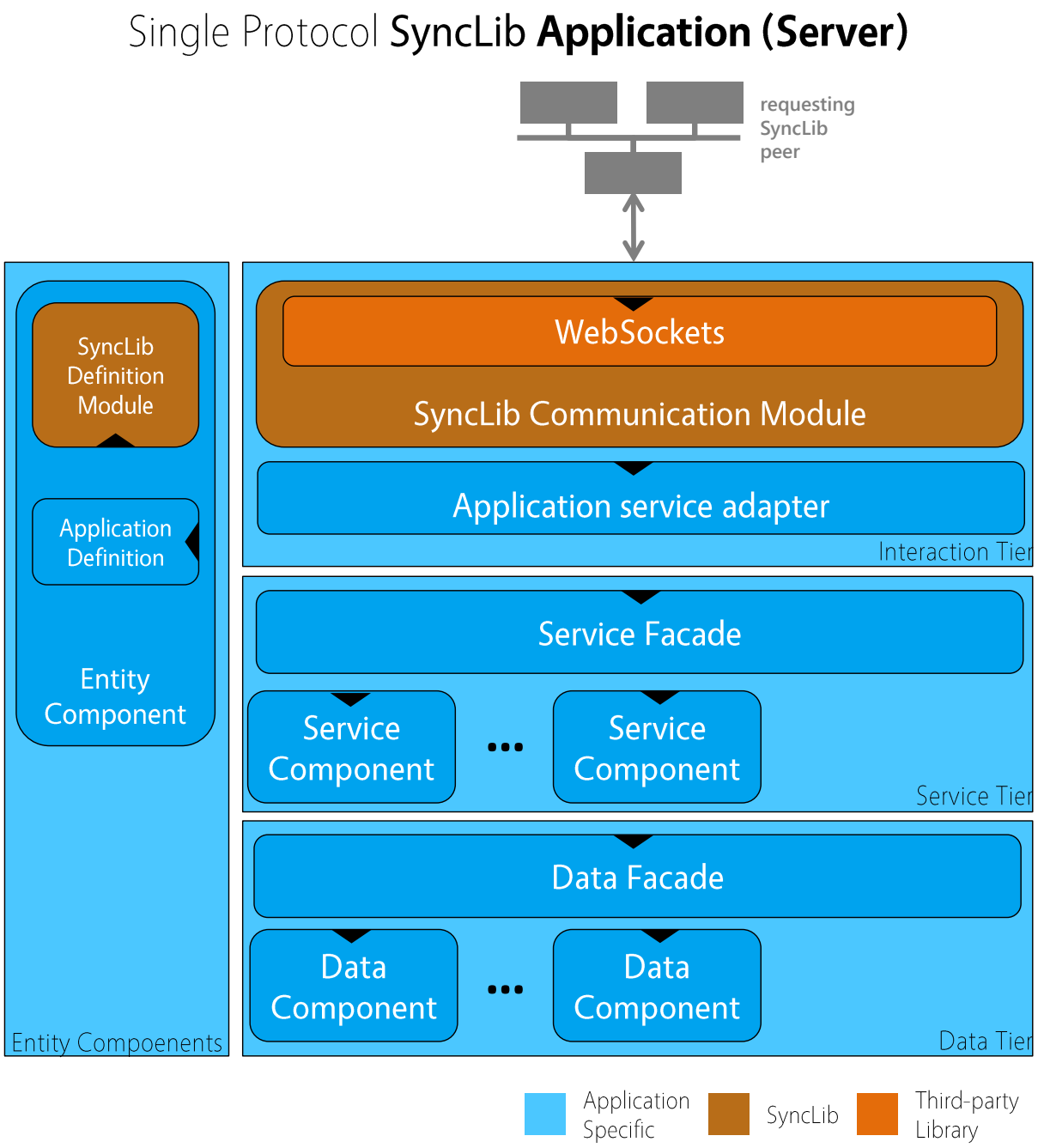


Figure 4.7: Functional View - Single Protocol SyncLib Application-Server

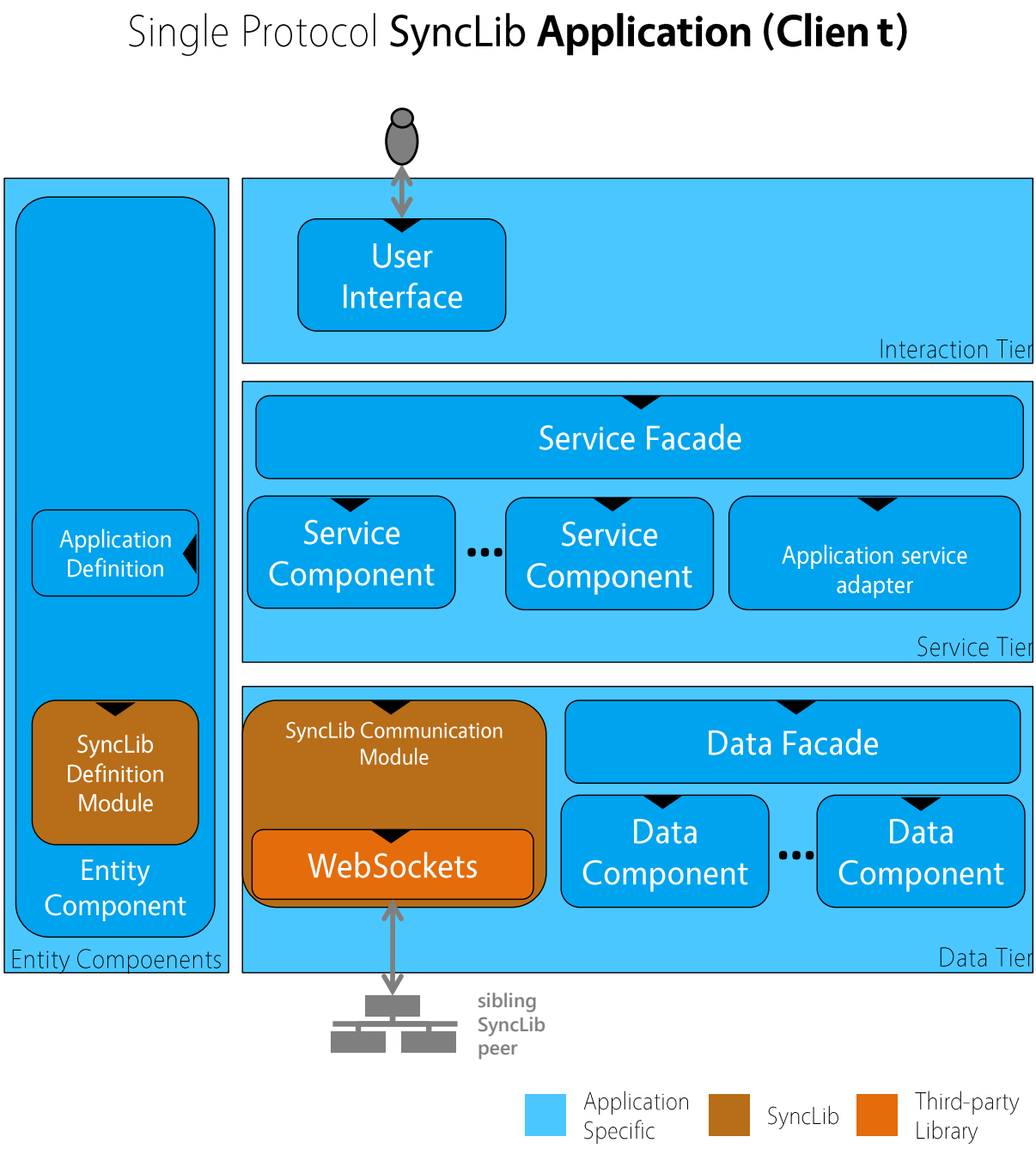


Figure 4.8: Functional View - Single Protocol SyncLib Application-Client

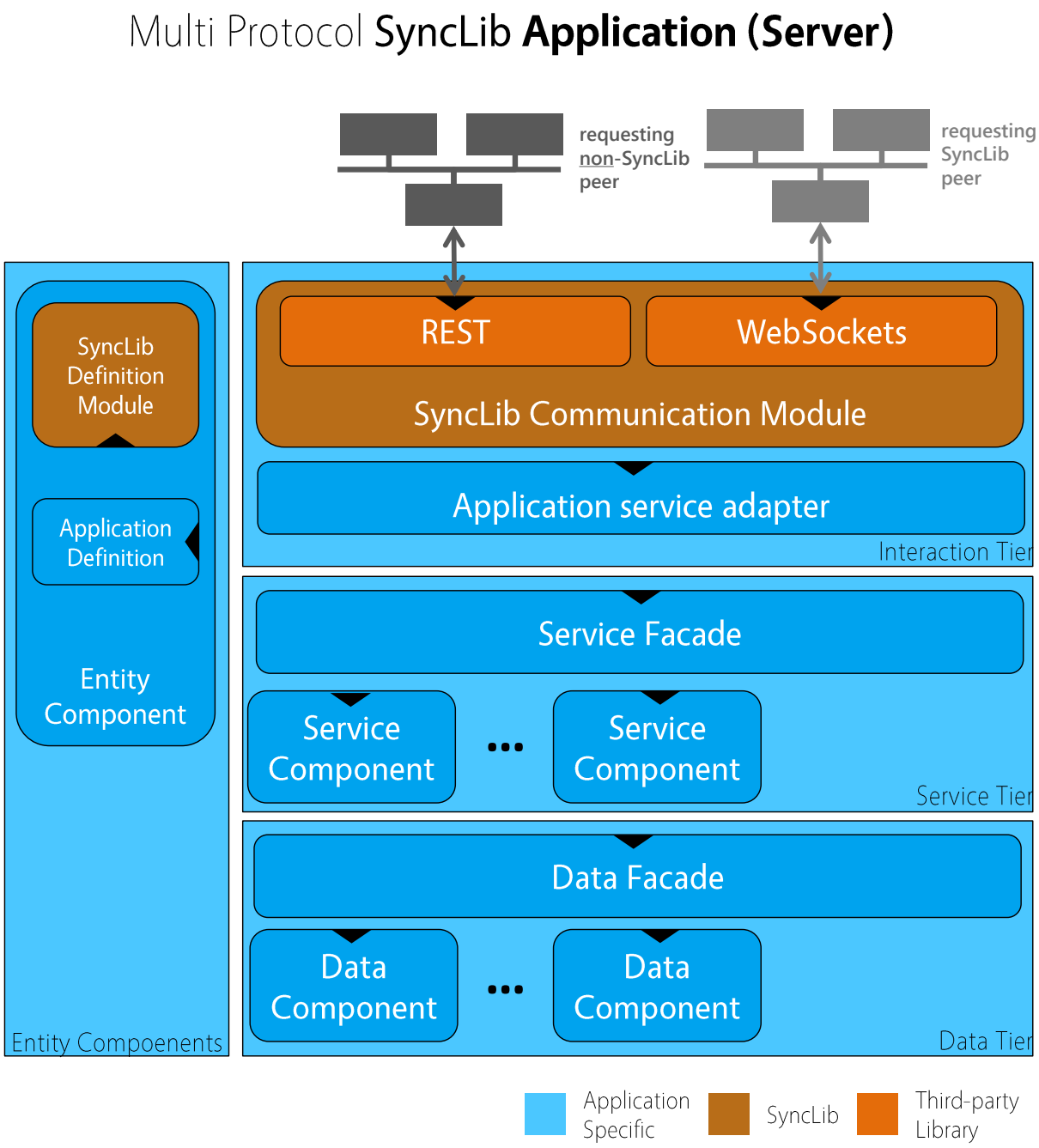


Figure 4.9: Functional View - Multi Protocol SyncLib Application-Server

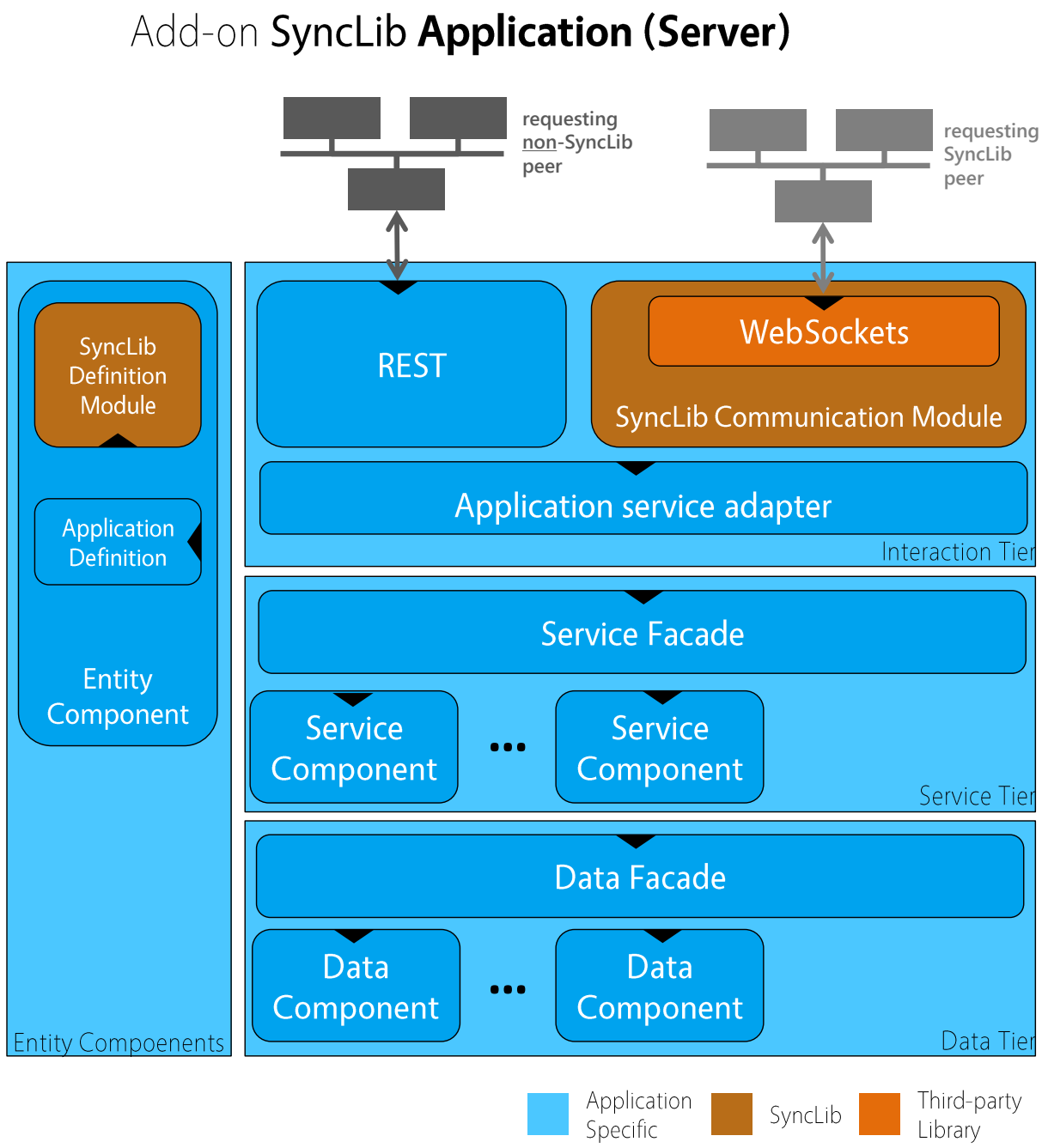


Figure 4.10: Functional View - Add-on SyncLib Application-Server

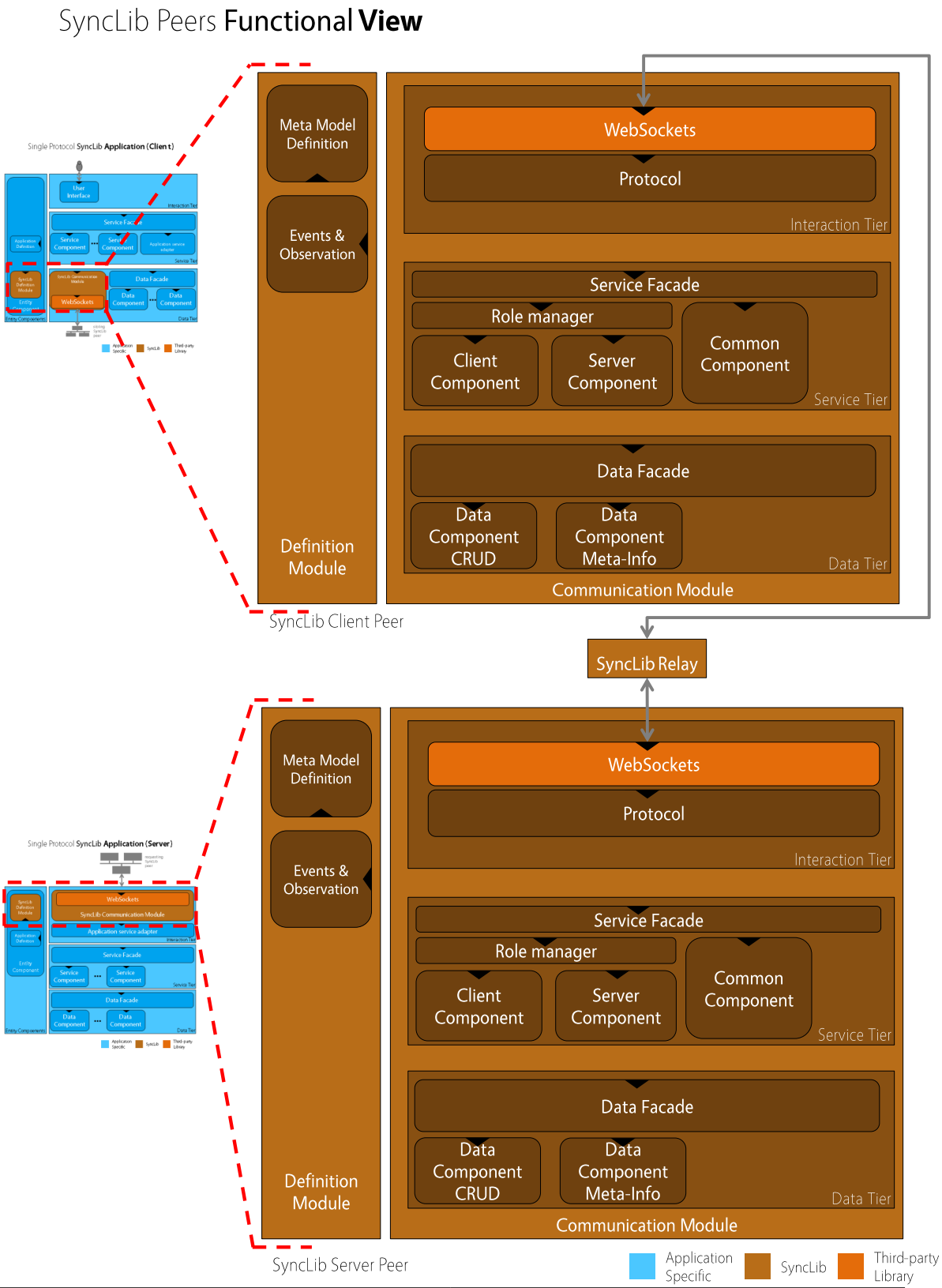


Figure 4.11: Functional View - SyncLib Peer

transport protocol. The messages are then sent to the “SyncLib Protocol” layer, that knows how to process the incoming SyncLib messages and what methods to call from the SyncLib service façade.

The service tier does not depend on the interaction tier. Its first layer is an service façade, so that the interaction tier is decoupled from the concrete service components. Under the façade are three service components: Client, Server ans Common. The most functionality is in the Common component, because the client and server peers are very similar, as discussed in chapter 4.3. The Client Component and the Server Component are used if the peer is configured to run in the client or server role.

At last, the data tier is concerned with the data access. Its first layer is again a façade, because of the two concrete data components. The CRUD data component is an adapter to a data model, where business data is written and read. The data model can be inside the data tier (if the application is written from scratch with the SyncLib library), or it can be external (in case that the SyncLib library is added to an already existing application).

The meta-info data component has the same semantic, but it is concerned just with the meta-info data. Another important point is that in correspondence with the 3-Tier Component Application Reference Architecture, the dependencies go between tiers just downwards, as displayed by the black arrows. This facilitates easy tier replacement.

The definition module exports the API for defining ObjectTypes, creating objects, observing object changes and reacting on specific events. For example the server tier must observe the objects in order to know when a meta-information or data has been changed. Events are usually triggered by messages received from other peers (data update, meta-info update). When such a message is received the SyncLib definition module triggers an event and the surrounding application can react to it.

5 Synchronization Operations

5.1 Synchronization Challenges

Maintaining data in sync across many network peers is not trivial and raises some thoughts. Consistency is one of them. In the ideal case, the peer data models should be the same at any given time. This is similar to the atomic commitment problem in distributed databases. In order to achieve an atomic commit, a transaction must finish successfully or fail on all peers.

This means each transaction must be atomic and its effect must be consistent across peers. Atomicity and Consistency are two key properties of the ACID(Atomicity, Consistency, Isolation, and Durability) model. The ACID properties are prerequisites for a reliable database and are nowadays fulfilled by all serious RDMS (Relational Database Management Systems).

Jim Gray introduced this idea in the 1970s and published afterwards in June 1981 a paper entitled “The Transaction Concept: Virtues and Limitations” [5] where he presented first the terms Atomicity, Consistency and Durability. Isolation was introduced later. The term ACID was announced 1983 in the paper “Principles of Transaction-Oriented Database Recovery” [6] published by Andreas Reuter and Theo Härder and has the following meaning:

Atomicity:
Either the task (or all tasks) within a transaction are performed or none of them are. This is the all-or-none principle. If one element of a transaction fails the entire transaction fails.

Consistency:
The transaction must meet all protocols or rules defined by the system at all times. The transaction does not violate those protocols and the database must remain in a consistent state at the beginning and end of a transaction; there are never any half-completed transactions.

