An architecture for data synchronization in a mobile environment

Bouwmans, P.F.M.

Award date:
2009

An architecture for data synchronization in a mobile environment

P.F.M. Bouwmans
August 2009

Abstract

This thesis describes an architecture for data synchronization in a mobile environment, optimizing the cost for communication, processing time and consistency. The research is based on problems encountered in the Connect-It application: several handheld computers communicate via a wireless GPRS connection with a central server. Limited bandwidth, limited connectivity and expensive data transport are the major problems the application has to cope with. The solution presented in this thesis provides an architecture that guarantees consistency and reduces the bandwidth usage and processing time at the client side.

Acknowledgements

There are several people I'd like to thank for their help and support. Without them, I couldn't have written this thesis.

- J.J. Lukkien, supervisor at the Technische Univerisiteit Eindhoven. His knowledge and feedback has been invaluable.
- A. Kuindersma, supervisor at ViaData Heerenveen. His in-depth knowledge of Connect-It has helped me a lot.
- J. van der Woude, mentor at the Technische Universiteit Eindhoven, for believing in me.
- My parents, brother, girlfriend and other family, for unconditional love and support during my whole life.
- My friends, for being there when I need them.
- TOPIC Embedded Systems, for financial support and patience while I was working on this thesis.
- Océ Technologies R&D, for a challenging and motivating work environment.

Contents

Chapter 1

Introduction

This thesis is structured as follows. In this chapter the problem is introduced. In chapter 2 the problem is analyzed and compared to existing problems and solutions. In chapter 3 the new architecture and algorithms are presented. Chapter 4 analyzes the results. It ends with a conclusion and remarks about future work in chapter 5.

Section 1.1 introduces the application on which the research in this thesis is based. The current architecture is briefly described in section 1.2 (section 2.3 goes into detail about the current architecture). Section 1.3 defines a model of the application and subsequently in section 1.4 the problem is stated. Finally, in section 1.5 a summary of this chapter is given.

1.1 Background

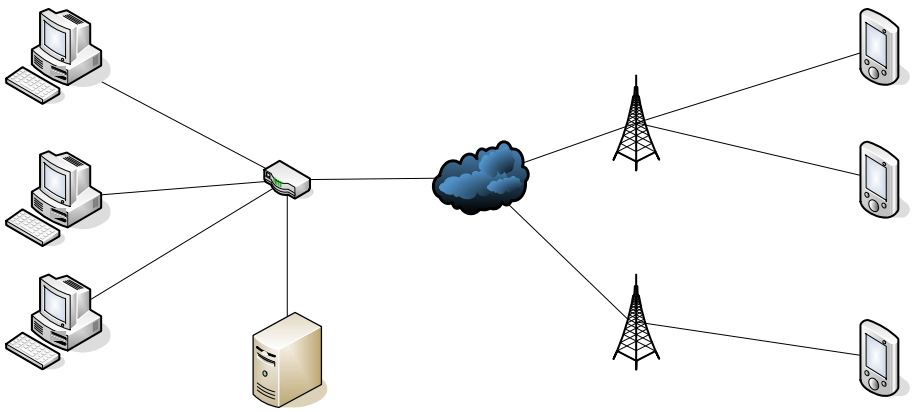

Figure 1.1: Overview of the system

The research in this thesis was the result of a problem in the application Connect-It. Connect-It is developed by ViaData in Heerenveen. It is a software solution that uses different network infrastructures (see figure 1.1) and allows digital work orders to be imported from an enterprise resource planning (ERP) system, integrate them in a schedule and finish them on a PDA. When they are finished they can be exported again to an ERP system.

Mechanics and service engineers travel to customers to repair machinery and other equipment or to conduct periodic maintenance on them. They receive their schedule with customers and work orders on a PDA. They often have to fill out forms, which they also do on the PDA. A portable bluetooth printer allows them to print the form while the PDA sends them back to the back office. On their PDA they can also bring up the history of the serviceable objects to see what repairs have been done in the past. At the end of the day, they can return home without going to the office first.