Isolation:
No transaction has access to any other transaction that is in an intermediate or unfinished state. Thus, each transaction is independent to itself. This is required for both performance and consistency of transactions within a database.

Durability:
Once the transaction is complete, it will persist as complete and cannot be undone; it will survive system failure, power loss and other types of system breakdowns.

In a distributed environment, relational databases implement the two-phase commit

synchronization protocol (2PC) [7] in order to ensure global atomicity of transactions.

Considering availability and performance, the 2PC is problematic because it is a synchronous algorithm based on consensus. As its name suggests, 2PC operates in two distinct phases. The first phase involves proposing a value to every participant in the system and gathering responses. The second commit-or-abort phase communicates the result of the vote to the participants and tells them either to go ahead and decide or abort the protocol.

In a real-time web applications, for which the SyncLib is designed to be used, availability is the most important aspect. Using the 2PC protocol, peer availability would suffer considerably.

This brings us to the CAP Theorem, also known as Brewer's Theorem, presented in his keynote speech [8] at the ACM Symposium on the Principles of Distributed Computing. The central tenet of the theorem states that it is impossible for a distributed system to ensure simultaneously Consistency, Availability and Partition Tolerance. It works just by considering only any two of the three requirements.

Charles Roe states in his article "ACID vs. BASE: The Shifting pH of Database Transaction Processing" [9] that "the constraints of CAP Theorem on database reliability were monumental for new large-scale, distributed, non-relational systems: they often need Availability and Partition Tolerance, so Consistency suffers and ACID collapses. "Run for the hills" is an apt phrase."

So an alternative to ACID emerged, called BASE (Basically Available, Soft state, Eventual consistency). Its main features, as described in [9], are:

Basically Available:
: This constraint states that the system does guarantee the availability of the data as stated by the CAP Theorem. In other words, there will be a response to any request. Yet, that response could still be 'failure' to obtain the requested data or the data may be in an inconsistent or changing state, much like waiting for a check to clear in your bank account.

Soft state:
: The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'

Eventual consistency:
: The system will eventually become consistent once it stops receiving input. The data will propagate everywhere it should sooner or later, but the system will continue to receive input and will not check the consistency of every transaction before it moves to the next one.

John Cook presents a good comparison between ACID and BASE in his article "ACID versus BASE for database transactions" [10]. The main idea is that ACID works only for smaller systems with few users. Once the software systems scale up to millions of users, such as Amazon, the "I" in ACID is violated by tolerating a small probability that simultaneous transactions could interfere with each other. All this for achieving more performance and availability for the customers. Consistency is in the BASE paradigm less strict than ACID considers it. The peer data model state can be eventually consis-

tent.

This means the data model will reach a consistent state sometime in the future. So inconsistency between peers is allowed with the condition that the peers will reach in the future a consistent state.

"Rather than requiring consistency after every transaction, it is enough for the database to eventually be in a consistent state. (Accounting systems do this all the time. It's called "closing out the books.") It's OK to use stale data, and it's OK to give approximate answers." [10].