Connect-It is implemented as follows: several back office computers are connected via a local area network (LAN) to a central server. Employees can import data from the ERP into the Connect-It back office application, view and edit the planning of the work orders, etc. Customers can log in to a web site at the server to view the progress of their orders. The PDAs use a wireless GPRS internet connection to communicate with the same server. The GPRS connection might not be always available, network coverage is not guaranteed. This results in each PDA having a local data storage, which consists of a small part of the total data available. This local data storage ensures the employee has all required data available to him even when there is no connection to the server readily available. Once every while this local data storage is updated to keep it consistent with the data at the server. When there are changes made at the PDA, the data is transferred back to the server as soon as possible.

1.2 Architecture

Most existing software applications that distribute data between a server and several clients don't work on the PDA platform. Therefore, ViaData designed and implemented their own software to do this. It is currently implemented in Microsoft C# .NET 1.1, but a transition to Microsoft C# .NET 2.0 will be made in the near future. All software implementations that are the result of this thesis will therefore be made in C# .NET 2.0. As a side effect, experiments can only be tested in a controlled environment instead of a real world environment. While testing in a controlled environment is a good thing, real world data would also be valuable.

The software architecture of the Connect-It system consists of a server application and multiple client applications. The clients communicate with the server via a web service interface [6]. The reason behind this, is that the server is probably located behind several routers and firewalls. The client application can also be behind a firewall. As clients are allowed to access websites by default and the web server uses the standard port 80, no additional configuration or maintenance of routers or firewalls is necessary. Most customers who buy the Connect-It application don't have a big IT department, if they have one at all, so this is mainly commercially motivated. For the developers it is also easy to implement: the web service is generated by the services provided by the application logic, allowing the clients to call the methods provided by the server application logic without any difficulty. Although the W3C web service definition encompasses many different systems, in this case the most common usage is meant: communication over the HTTP protocol using XML messages that follow the SOAP [4] standard.

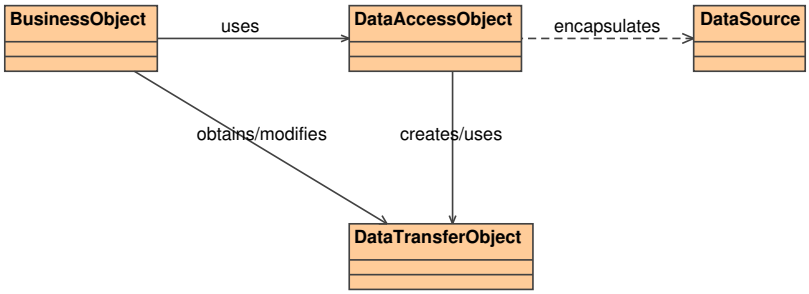

Figure 1.2: DAO pattern class diagram

The data layer is implemented using the Data Access Object (DAO) design pattern [2], it provides an object oriented interface to work with the data. It also separates the higher level functions like saving the data from the specific type of data storage used. This way, it is possible to change data stores without influencing the application logic itself. Possible data storages are a SQL server or a comparable database, a set of XML, text or binary files (in any format), or even a shared data space. Note that for each type of data storage, a separate data layer (using the DAO pattern) needs to be developed.