5.2 CRUD Operations

In this section we will discuss the data synchronization operations behind the SyncLib. More exactly, what happens when a new object is created, updated or modified by a SyncLib peer. In this context, the object is the central synchronization unit. Data is synchronized across peers through objects.

Create

When a new object is created on a SyncLib peer, its data must be synchronized with all other peers. First, an "interested" message is sent to the relay, so that the peer will receive updates or meta-information about the object in future. After that, the actual object will be sent to the relay in an upadate message in order to send it to all interested peers. In order to get a new object update created by an other peer, an interest on the object type must be sent beforehand to the relay.

More details about the synchronization message protocol will be presented in chapter 7. In listing 5.3 it is shown how an object type is defined and how a new object can be created from it.

```
1  SyncLib.define("Person", {
2      id:        "string",
3      firstName: "string",
4      lastName:  "string",
5      title:     "string"
6  });
7
8  var person = SyncLib.create("Person",
9              { id: "007",
10               firstName: "James",
11               lastName: "Bond",
12             });
13 person.set("title", "agent");
```

Listing 5.1: Create an object from object type

Update

Once an object is created, every time one of its fields is changed, an update message is sent to the relay and accordingly to the other peers, with the new value.

Now, when a peer receives an update message for a field, the object fields value is modified. How exactly this modified value gets displayed in the UI is outside the scope of the SyncLib. The SyncLib defines and operates just on the business model. All the objects together represent the business model. Besides the business model, there is usually also a presentation model that is tailored specifically for the UI.

Between these two data models, a bidirectional data binding exists, in order to pass the data changes back and forth. Many libraries have something similar already built in, or, in less desirable cases, only a business model exists that is tied directly to the UI.

It could be useful to intercept the field update, in order to do some checks or processing with the new value and assign it manually afterwards. Listing 5.2 shows how this can be done.

```
1 person.onUpdateValue(updateCallback(fieldName, newValue) {
2         //check newValue
3         //assign newValue to oldValue
4 } )
5
6 person.onUpdateMetaInfo(updateCallback(metaInfo, newValue) {
7         //change meta-info ui element
8         //assign new meta-info value to old meta-info value
9 } )
```

Listing 5.2: Intercept object updates

Delete

Delete would seem at the first taught very similar to update, but it's not. Delete refers to deleting an entire object, and not deleting individual field values. When a delete is performed, the object is not removed immediately. It must be checked first if a server peer accepts the delete operation.

In case a client peer intends to delete an object that is obsolete compared with the server version, the operation is not possible. The reason is that the user needs to get first all the object updates from the other peers, before it can delete it. This is why the object is first marked as deleted in its "Local Life Cycle" meta-information and after that a delete message is sent to the relay. If an acknowledge is received, the object will be deleted. Otherwise the user must be notified that he is trying to delete an obsolete object.

5.3 Synchronizing Object Sets

An Object Set is a regular object that has usually one or many Relationship Sets as field types. Its role is to group relationship sets in order to synchronize them together, as a unit. Relationship Sets, as presented in section 3.1.4, contain references to many other objects of the same object type. The main benefit of Relationship Sets is the grouping of objects with the same type in a collection.

Use cases where objects have relationships to other ones and need to be modified atomically for consistency reasons are common. Once an object is set to be an object set, all its fields will not be synchronized automatically anymore. Synchronization must be triggered manually on the object set, action that will send all modified values of its fields to the relay in a single update message.

If a field is a relationship set, it contains a list of other objects. They are affected too, if the relationship set becomes part of an object set. This means every object in the relationship sets will not be synchronized automatically, but only at the time the enclosing object set is synchronized.

```
1  SyncLib.define("OrgUnit", {
2      id:        "string",
3      name:      "string"
4  });
5
6  SyncLib.define("Person", {
7      id:        "string",
8      firstName: "string",
9      lastName:  "string",
10     title:     "string"
11     orgUnits:  "OrgUnit+"
12 });
13
14 SyncLib.define("ObjectContainer", {
15     P:  "Person*",
16     OU: "OrgUnit*",
17 });
18
19 var xt  = SyncLib.create("OrgUnit", {
20                            id: "xt",
21                            name: "msg Applied Technology Research"
22                          });
23
24 var james = SyncLib.create("Person", {
25                            id: "007",
26                            firstName: "james",
27                            lastName: "Bond",
28                            orgUnits: [ xt ]
29                          });
30
31 var eve = SyncLib.create("Person", {
32                            id: "008",
33                            firstName: "Eve",
34                            lastName: "Moneypenny",
35                            orgUnits: [ xt ]
36                          });
37
38 var objContainer = SyncLib.create("ObjectContainer", {
39                                    P: [ james, eve ],
```

```
40                                  OU: [ xt ]
41                          });
42
43 objContainer.enableSet();
44 //...
45 //change object set field values
46 //...
47 objectContainer.synchronize();
```

Listing 5.3: Define, build and synchronize an object set

5.4 Ephemeral Objects

Ephemeral Objects hold data that does not need to be persisted in a database and synchronized with other peers. It is shared between a server peer and exactly one client peer.

Examples are predefined objects like: Integer, Boolean, String that come as a result of a query (aggregation values).

5.5 SyncLib Queries

In every business application system, requests must be sent from a client peer to a server peer in order to get data. For example it is required to load all persons above 18, all the companies from a given country, or slightly more complicated: all employees along with their total income that work for a specific organization. The requests hold information, structured in a standardized way (based on what technology is used), that define exactly what data sets are needed. When a request is received by a server peer, it is analyzed and based on that information a database specific query is evaluated in order to get the data.

Such requests are usually like queries, because even if they have an other structure, their semantics is similar to that of the database queries.

Basically, in order to make a query, some data needs to be sent from a client peer to the server peer and after that, a result must be sent back to the client containing the requested data sets.

Nowadays, most web applications need to send every time a query request in order to refresh the results sets. This means that after a query is executed and the results are displayed in the UI, the user has no guarantee that the data he is looking at is up to date.

A new vision regarding queries, would be to send the query just once to the server and after that having up-to-date results without manually resending the query to the server. For example, we can request a list of all available action movies using a rental shop web application. The difference to traditional applications would be that the

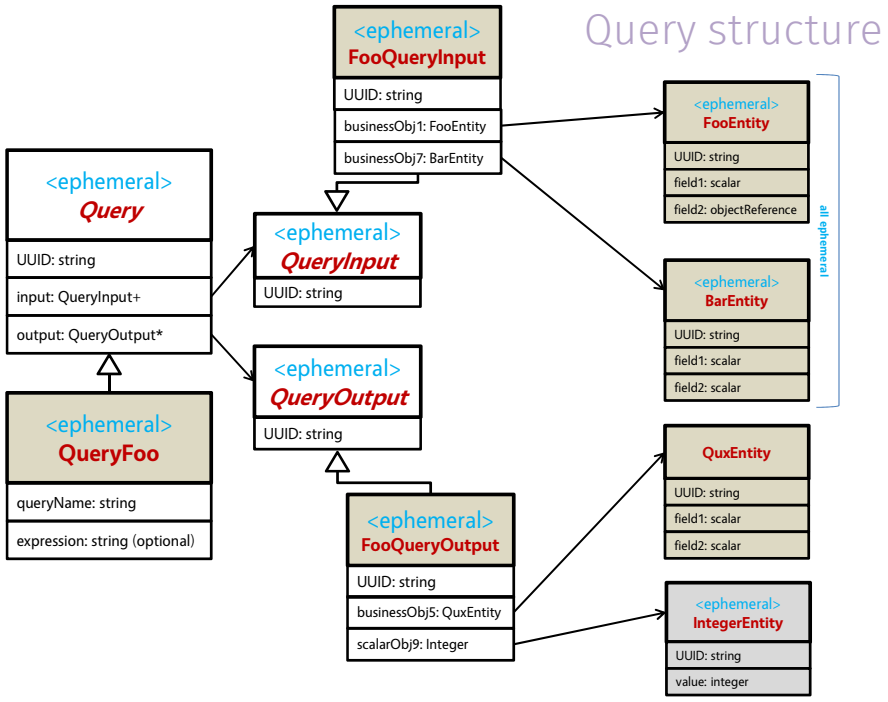


Figure 5.1: Query Structure

movie list is updated dynamically if a new action movie is added, removed or changed in the server database. Without interacting with the UI the user can see how new movies appear or disappear in his list.

The great thing about SyncLib is that it already offers a mechanism to exchange data across different peers. The only thing that needs to be done is to define the query structure using SyncLib.

In this concern, two solutions were conceived, each with advantages and disadvantages.

5.5.1 Queries with Object Sets

Very simply speaking, the purpose of a query is to request a specific dataset based on an expression or filter values. These must be sent over to the server to be interpreted, such that an actual database query is executed and the results are sent back. The filter arguments can be mapped in most cases on business object type fields. Such examples are: name='John', age=35, color=blue, gender=female, etc. All these arguments are already defined in different object types.

Consequently, in order to transmit these arguments to the server, objects can be created from existing object types and synchronized over to the server. Important to note here is that those objects must be marked as “Ephemeral” because they must be shared just between a single client and server peer.

Another observation is that the arguments must be sent over in a single update message, so that the server receives them in a single chunk and knows that he can start executing the query. If the filter arguments are synchronized separately, the server has no clue if it got all messages in order to start executing the query. A first thought would be to send a “startQuery” message after the last update message.

Fist of all, this would introduce a new message type just for queries. As a result, the SyncLib must differentiate between synchronizing normal data and query arguments. Secondly, because the asynchronous message transmission it is not guaranteed that the “startQuery” message will arrive after the last update message containing filter arguments. So it is clear that in an asynchronous environment a good solution is to send all filter arguments in a single update message.

As previously stated, the query arguments are set in different objects. To synchronize them together, an object set can be used as defined above.

Figure 5.1 illustrates the query structure using object sets. First, a query object type is defined that has an UUID (Universally Unique Identifier) and two relationship sets (one for query input and another for the query output). The three object types (“Query”, “QueryInput” and “QueryOutput”) are abstract, predefined in the SyncLib and need to be derived by concrete objects. “QueryInput” and “QueryOutput” are abstract, because at compile time the concrete input object types can’t be known.

This means that in order to define the query, we need at compile time a type for the input and output relationship sets. In this way a type safe definition of a query is achieved. Concrete query objects like “QueryFoo” can be derived from the predefined query base object. “FooQueryInput” and “FooQueryOutput” are object sets and contain relationship sets to other objects containing filter arguments and respectively result objects.

Object sets are great if we want to synchronize an object together with its relationships. It does not work for a higher depth: object, its relationships and the relationships of its relationships. For this query structure exactly this is desired. Moreover, it could be in some cases desired to ignore some relationship sets. To achieve this, a more complex concept is needed where an object graph can be defined based on the object relationships. That object graph then has the semantics of an extended object set.

Another solution would be to define for each relationship set if cascading applies or not. In this way an object set will encapsulate all objects it can reach though relationships that have cascading enabled. The disadvantage is that it is unpleasant for the developer to set the cascading mode for each relationship. At this point it is clear that a concept for grouping more objects together is desired, so that all changes made on them get synchronized in a single update message.

5.5.2 Queries with Sessions

For the problem stated above, Ember.js Persistence Foundation (EPF) has an interesting concept called *session*. All changes are recorded in a session and can then be sent as a single unit to the server when the session is flushed. More details and examples are described in [11].

Adopting this concept into the SyncLib, we could store the objects that need to be synchronized together in a session, then make all the changes and at the end flush the session. Listing 5.4 shows how session can be used.

```
1  var session = SyncLib.newSession(queryFoo,
2                                   fooQueryInput,
3                                   fooEntity,
4                                   barEntity);
5  //changes made on the object inside the session are not synchronized
6
7  //make changes
8  fooEntity.set("field1", 30);
9  barEntity.set("field2", "George");
10
11 session.flush();
12 //merge objects with incoming data
13 //changes will be synchronized in one update message
```

Listing 5.4: Working with sessions

In this way query data can be sent to the server using the SyncLib. Session can also be used also outside the query context. While objects are in a session, updates can arrive from other peers over the network. Those values are stored in each field's foreignValue. After the session is flushed, the foreignValue is merged with the localValue for each field and the user is notified that changes happened. It can also occur that the user is required to resolve some conflicts in case the foreignValue can't be merged with the localValue.

In the previous two sections we described the structure of a query and how filter arguments are sent to the server. In the next two chapters we will describe how the server handles the received queries. For this, two types of query concepts are presented, each with advantages and disadvantages.

5.5.3 Static Query

Static queries have just a name, input and output values. The server identifies the queries using their names and can then refer to a function that contains the code in a database specific query language. The good part is that the SyncLib queries are in this way query language agnostic. The only responsibility is to transfer filter values to the server and receive result values back. All the technology specific query execution code is outside the SyncLib, in the application functionality.