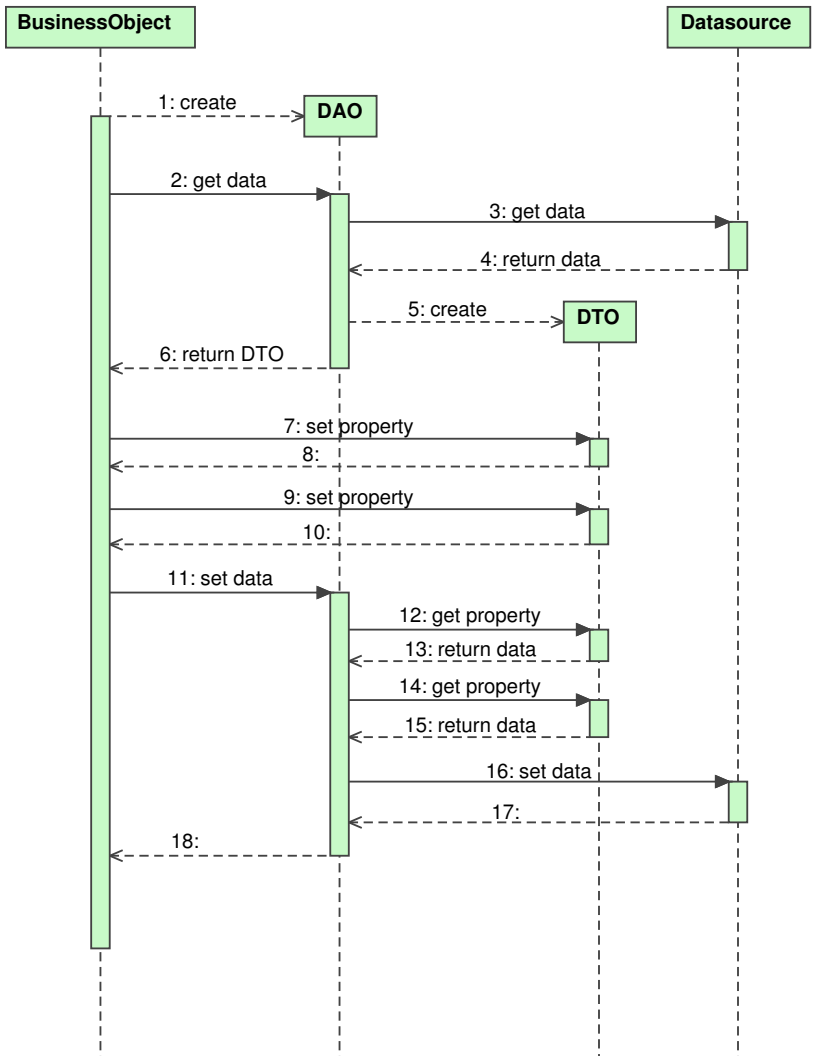

Figure 1.3: DAO pattern sequence diagram

Figures 1.2 and 1.3 show the DAO design pattern. The different participants and their responsibilities are:

- **BusinessObject**
  The BusinessObject represents the client that requires access to the data source to obtain and store data.

- **DataAccessObject**

The DataAccessObject (DAO) abstracts the data source and its access implementation for the BusinessObject to enable transparent access to the data source.

- **DataTransferObject**
  The DataTransferObject (DTO) is used as a data carrier. It is used to communicate the data between the BusinessObject and the DataAccessObject.
- **DataSource** The DataSource can be anything from a SQL database to a set of text files.

The client application also uses a DAO Layer to be able to work with the data as objects, while keeping the details of a specific data storage hidden. The application logic communicates with the server to send and receive changes in the data. Every once in a while, between 5 minutes and 4 weeks depending on which data type is being updated, the client deletes all local data of that type and requests the new data from the server.

In section 2.3 a more detailed analysis of the architecture is shown.

1.3 Model

To be able to define a (mathematical) problem or requirement within Connect-It, a mathematical representation of the Connect-It system is necessary. Designing the following model of the new system behavior was also a part of the assignment (see requirement 1 from section 1.4). No previous models exist and the current system behavior is unclear and not transparent at all.

N is a set of names, V is a set of values. A state in the system is a function $f : N \to V \times \mathbb{Z}^+$ and can be written as $f(n) = \langle v, t \rangle$ where $n \in N$, $v \in V$ and $t \in \mathbb{Z}^+$. Here, t represents a timestamp and can be an integral version counter or a real time. For the rest of this model it is assumed that t is an integral version number.

The state of the server is s and the state of client i is c_i. The invariant of the system is $[dom(c_i) \subseteq dom(s) \wedge \forall n \in dom(c_i) : c_i(n) = s(n)]$.

Each client i has a buffer state b_i. Note that b_i has no direct relation to s or c_i: it can contain different names, and values of existing names don't have to be equal. The buffer state represents data that a client wants to transmit to the server.

The behavior of the system is:

$(((Change)^*; Change; Commit)^*; Refresh)^*$

The next part will clarify on this. Note that I use the following syntax:

- $f(n) \leftarrow \langle v, t \rangle$ indicates that the mapping of n in f changes to $\langle v, t \rangle$.
- $f(n).v$ returns the value v where $f(n) = \langle v, t \rangle$.
- $f(n).t$ returns the timestamp t where $f(n) = \langle v, t \rangle$.
- $dom(f) \leftarrow D$, where $D \subseteq N$ indicates that the domain of f changes: new mappings are created or existing ones are removed.
- $time()$ returns a new unique timestamp which is larger than all previously created timestamps.

Change Client i can change the value of one of its names, or add a new name. These modifications are done on a the buffer state b_i. Because c_i and s don't change, this operation has no influence on the invariant.

Change(c_i, b_i, n, v)
if $n \notin dom(b_i)$ **then**
{There is no mapping of n, add n to $dom(b_i)$}
$dom(b_i) \leftarrow dom(b_i) \cup \{n\}$
end if
if $n \notin dom(c_i)$ **then**
{The name n is a new name, created by client i}
$b_i(n) \leftarrow \langle v, 0 \rangle$
else
{The name n already exists, use the existing timestamp}
$b_i(n) \leftarrow \langle v, c_i(n).t \rangle$
end if

Commit This operation is always preceded by a change. *Commit* takes a buffer state b_i and applies the changes in it to the server state s. *Commit* should guarantee that all changes are applied in one atomic operation that either fails or succeeds.

Commit(s, b_i)
for all $n \in dom(b_i)$ **do**
if $n \notin dom(s)$ **then**
{There is no mapping of n, add n to $dom(s)$}
$dom(s) \leftarrow dom(s) \cup \{n\}$
{Update the value of $s(n)$ and create a new timestamp}

$s(n) \leftarrow \langle b_i(n).v, time() \rangle$
else
{There is already a mapping of n, check if there is a conflict}
if $s(n).t \neq b_i(n).t$ **then**
{The timestamps don't match, there is a conflict}
ResolveConflict(n, s, b_i)
{The conflict has been resolved}
else
{There is no conflict, update the value}
$s(n) \leftarrow \langle b_i(n).v, time() \rangle$
end if
end if
end for
{Everything succeeded, clear b_i}
$dom(b_i) \leftarrow \emptyset$
$codom(b_i) \leftarrow \emptyset$

When there is a conflict, there are two options to resolve it:

- Overwrite the existing value on s
- Discard the value from b_i

ViaData wants this conflict to be resolved by human intervention. There are only a few possibilities for the reason of this conflict:

- Multiple employees are working on the same order data
- A PDA was shut down before the data was submitted to the server and work has begun on a new PDA

The first possibility will never happen, if there are multiple employees working on the same order, they only use one PDA. The second possibility will probably only occur when a PDA has crashed, or ran out of battery power. The data that hasn't been submitted yet could contain the correct value for a certain name, or it could contain an outdated value. Which one is the correct case is not detectable by the software, therefore human interaction is required. This can be done by a back office employee, the employee that controls the PDA, a system administrator, or another person with enough knowledge and access to the data.

This analysis shows a possible consistency error: if other data relies on the value that was discarded, the system might loose consistency. For example: if the total price of an order is stored somewhere, and the order

data might change afterwards, the total price that is stored might not be the correct price. When manually resolving a conflict, these factors have to be taken into account. To find a solution for this problem is outside the scope of this thesis.

Because *Commit* changes the server state and not the client states, the invariant doesn't hold anymore. A $Refresh$ is necessary to make the invariant hold again. Because the client doesn't know when other clients perform a *Commit*, each client will periodically perform a $Refresh$. The result will be that the system as a whole will adhere to the invariant.

Refresh This operation is always preceded by a *Commit* if there were changes done to make sure the buffer state b_i is empty and all changes have been submitted to the server. The operation $Refresh$ checks for inconsistencies between c_i and s, and updates c_i accordingly.

Refresh(s, c_i)
for all $n \in dom(c_i)$ **do**
{No names can be removed from $dom(s)$, the name n exists in s if it exists in c_i}
if $s(n).t \neq c_i(n).t$ **then**
{There is a newer value of name n on the server, update c_i}
$c_i(n) \leftarrow s(n)$
end if
end for

Because b_i is empty, no conflicts can occur during $Refresh$. When $Refresh$ is finished, the invariant holds.

To summarize, these assumptions were made:

- There is no parallelization on the server, all client requests are handled one by one.
- After every $Refresh$ for client i, the invariant holds for that client.
- All changes made by a client (changing an existing value, or adding a new name) are done on b_i.
- New names are unique systemwide.
- During $Refresh$ for client i, b_i is empty.