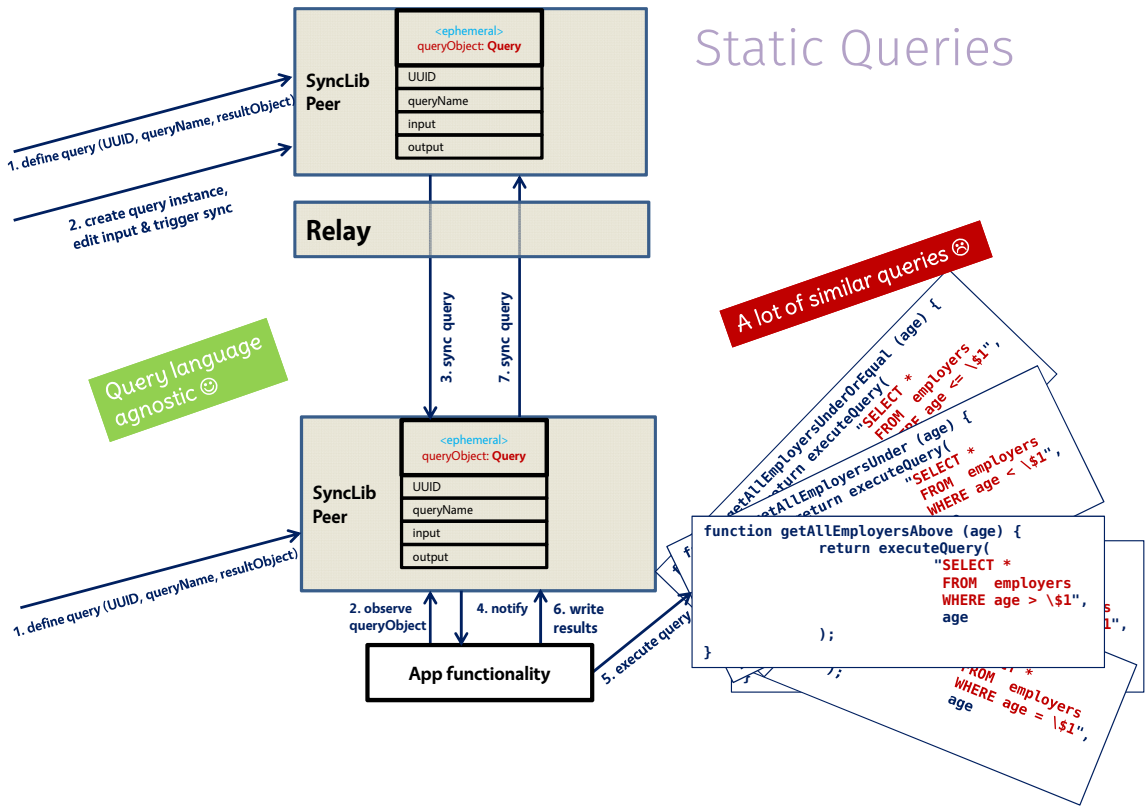


Figure 5.2: Static Query

A disadvantage is that for very small query condition changes, a new query must be defined. The only point of flexibility lies in the query condition values, which can be changed dynamically. But the expression of the query *WHERE* clause is static. Therefore, as shown in figure 5.2, different very similar functions are needed on the server peer if the same query has different conditions (>, <, <=, >=, =).

As a result, all queries must be defined at compile time and cannot be dynamically generated.

As an example, lets assume a client wants all employers under 30 and later all employers above 20. Using predefined queries two functions need to be defined as follows:

```
1  function getAllEmployersAbove (int age) {
2          return executeQuery("SELECT *
3                               FROM  employers
4                               WHERE age > \$1", age);
5  }
6
7  function getAllEmployersBelow (int age) {
8          return executeQuery("SELECT *
9                               FROM  employers
10                              WHERE age < \$1", age);
11 }
```

Listing 5.5: Static Query functions

As we can see, the two queries are almost identical but still need to be placed separately. If “greater or equal” or “less or equal” is needed two more function need to be added, even if the query logic is almost the same. For queries with composed conditions a function can be sufficient, but the *WHERE* clause must be constructed dynamically from the arguments that are not null.

The query execution process adheres to the following steps:

1. First the query object type is defined on both the client and server peer.
2. The server then registers an interest for all query objects belonging to that object type. In other words, it observes the query objects created from that object type and will get notified by the SyncLib if a new query object was created or changed. On the client side a query object is created and the input is filled with the filter arguments.
3. The query object is synced over to the server.
4. The application functionality component is notified that a new query object has been created and has access to it.
5. Based on the query name, a function is called containing the concrete query. That function will receive the filter arguments from the query input as parameters.
6. The function results are written into the query output.
7. Because the query changed on the server side, it will be synchronized back to the client and in this way the results can be shown in the UI.

5.5.4 Queries with Dynamic Condition

Static queries have the disadvantage that the query condition can't be modified during runtime. Just the filter values can be changed on the fly. To allow more flexibility, expressions can be added to queries as illustrated in figure 5.3.

This approach implies the query to have an additional expression that describes the query conditions. For the semantics, a small Domain Specific Language (DSL) can be used to express the conditions. The expression can then be parsed and turned into an abstract syntax tree, which can then be relatively easy converted in the application functionality component into a concrete query language (SQL, the Neo4j Cypher Query Language, etc.).

The advantage here is that queries that have the where clause as the only difference, will reside in a single function. On the other hand, the complexity rises due to the introduction of a DSL that needs to be parsed and converted into an abstract syntax tree. This impacts not just the library, but also the developers also who need to learn additionally the SyncLib query DSL syntax.

Only the projection and table part of the query are static (illustrated in the figure with red). It is also possible to represent the whole query in the SyncLib expression, but this would increase the complexity considerably. The query with dynamic conditions can be considered a good compromise between the two 'extreme' solutions.

The query execution process has in addition to the static version a supplemental expression parsing and abstract syntax tree building step.

5.5.5 Triggers for Query Reevaluation

As presented in the query introduction, SyncLib adopts the vision of automatically updated queries. This means that the server peer must know when it needs to reevaluate a query and update the query object, which is afterwards synchronized with the peers.

The following cases have been identified for query reevaulation:

1. The query input values have been changed.
2. Another peer changed an object that is or could be part of the query result. The server can check if the object belongs to an object type that is part of the query result.
3. An object that is part of the result is deleted.

5.6 Meta-Information

In the data meta model we presented shortly all meta-information attributes and explained why they are needed. In this section, we present all the meta-info states and all possible transitions between them. One of the best ways to illustrate this is through state diagrams, as in figure 5.4.

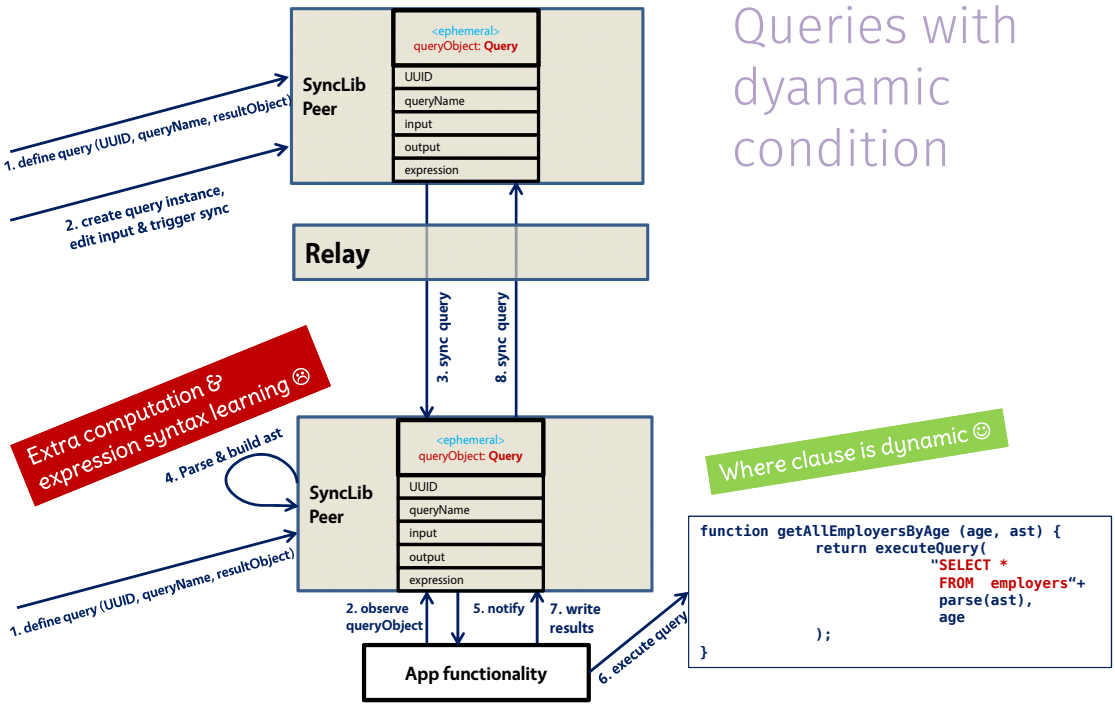


Figure 5.3: Query with dynamic condition

In each of the 6 state diagrams different meta-info states are represented in orange circles. Each arrow represents a transition from one state to another. In some diagrams there are arrows that start from the edges and point to different states. The semantic behind this is that the transition can start from any state in the diagram and leads into the state where the arrow is pointing to. For example, looking at the "modify data" transition from the "Server Peer Status" diagram, we can conclude that whatever the current state is, when the "modify data" event occurs, the current state will switch to "NSS".

Each transition has an input event and optionally an output event delimited by a slash. The events coloured in blue are triggered by user actions, whereas the black ones by messages received over the network from the relay. The black dots represent the initial state.

Persistence Control

"Persistence Control" is an object meta-information and is set only programatically when objects are created. An object is set to *Ephemeral* or *Persisted* based on the business rules. The user UI interactions don't have any effect on "Persistence Control". Usually after the "Persistence Control" is set at object creation time, it doesn't change at all. But changing the value from *Ephemeral* to *Persisted* and vice versa is possible.

Local Life Cycle

When an object is created or loaded from another peer its "Local Life Cycle" is in the *Complete* state. The only way this state can be changed to *Deleted* is if it is deleted locally, or a delete message is received over the network from another peer. Thus, there are two types of deletes, represented in the diagram separately in blue and black. The blue one is a result of an local user UI interaction (e.g. deleted an item from a list). The black one refers to a remote delete action, that is propagated through the SyncLib to all other peers, in order to be replicated on them.

Server Peer Status

As described in chapter 3.2.2, the "Server Peer Status" gives the user information about the synchronization progress for a specific field. Beginning with the initial state represented by the black dot, a transition can be made to *Not Server Synced* if the object containing the field is created. In this case, just the local peer holds the data about the newly created object and therefore all its fields are in the *Not Server Synced* state.

If an object is loaded from a SyncLib server peer, all its fields will be in the *Server Synced* state. This is because that object now exists on both the local peer and the server and they are identical.

If a field is modified locally, the "Server Peer Status" switches automatically to *Not Server Synced*, whatever the current state is. This is modeled by the blue modify data arrow starting from the edge of the diagram and pointing to the *Not Server Synced* state. "New", "Load" and "Modify data" are the only blue transitions in the "Server Peer Status" state diagram because they represent local peer actions. The rest of the transitions are triggered by messages received over the network.

The transition from *Not Server Synced* to *Server Synced* is done after an acknowl-

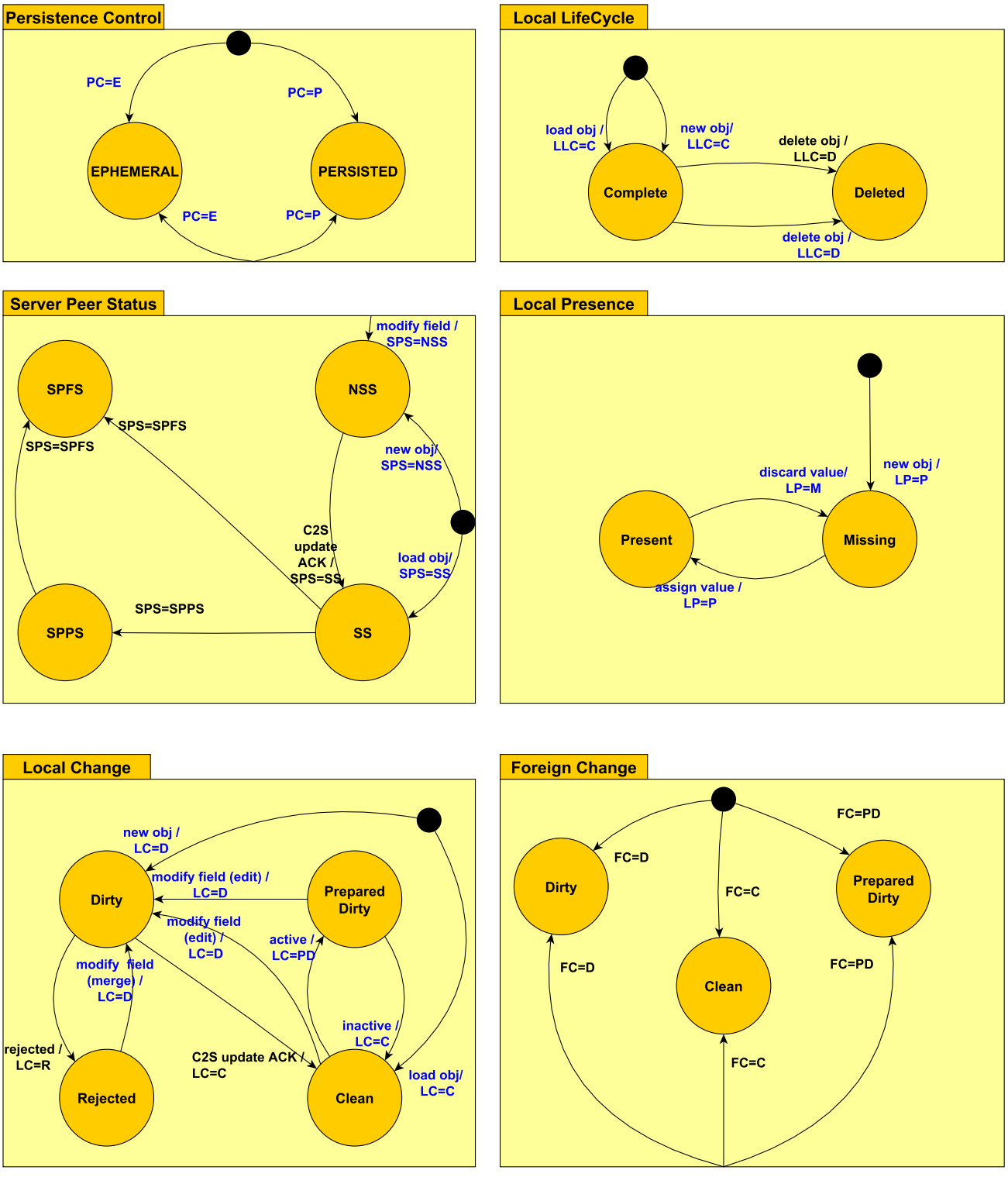


Figure 5.4: Meta-information state diagrams

edge (*ACK*) is received that the client-to-server (*c2s*) update with the new field value has been received and persisted by a server peer. The rest of the transitions ("SPS=SPPS", "SPS=SPFS") are the result of messages sent from the relay regarding the synchronization progress for a field.

Local Presence

"Local Presence" is in the *Present* state if the field holds a value and in the *Missing* state otherwise. The *Present* state is active if the field gets a value assigned. When a new object is created and its fields are not yet initialized, they are all in the *Missing* state.

An object can be loaded partially. This means that some of its fields are *Present* and others are *Missing*.

Local Change

"Local Change" shows how local field values are compared to the server field value. Every time the field value is modified or deleted, the *Dirty* state is active, because through the change, the field is out of sync with the server peers.

If an object is loaded, all its fields are *Clean* because they are equal to the server version. Once an UI element bound to a field is active (mouse-over, textbox select, etc.), the fields "Local Change" switches to *Prepared Dirty*, in order to notify the other peers that the user has the intention to change the value. If the UI element is not selected anymore (inactive), then the state switches back to *Clean*.

After a field is modified and its state switches to *Dirty*, a client-to-server update is sent with the new value, in order to synchronize it with all other peers. If this update is accepted by the server and an acknowledgment is sent back to the client peer, then the fields "Local Change" state goes back to *Clean*.

There is also the possibility that the server rejects the update because its field has been updated with a value that the sending peer did not receive yet. In this case, the server sends a reject message instead of an acknowledge. When the reject message is received, the "Local Change" state switches to *Rejected*.

Next, a server-to-client update arrives and the client must merge the two values and submit it. This brings the state back to *Dirty* and a client-to-server update is sent with the merged value.

Foreign Change

The "Foreign Change" meta-information reflects the actions of other peers. It contains information if a field is not, is probably going to or has been modified by other peers. As shown in the state diagram, the "Foreign Change" meta-information can switch to any state, no matter what the current state is.

6 Synchronization Strategies

When synchronizing data across many individual peers different strategies can be adopted in order to merge data edits together. Neil Fraser presents in his paper *Differential Synchronization* [1] a new synchronization algorithm that offers fault-tolerance, scalability, and support for a real-time data exchange over an unreliable network.

Additionally, he makes a very good analysis and comparison to classical synchronization approaches, which we will briefly present.

6.1 Locking

The most trivial method to guarantee data synchronization is avoiding concurrent data edits. This means that every time an instance wants to modify a certain data set, it needs first to be locked. Only after it has been processed it will be unlocked and therefore available to all other instances.

Locking implies mutual exclusive data manipulation and is therefore a pessimistic synchronization algorithm. “Pessimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycle, whereas algorithms delay the synchronization of transactions until their termination.” [12]. An example is editing a Microsoft Word document from a shared network drive. The one who opened it is allowed to perform changes, but all others have just read-only access.

Locking all the data for the whole UI view, makes collaboration impossible. Therefore, different granularity levels can be defined for locking. This helps if the user tasks target exactly that data portions. But in collaborative use cases even with fine-grained locking, a real-time interaction is very difficult, if not impossible to achieve. Moreover, when a lock is set on a peer, a notification must be sent to all other peers. In case of an unreliable network, those notification messages can get lost.

6.2 Event passing

Another strategy is to record all operations made on a data set and send them to the other peers over the network. The central concept here is that no concrete data is sent, but instead a list with all edit operations. When a peer receives a list of operations, it just needs to apply them in a specific order on its local data sets.

All edit based synchronization strategies rely on *Operation Transformation (OT)*. emphOperational Transformation (OT) is a class of optimistic concurrency algorithms and

data structures that are well-suited to satisfying convergence, causality preservation and intention preservation [13].

Clarence Leung describes in [13] the three properties of the consistency model of collaborative editing systems as follows:

Convergence:
When the same set of operations has been executed at each site, then the copies of the document are also identical.

Causality preservation:
Given two operations O_A and O_B if $O_A \rightarrow O_B$ (O_A causally occurred before O_B), then O_A is executed before O_B at each site.

Intention preservation:
For every operation O, the intention of O at the initial site where O is initially submitted will be identical to executing O at all other sites. The intention of an operation O is defined as the resulting document which is achieved by applying O on the document state from which O was generated.

The challenges these sort of algorithms are facing is recording user actions. In rich modern user interfaces inside a browser-based environment this is very difficult, because of the big variety of possible user actions. Simple ones are normal text edits that are simple to observe, but it is more difficult in case of drag and drop, auto-correct, copy-paste, swipe, etc.

All these user actions must be turned into data transformations and then sent to all other peers over the network. The challenge is to generate the concrete data transformations for each of the possible user action.

In addition to this, the event passing algorithms are divergent. In other words, it is sufficient for a small operation to be applied incorrectly or to get lost in order to lead to an inconsistent and non-recoverable data model. Every further operation will increase the gap between the correct and the incorrect version. Altogether, a recovery mechanism must be designed for *Operational Transformation* algorithms. A good example where this kind of strategy is used and works good, is the Google Wave project.

6.3 Three-way merge

Three-way merges are very common in version control system implementations. As its name states, it works with three data versions: source, base and destination. The source is the data version received over the network, the destination is the local copy of the data and base is the common ancestor of the two (figure 6.1). The algorithm has following steps:

1. The base is compared with the source and the difference is computed.
2. The base is compared with the destination and the difference is computed.
3. The two difference sets are merged together and a new base version results.
4. The new base version must be taken over by both peers.

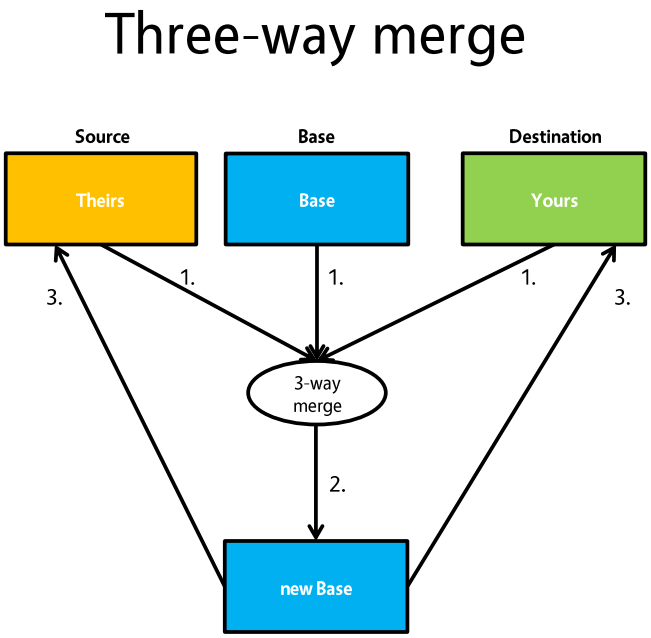


Figure 6.1: Three-way merge

The main disadvantage here is that the three-way merge algorithm is half-duplex. This means that if changes are made by a peer, while the 3-way merge synchronization is in progress, the new received version (new Base) can't be used anymore. In other words, while a user is typing, he can't receive any updates. When he stops typing, updates will arrive and be merged automatically or signal some conflicts that must be resolved manually.

In a real-time environment this is not suitable, because edits and updates can't happen concurrently. Such a solution is only feasible if the network speed and the text *diff* algorithm is faster then the user can type or if the user is willing to wait after each edit. Neil Fraser makes a very good analogy with driving.

Three-way merge is similar to "an automobile with a windshield which becomes opaque while driving. Look at the road ahead, then drive blindly for a bit, then stop and look again. Major collisions become commonplace when everyone else on the road has the same type of "look xor drive" cars." [1]

6.4 Differential Synchronziation

Differential Synchronization is a new symmetrical synchronization algorithm designed and explained by Neil Fraser in [1]. Its main advantage is that data differences are syn-

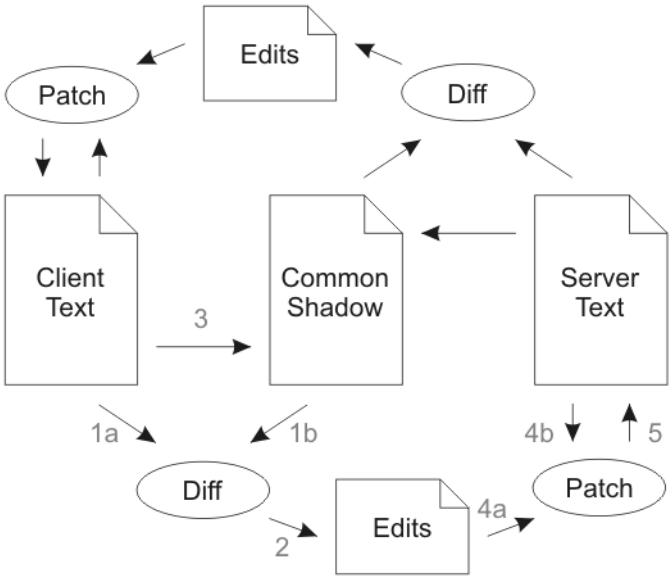


Figure 6.2: *Differential Synchronization* without a network (ref: [1])

chronized continuously across peers, without being forced to suspend the edit actions during the process. This is an important aspect that supports real-time collaborative editing applications. Figure 6.2 shows how two documents (*Client Text* and *Server Text*) can be kept in sync locally using *Differential Synchronization*. One of them is called misleading *Server Text*, even if figure 6.2 refers to a local peer. At the beginning the *Client Text*, *Common Shadow* and *Server Text* are identical.

When *Client Text* is modified the following steps are followed, as described in [1]:

1. *Client Text* is *diffed* against the *Common Shadow*.
2. This returns a list of edits which have been performed on *Client Text*.
3. *Client Text* is copied over to *Common Shadow*. This copy must be identical to the value of *Client Text* in step 1, so in a multi-threaded environment a snapshot of the text should have been taken.
4. The edits are applied to *Server Text* on a best-effort basis.
5. *Server Text* is updated with the result of the patch. Steps 4 and 5 must be atomic, but they do not have to be blocking; they may be repeated until *Server Text* stays still long enough.

Symmetrically changes to the *Server Text* can be applied to the *Client Text*. This algorithm guarantees that the peers will get into a consistent state after every change. It can happen that the patch from step 4 can't be applied, because the *Server Text* changed too much. In this case different options can be used: ignore patch, pop up a dialog and let the user choose, give preference to one side or the other, etc. The exact outcome of a collision does not matter in this algorithm, the key aspect is that whatever decision is taken, the system will go every time in a consistent state.

To make the steps more clear, the following example from Neil Fraser's paper [1] is depicted:

a *Client Text*, *Common Shadow* and *Server Text* start out with the same string: "Macs had the original point and click UI."

b *Client Text* is edited (by the user) to say: "Macintoshes had the original point and click interface." (edits underlined)

c The Diff in step 1 returns the following two edits:
@@ -1,11 +1,18 @@
Mac
+intoshe
s had th
@@ -35,7 +42,14 @@
ick
-UI
+interface
.

d *Common Shadow* is updated to also say: "Macintoshes had the original point and click interface."

e Meanwhile *Server Text* has been edited (by another user) to say: "Smith & Wesson had the original point and click UI." (edits underlined)

f In step 4 both edits are patched onto *Server Text*. The first edit fails since the context has changed too much to insert "intoshe" anywhere meaningful. The second edit succeeds perfectly since the context matches. Step 5 results in a *Server Text* which says: "Smith & Wesson had the original point and click interface."

g Now the reverse process starts. First the Diff compares *Server Text* with *Common Shadow* and returns the following edit:
@@ -1,15 +1,18 @@
-Macintoshes
+Smith & Wesson
had

h Finally this patch is applied to *Client Text*, thus backing out the failed "Macs" → "Macintoshes" edit and replacing it with "Smith & Wesson". The "UI" → "interface" edit is left untouched. Any changes which have been made to *Client Text* in the mean time will be patched around and incorporated into the next synchronization cycle.

In the previous example the failed patch was ignored and therefore the change was discarded, but the system got finally in a consistent state. Data loss can be avoided by letting the user decide, when a patch can't be applied.

In this form the *Differential Synchronization* algorithm can be applied just locally because it has just a common shadow. In a distributed situation the *Client Text* and *Server Text* are on different network peers and both need a *Shadow Copy*.

Consequently, two improvements can be made to the algorithm resulting in the *Dual Shadow Method* and the *Guaranteed Method*.

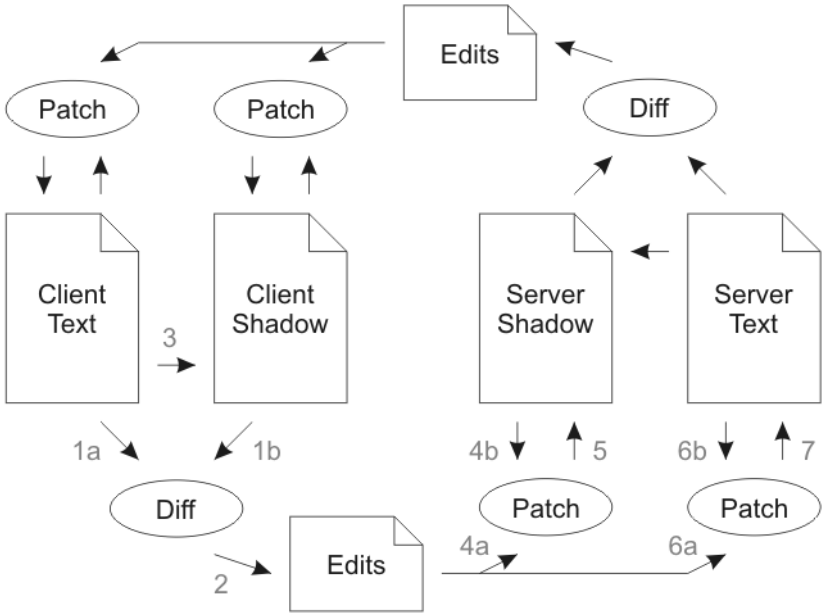


Figure 6.3: Differential Synchronization with shadows (ref: [1])

We will not discuss them in detail, but rather just present the central improvement points.

Dual Shadow Method:

As its name states, the main difference is that the common shadow is duplicated resulting in a *Client Shadow* and *Server Shadow* (fig. 6.3).

Important to note is that after every half of the synchronization the *Client Text* must be identical to the *Server Shadow* and the same applies for the *Server Text* and the *Client Shadow*. Theoretically the patches on the shadows can be applied every time without causing any collisions because the shadows do not contain any new changes.

The only problem the *Dual Shadow Method* has is potential data loss due to an unreliable network. Therefore *Client Text* checksums can be sent along the edits to ensure that the *Server Shadow* is identical with the *Client Text* after the patch. If the checksum fails then the two parts must be brought back in sync by copying the whole text from one part to the other. This action can imply data loss.

Guaranteed Delivery Method:

The final refinement resolves the problem of packet loss and is presented in figure 6.4. The server has additionally a *Backup Shadow* in case a previous transmission has not been received. The edits are tagged with the shadow version and stored in a stack. This is useful for saving edits locally in case of network connection failures. The shadow version number is updated every time it is overwritten by the text. The role of the version numbers is to avoid patching the same edits twice. A shadow will allow just patches with a greater version number.

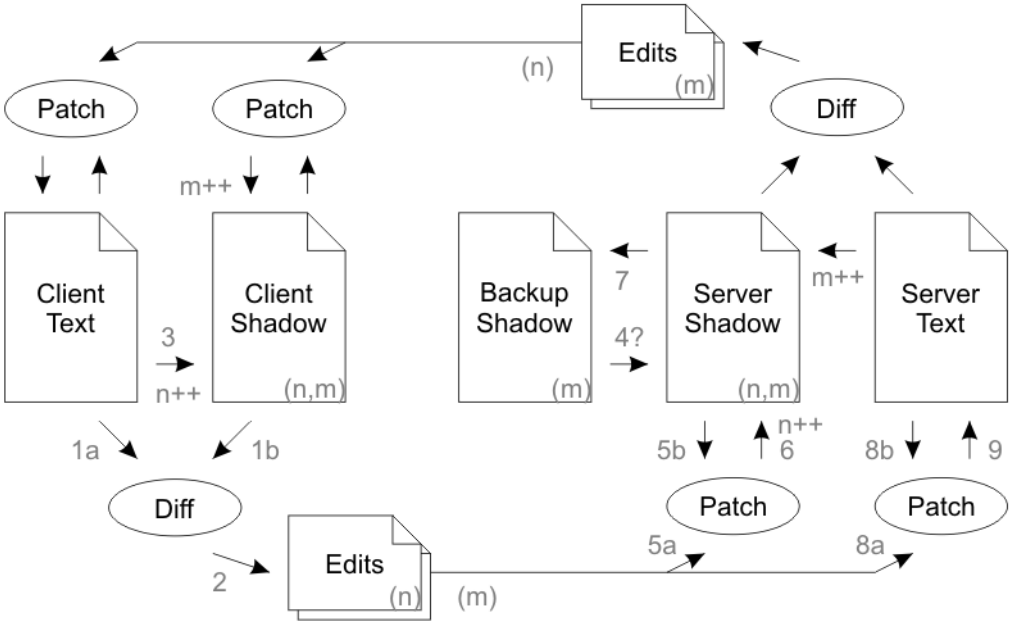


Figure 6.4: Differential Synchronization with guaranteed delivery (ref: [1])

Acknowledgments are introduced as a response to edits. A peer will try to send its stack of edits until it receives an acknowledge for a certain version number. If so, it can discard all the edits from the stack having an equal or smaller version number.

There are many different scenarios that can happen and are presented in detail in Neil Fraser's paper [1].

A last observation that is worth to be mentioned is the asymmetry of the figure. Why does the server have a backup shadow and the client none? The reason is the asymmetry of network connections in a web-based client-server configuration, as the client initiates every time the connection. Therefore the situation where the client data gets lost and the server data is received cannot happen, because the server responds just after a successful client request.

For symmetrical network architectures (peer-to-peer), which is also the case of the SyncLib architecture, all parties need a *Backup Shadow* because a connection can be initiated by anyone.

6.5 Synchronizing different data types

All the synchronization algorithms presented above, show how text fields can be synchronized. As presented in the SyncLib Data Meta Model, we need to deal with different field types.

Scalar Type
The scalar type can be number, string or boolean. For strings merges can be made, therefore *diff* algorithm is used as shown in the algorithms above. Numbers and booleans are easier to handle, because the old values are overwritten with the new ones. No merges are required.

Sequence Type
The sequence is an array of a scalar type. To synchronize arrays the observations from the scalar type apply, but additional information must be sent to identify which element of the sequence is referred.

Field Reference Type
The field references are pairs of object ids and field ids. Synchronization is identical to that of the sequence types.

Relationship Set Type
Relationship sets are sequences of object ids, in other words arrays of number types. This means that relationship sets are synchronized exactly like sequence types.

6.6 Local Value and Foreign Value

As discussed in the previous chapters, a field can be edited locally and at the same time receive updates over the network. Therefore every field has a local value and a foreign value. The local value is the one displayed into the UI and the foreign value is merged at different points in time with the local value.

Schematically it can be visualized in figure 6.5 like different branches. The foreign value is at the beginning equal with the local value until the first field update arrives. After each update it is merged into the local value. The figure illustrates also a snapshot. Such a snapshot is needed if, for example, a user opens a dialog and needs to fill out some fields.

At the end he has the option to submit the changes or cancel the dialog. So the changes should not be written directly into the local value, but rather in a new copy (snapshot). If the user submits them, then the values are merged with the local values and if not the snapshot is simply deleted. A snapshot is formed more exactly of duplicate objects. The local and the foreign value reside in the fields of the original objects.

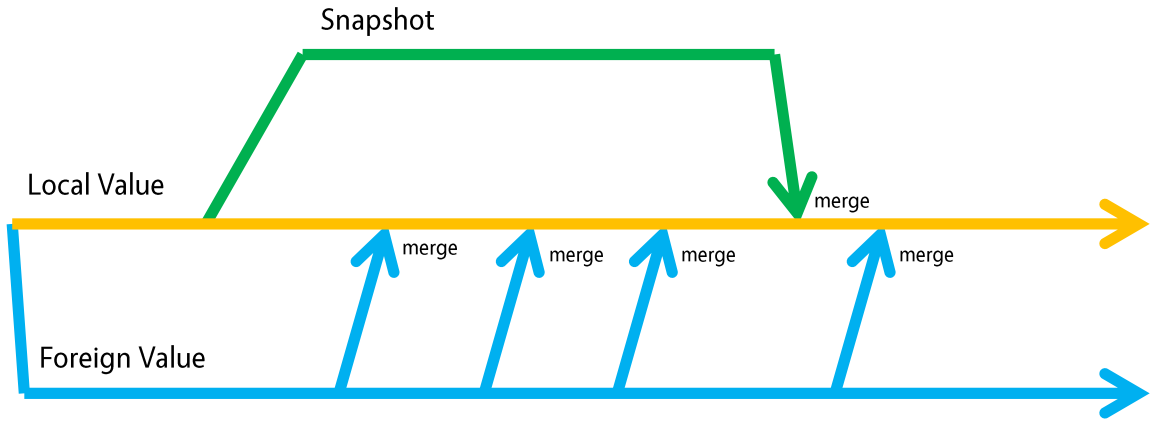


Figure 6.5: Local Value, Foreign Value and Snapshot branch representation

7 Synchronization Protocol

SyncLib is a library present on every network peer that is part of a software system. In order to keep the peer data models in sync during run-time, information must be exchanged by passing messages between relay and client and server peers, as described in the previous chapters. All the synchronization information is transmitted in form of messages, using the WebSockets network protocol.

On top of that, a SyncLib specific protocol must be designed that defines all the possible message types and their concrete structure. It must also be determined what the effect of each message is and if it requires a response.

7.1 Message Types

When designing the protocol messages, the main guideline was to keep the message number low. If the protocol has a lot of different messages, the complexity escalates quickly and failures are more likely to happen. A second point to consider is that the messages are asynchronous.

The following messages are used in the SyncLib protocol:

CONNECT

The CONNECT message is the first one that is sent when a peer wants establish a connection to the SyncLib relay.

It first negotiates the protocol. This is important because it can happen that the peer and the relay can have different protocol versions. The negotiation step has the role to check if the versions are compatible, or a fall-back is required to an older version. When receiving the connect message the relay must update his peer connection queue.

The message structure is as follows:

```
1 {
2     msgID           = "unique_msg_id",
3     name            = "CONNECT",
4     peerID          = "unique_id",
5     protocolVersion = "1.0",
6     applicationID   = "unique_applicationID"
7 }
```

Listing 7.1: CONNECT message structure

DISCONNECT

The DISCONNECT message signalizes that the peer closed the web application. The relay must therefore update its peer connection pool and discard all the peer data.

```
{
    msgID          = "unique_msg_id",
    name           = "DISCONNECT",
    peerID         = "unique_id",
    applicationID  = "unique_applicationID"
}
```

Listing 7.2: DISCONNECT message structure

INTERESTED

The INTERESTED message is sent by a peer right after CONNECT and has the role to notify the relay for what objects it is interested to receive updates. It is possible to express interest into an object type. This means that the peer will receive updates and meta-information for all objects belonging to that object type.

Another option is to enumerate a list of object IDs or more granular, pairs of object IDs and field names to express interest just in specific fields. An important aspect is that through interest messages it is possible to load new objects. If a peer expresses interest in one or more objects, the relay will transmit all the objects back.

In this way, simple object fetching can be made using INTERESTED messages, instead of queries.

```
{
    msgID          = "unique_msg_id",
    name           = "INTERESTED",
    peerID         = "unique_id",
    applicationID  = "unique_applicationID",
    objectType     = "object_type",
    objectID       = ["objectID 1", ...,"objectID N"],
    fields         = [ ["objectID 1","fieldName"],
                         ...,
                       ["objectID M","fieldName"]
                     ]
}
```

Listing 7.3: INTERESTED message structure

UNINTERESTED

The UNINTEREST message is sent by a peer if it doesn't need any updates or meta-information for object types, objects or fields. A typical situation is when an object is deleted.

```
{
    msgID          = "unique_msg_id",
    name           = "UNINTERESTED",
```

```
    peerID          = "unique_id",
    applicationID   = "unique_applicationID",
    objectType      = "object_type",
    objectID        = "objectID",
    fieldName       = "fieldName"
}
```

Listing 7.4: UNINTERESTED message structure

UPDATE

The UPDATE message is the most frequently exchanged between peers. Its main role is to transmit data changes. The client peers create the UPDATE messages and send them to the relay, which forwards them to the other interested peers. It is important to notice that more updates can be transmitted in a single message.

This feature is needed especially if sessions are used, as described in section 5.5.2. An update has first an object ID and a field name for identification. If the value type is not a string and as a result does not require a merging operation the new value is put directly in the update. In case of strings, a patch and shadow version must be transmitted for differential synchronization as described in section 6.4.

```
{
    msgID           = "unique_msg_id",
    name            = "UPDATE",
    peerID          = "unique_id",
    applicationID   = "unique_applicationID",
    updates = [

        [objectID        = "objectID 1",
         fieldName       = "fieldName 1",
         newValue        = "newValue",
         patch           = "diff_sync_patch",
         shadow_version  = "diff_sync_shadow_version"],

        ...,

        [objectID        = "objectID N",
         fieldName       = "fieldName",
         newValue        = "newValue",
         patch           = "diff_sync_patch",
         shadow_version  = "diff_sync_shadow_version"]

              ]
}
```

Listing 7.5: UPDATE message structure

REJECT

The REJECT message is sent always from the relay to a client peer and is sent as

a response to a received UPDATE message that cannot be accepted because the shadow version is obsolete. The REJECT message sends back a list of rejected updates. The client peer then knows that those updates are rejected and should not send them anymore. The relay will then send updates to the client peer after the REJECT message with the latest values and versions.

```
{
    msgID          = "unique_msg_id",
    name           = "REJECT",
    peerID         = "unique_id",
    applicationID  = "unique_applicationID",
    rejected_updates = [

        [objectID        = "objectID 1",
         fieldName       = "fieldName 1",
         newValue        = "newValue",
         patch           = "diff_sync_patch",
         shadow_version  = "diff_sync_shadow_version"],

        ...,

        [objectID        = "objectID N",
         fieldName       = "fieldName",
         newValue        = "newValue",
         patch           = "diff_sync_patch",
         shadow_version  = "diff_sync_shadow_version"]

              ]
}
```

Listing 7.6: REJECT message structure

METAINFO

The METAINFO message contains the payload for all meta-information notifications (“Local Life Cycle (LLC)”, “Server Peer Status (SPS)”, “Local Presence (LP)”, “Local Change (LC)” and “Foreign Change (FC)”). The message structure is simple. The type expresses which meta-information is considered, which object ID and which field name are used to identify the targeted field and at last the new state. If the meta-information targets an object then the field name is empty.

```
{
    msgID           = "unique_msg_id",
    name            = "METAINFO",
    peerID          = "unique_id",
    applicationID   = "unique_applicationID",
    type            = "metainfo_type",
    objectID        = "objectID",
    fieldName       = "fieldName",
    metainfo_state  = "metainfo_state"
}
```

Listing 7.7: METAINFO message structure

ACKNOWLEDGE

ACKNOWLEDGE (ACK) is used as a response to every message presented above, in order to confirm that it was received successfully. If a peer does not receive an ACK after it sends a message, then it resends it at precise intervals.

```
1 {
2     msgID = "unique_msg_id",
3     name  = "ACK",
4 }
```

Listing 7.8: ACKNOWLEDGE message structure

7.2 Synchronization Protocol Use Cases

In this section two use cases are presented in order to exemplify the interaction between two client peers. The figures are conceived to illustrate the user interface changes, meta-information states and protocol messages in order to create an overall image. Both scenarios present just two client peers and a relay.

The protocol messages are represented by black arrows and the annotations contain just the type of the message. The full message body, as described in the previous section, could not be represented due to the lack of space and in order to keep the figures visually understandable.

The messages are numbered in order to clarify the transmission order and to pair up the messages with their corresponding ACK response. In each step the UI is shown, the meta-information abbreviations (see section 3.1) with the values for the text box, its current value, the update value received over the network (technically just some string differences are transmitted and patched, but for clarity we display the whole values), and the new value after the merge operation.

The figure below illustrates one of the simplest scenarios where one peer(A) interacts with its user interface and modifies data. The other peer(B) is just viewing, without changing anything in its UI. Next, each message will be described together with its cause and effect:

1. INTERESTED: At the beginning client peer A doesn't have the text box data loaded, what is also reflected in its "Local Presence" state that is *Missing* (LP=M). Peer B has already loaded the text box data. In order to get data, peer A must send an INTEREST message to the peer, indicating in its body the object ID and the field name, as shown in the previous section. The relay will then fetch the data from a server peer (not shown in the figure) and send the data in the subsequent UPDATE message.
2. UPDATE: Peer A receives the text box data with this message and displays it in the UI. It can be noticed that the meta-informations changed now. "Server Peer

Status" is set to *Server Peer Fully Synced*, "Local Presence" is *Present*, "Local Change" and "Foreign Change" are *Clean* and "Local Life Cycle" is *Complete*.

3. METAINFO (LC=PD): The user selects th text box and the cursor becomes visible. This action makes the "Local Change" to switch to *Prepared Dirty* (PD), what means that the user is probably going to modify the text box data. This information must be sent to the other interested peers in order to prevent conflicts. As a result a METAINFO message is sent indicating in its body that "Local Change" changed to *Prepared Dirty*.
4. METAINFO (FC=PD): This is the same message as the previous one, but the metainfo type has been changed by the relay to "Foreign Change„ and forwarded to client peer B. When client peer B receives the message, it switches its "Foreign Change" to *Prepared Dirty* and the text box in the UI turns yellow in order to warn the user.
5. METAINFO(LC=D): Client peer A has modified the text box value to "Computer Apple" and as a result the "Local Change" switched to *Dirty*. This information is that sent to the relay in a METAINFO message.
6. METAINFO(FC=D): The METAINFO message is forwarded to the other peer with "Local Change" transformed into "Foreign Change". Peer B changes therefore the "Foreign Change" to *Dirty* and the text box turns red, indicating that any change could cause a conflict.
7. UPDATE: The user changes must be synchronized with the other peer. Therefore an UPDATE message is sent with the patch to be applied on the other peer. Besides the textbox, three dots appeared in order to signalize the user that the changes are still under transmission.

   After client peer A received the ACK for the UPDATE message from the relay, the three dots are replaced by a check mark, indicating that a server peer received and persisted the new value. "Local Change" switches then back to *Clean* and "Server Peer Status" changes to *Server Synced*.
8. UPDATE: The UPDATE message is subsequently sent to peer B, which patches it and gets back in sync.
9. METAINFO (SPS=SPFS): Once the relay received the ACK from peer B, it knows that every peer received the update from peer A. Therefore, it sends back a METAINFO message that will change the "Server Peer Status" into *Server Peer Fully Synced* and the check mark disappears.

The next figure presents a slightly more complex scenario where both peers interact with their user interface and modify data. This time both peers start with the text box data already loaded and in sync. Next, each message will be described together with its cause and effect:

1. METAINFO (LC=PD): As in the previous example, peer A selects its text box and a message is sent to the relay to notify the other peers.
2. METAINFO (FC=PD): The relay replaces "Local Change" with "Foreign Change" and forwards the message. When peer B receives it, it updates its "Foreign Change" and the text box turns yellow as a warning.

3. METAINFO (LC=D): Peer A starts typing and therefore the “Local Change” switches to *Dirty* and a corresponding message is sent to the relay. The “Server Peer Status” switches also to *Not Server Synced*, because the peer is now out of sync.
4. METAINFO (FC=D): Peer B receives the METAINFO message, updates its “Foreign Change” and the text box turns red.
5. METAINFO (LC=PD): Peer B selects the text box, what automatically changes the “Local Change” to *Prepared Dirty* and a METAINFO message is sent out in order to signalize the other peers that peer B will probably change the field value.
6. METAINFO (FC=PD): Peer A receives the message, updates its “Foreign Change” and the text box turns yellow.
7. METAINFO (LC=D): Peer B ignores the red warning and starts typing. The “Local Change” switches to “Dirty” and a METAINFO message is sent therefore immediately to the relay.
8. METAINFO (FC=D): Peer A receives the METAINFO message, updates its “Foreign Change” and the text box turns red. At this stage both peers changed locally the text box value and are out of sync.
9. UPDATE: Client peer A sends out first an UPDATE message with his changes and the three dots appear beside the text box. Only when the ACK is received from the relay, a check mark appears to signalize that the changes are persisted.
10. UPDATE: Peer B sends its UPDATE message a bit later and because the server could not merge its changes with the new version, a REJECTED message is sent back instead of an ACK. The “Local Change” switches to *Rejected* and this is signalized through the exclamation mark.
11. METAINFO (LC=PD): The user at peer A finished typing, but the text box remains still selected. A METAINFO message is therefore sent to the relay.
12. METAINFO (FC=PD): Peer B receives the message and the text box turns from red into yellow.
13. UPDATE: The relay sends the UPDATE message with the changes from peer A to peer B.
14. METAINFO (LC=D): Peer B merges his value with the received one. Subsequently the “Local Change” will switch to *Dirty* and a METAINFO message is sent out.
15. UPDATE: Peer B sends the new text box value after the merge to the relay.
16. METAINFO (LC=C): Because peer B has received an ACK for the UPDATE message and its text box no longer selected the “Local Change” switches to *Clean*.
17. METAINFO (SPS=SPFS): The “Server Peer Status” of peer A switches to *Server Peer Fully Synced* because peer B received his update.
18. METAINFO (FC=C): The relay observes that it received two METAINFO messages from peer B, one with *LC=D* and another with *LC=C*. Because it didn’t sent neither of them until now, it ignores the first one and sends just the second one. When peer A receives the message, its “Foreign Change” switches to *Clean* and the text box returns to the default color again.

19. UPDATE: The relay sends the UPDATE with the new value from peer B to peer A and it can be merged automatically with the current value.
20. METAINFO (LC=C): This METAINFO message is sent because peer A has no longer its text box selected.
21. METAINFO (FC=C): Peer B receives the message, updates its “Foreign Change” and the text box returns also to the default color.
22. METAINFO (SPS=SPFS): Because the relay received an ACK for the UPDATE message (19) sent to peer A, it notifies peer B and the “Server Peer Status” state switches to *Server Peer Fully Synced*. The check mark disappears accordingly.

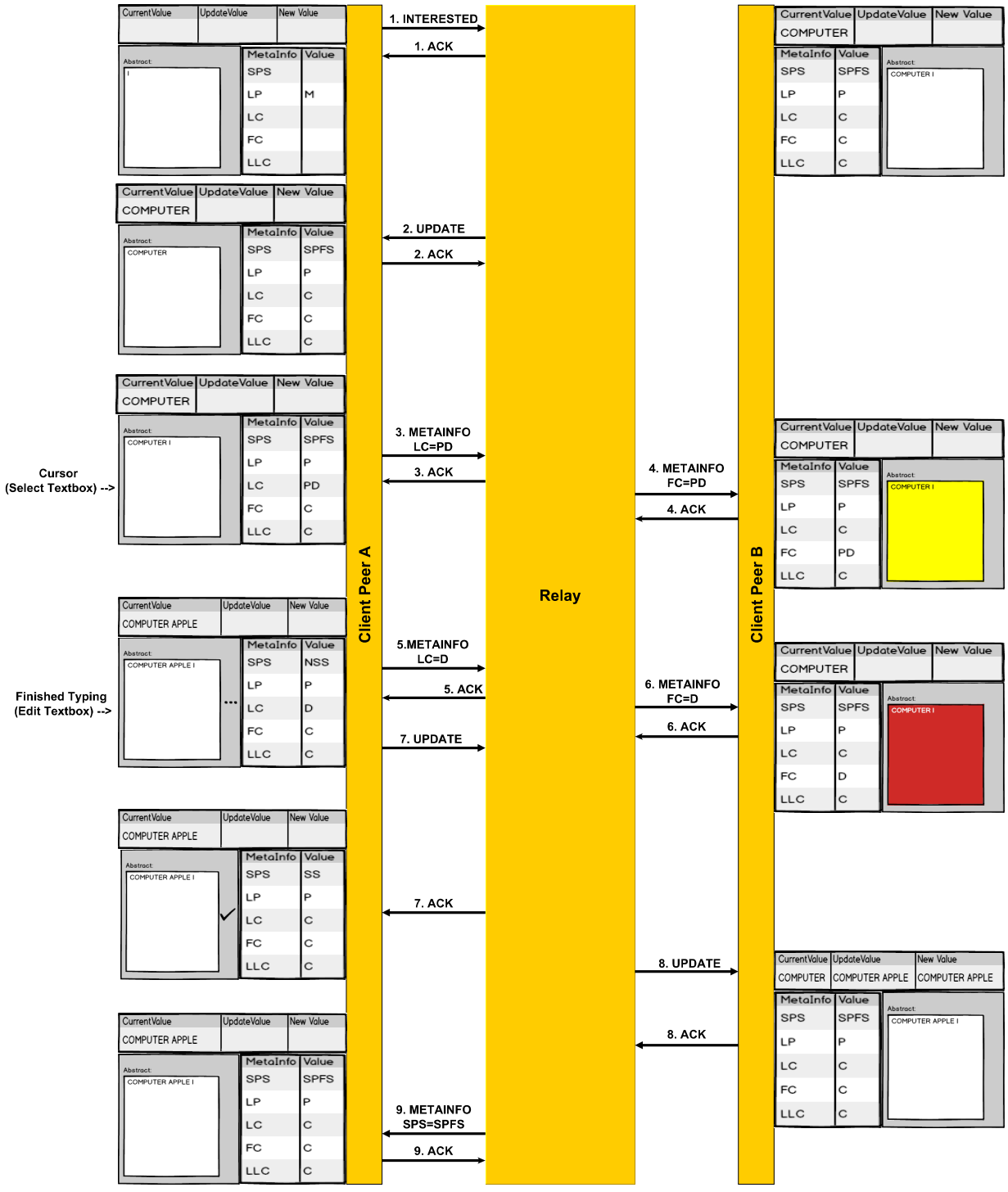


Figure 7.1: Synchronization Protocol Use Case 1

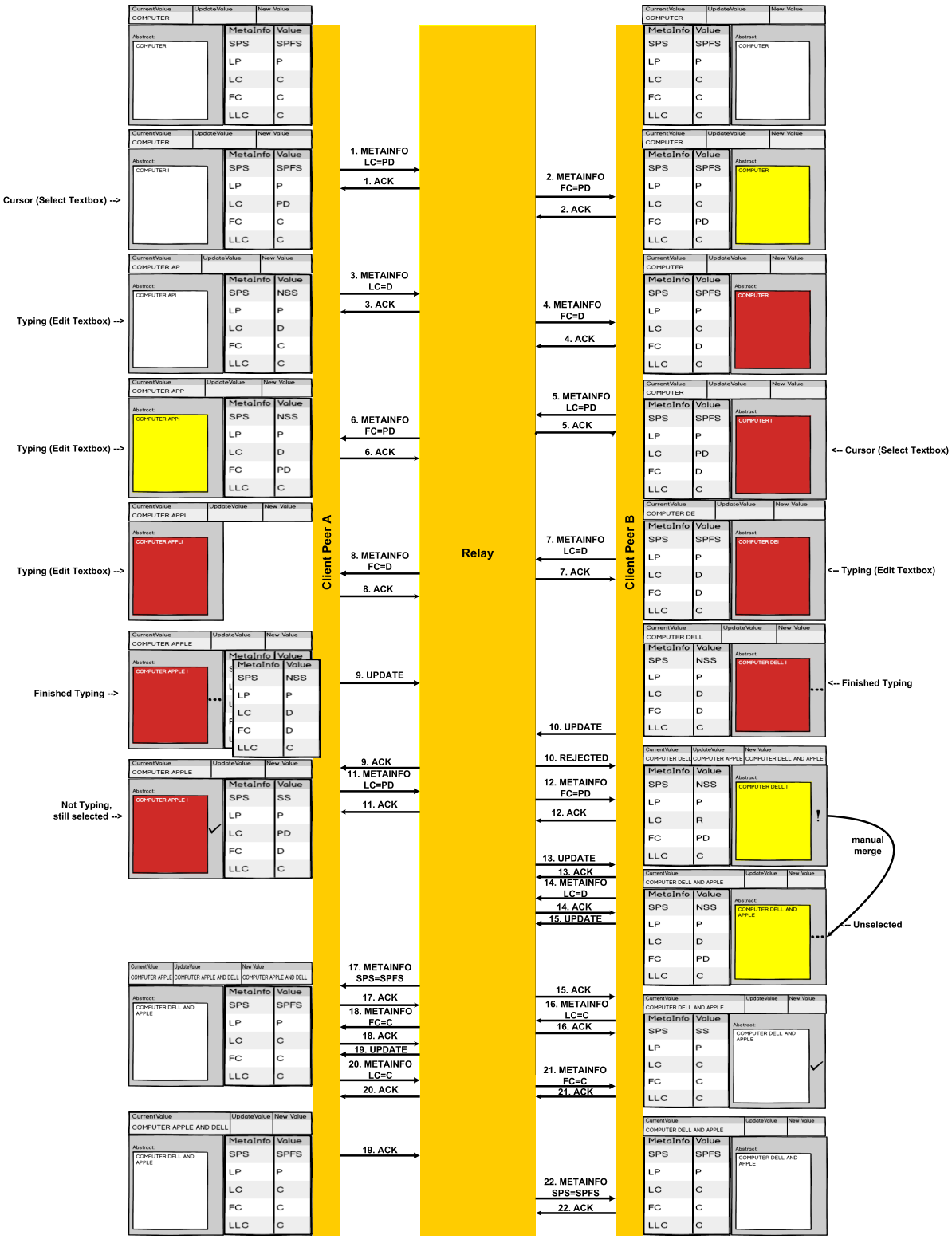


Figure 7.2: Synchronization Protocol Use Case 2

8 Conclusions

8.1 Summary

We showed a solution that could make real-time collaborative web applications possible. The main challenge was to keep the continuously changing peer data models in sync during run-time. This involved designing a library (*SyncLib*), that is deployed on every system peer. The main features included meta-model definition, object creation and management, meta-information processing, data transmission and conflict resolution. All these were conceptually and visually integrated in a *SyncLib Data Meta Model*.

The next aspect covered was an architectural system overview, composed of SyncLib driven peers. First we identified system components, together with their roles. Subsequently, in order to point out how they interrelate, we presented various architecture topology solutions, that meet different common enterprise requirements like multiple servers for scalability and management of numerous deployed applications. The next step was then to zoom in on each component and describe the internal architecture using functional views. An interesting point here was to analyze how the SyncLib library can be integrated in already existing systems.

After the architectural aspects were covered, we moved to the synchronization operations, describing how the *SyncLib* internally works and can be used. First, the effects of simple object *CRUD* operations were described, moving then forward to slightly more complicated concepts like *Object Sets*, *Ephemeral Objects* and *Queries*. The most notable concept here was how queries can be automatically reevaluated, based on triggers, resulting in a continuously up to date UI, without needing any data refresh requests.

Meta-informations were also an important aspect that contributed to conflict prevention and visual user collaboration support. Therefore, each of them was visually illustrated by state diagrams, describing what actions trigger the meta-information value changes.

Another challenge was to find synchronization strategies that can handle problems like data conflicts, data edit package losses and duplicate data edit packages, taking at the same time the real-time environment into considerations. The best solution found was Neil Frasers *Differential Synchronization* algorithm, using the *Guaranteed Delivery Method*.

Finally, we described the *SyncLib Synchronization Protocol* that defines all the messages exchanged between SyncLib peers. In order to exemplify how the messages correlate with the user actions and meta-information changes, we presented subsequently two representative scenarios.

8.2 Future work

Even though we set the conceptual base of a data synchronization library that can lead to a new kind of web applications, a reference implementation is still needed in order to transpose all the theoretical aspects into practice. The limited time available for elaborating this thesis was not enough to develop one, due to the complexity of the concepts that had to be conceived and then integrated into a single solution. There were often situations when individual concepts solved problems very good, but when put together, unpredictable difficulties arose, requiring to rethink them.

Content-related, new meta-informations can be added for example to enable the users to see each others cursors, or informing them exactly which users received the changes or are currently modifying data. Another point to consider is that the peers application functionality can be technologically heterogeneous. This means that the SyncLib needs to have an implementation for the most used languages (JavaScript, Java, C#, Phyton, Ruby, etc.). Additionally, in order to define the business data model just once, a Domain Specific Language (DSL) can be designed from which the concrete *ObjectTypes* can be generated for the desired languages.

Queries were discussed in section 5.5 where we presented different solutions. *Queries with Dynamic Condition* suppose an additional expression that describes the query conditions. A syntax and semantic must be further defined for that query expressions.

Furthermore, for the *Add-in SyncLib Application* option described in section 4.5, a data versioning mechanism must be determined so that changes can be identified also for objects created or updated from outside the SyncLib. A first idea would be to set an object-level *ETag (Electronic Tag)* that represents a hash of the object payload. By transmitting with each UPDATE message the old and new *ETag*, we could determine if the changes are based on the newest data version or not.

Additionally, authentication and authorizations are still two open points that need to be elaborated in the SyncLib context. For this the relays *Interested List* needs to adhere to the user authority rules, in order to grant each user access just to the data it is allowed to access.

For security reasons the business logic must have the chance to validate the data, before it gets synced with the data model. For this, a concept similar to the Java Enterprise Edition (Java EE) interceptors needs to be designed.

Last, the presented *Object Set* and *Session* should be reviewed, in order to see if a third concept can be found to group objects together, that is flexible and feels at the same time straight-forward from the API (Application Programming Interface) point of view.

Bibliography

[1] Neil Fraser. Differential synchronization, 2009.

[2] Ian Fette and Alexey Melnikov. The websocket protocol, 2011.

[3] Ralf S. Engelschall. Architecture patterns:components, 2013.

[4] Ralf S. Engelschall. 3-tier application component architecture, 2014. URL http://engelschall.com/go/EnTR-02:2014.01.

[5] Jim Gray. The transaction concept: Virtues and limitations, 1981.

[6] Theo Härder Andreas Reuter. Principles of transaction-oriented database recovery, 1983.

[7] YousefJ. Al-Houmaily and George Samaras. Two-phase commit, 2009. URL http://dx.doi.org/10.1007/978-0-387-39940-9_713.

[8] Dr. Eric A. Brewer. Towards robust distributed systems, 2000. URL http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[9] Charles Roe. Acid vs. base: The shifting ph of database transaction processing, 2012. URL http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/.

[10] John Cook. Acid versus base for database transactions, 2009. URL http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/.

[11] Transactional semantics and forked records, 2013. URL http://epf.io/getting_started.html.

[12] Patrick Valduriez M. Tamer Özsu. Principles of distributed database systems, 2011.

[13] Clarence Leung. Operational transformation in cooperative software systems, 2013.