1.4 Problem statement

The model described in 1.3 is the required model, the current architecture and algorithms do not fulfill this model. Connections to the server are made at many places in the PDA application and it is not clear how the system as a whole operates. In addition to this, there are suspicions that the synchronization of the clients with the server uses much more bandwidth than necessary. To address all problems, several requirements can be defined:

1. Design a model of the new behavior of the Connect-it system (see section 1.3).
2. Change the architecture in such a way that all communication is being done in separate components.
3. Change the architecture in such a way that changes in the application logic don't require changes in the new communication components.
4. Change the architecture in such a way that changes in the type of data storage used or changes in what data is stored don't require changes in the new communication components.
5. Minimize the bandwidth usage for synchronizing the clients with the server: $(bu(Commit) + bu(Refresh))$ where bu is a function that determines the bandwidth usage of a certain operation.
6. Impact on battery usage should be minimal: minimize the client processing time for synchronizing the clients with the server: $(pt(Change) + pt(Commit) + pt(Refresh))$, where pt is a function that determines the processing time of a certain operation.
7. The number of clients can change over time.

1.5 Summary

In this chapter the Connect-It application has been introduced. As Connect-It started small and has grown over the years, it's original architecture is not sufficient anymore. The exact behavior of the system is also unclear.

Chapter 2 will analyze Connect-It as it is. Chapter 3 will provide a new architecture that will satisfy the model described in section 1.3. Chapter 4 will analyze the results and a final conclusion is given in chapter 5.

Chapter 2

Analysis

In this chapter the problem introduced in chapter 1 will be analyzed in more detail. First, in section 2.1 an overview is given of related work in the field of distributed systems. In section 2.2, the challenges of designing a distributed system are explored and how they apply to Connect-It. A detailed analysis of the current architecture is given in section 2.3. Section 2.4 provides a method to compare different distribution protocols and the relation between the different parameters is studied. A brief comparison between several existing distribution methods is given in section 2.5. Finally, section 2.6 gives a summary and some thoughts about how to proceed.

2.1 Related work

The research field of distributed systems includes many different topics. In the following subsections, a brief overview is given of those research topics and their relation to Connect-It.

2.1.1 Caching on a failed hit

One method to locally store information, is to cache on a failed hit. The application requests a certain piece of data, if it is not locally available, it is requested from the server and stored locally. A second request can be served from the local cache. The caching of web pages is commonly done using this method, but it is not restricted for use with web pages. In [9] a framework is described for cache management for mobile databases, based on this caching scheme.

The major drawback is that it only caches items once there has been a request for it. If there is no connection possible at that time, the request simply fails. In the

case of Connect-It, the data should be available when the application requests for it and thus it needs be distributed before the first request.

2.1.2 Invalidation

At some point parts of the data aren't up to date anymore. There are several methods to ensure the data is up to date (combinations are also possible [22]):

- If Modified Since (IMS): when the application requests a data item, the client checks at the server if the item has been changed since last time he checked for it. If this is the case, the server sends the new item to the client. Otherwise, the client can use the locally stored item. Proxy servers often utilize this strategy.
- Time To Live (TTL): the client request data from the server. After it received the data, it assumes the data doesn't change during a window w. After that it simply deletes the data.
- Invalidation Reports (IRs): the server broadcasts IRs to the client, which contain the identifiers of the data that isn't up to date anymore. It is up to the client to request a new copy from the server when it wants an updated version. This does require a lot of administration at the server if not all clients have the same dataset, in that case the server has to keep track of which client has what data stored.
- Lease: the client registers a lease on data item x. During the lease time, it receives invalidation reports from the server. At the start of a lease the client either receives the item from the server, or the client can use a local copy in combination with an IMS request.

These methods are not very useful for Connect-It, as they require either a constant connection to a server (IRs and Lease) or a connection at the time the data is requested from the server (IMS and TTL). Missing an IR can be crucial and not knowing whether the data is up to date isn't good either.

There are a number of algorithms available that use adapted versions of these methods. The most popular are IRs. In [15] a mobility aware dynamic database caching scheme for mobile computing is presented. Other algorithms using IRs are described in [7], [8] and [13]. Although they use adapted versions for use in mobile environments, they still have drawbacks. To compensate for the disconnections, the server stores the IRs until the client is connected again after which he sends the IRs. This requires even more administration at the server. Furthermore, when a mechanic wants to review some details of a customer, it is very inconvenient if that information is "not available". If, for example, only a

fax number of that customer has changed, the address should still be available to the mechanic. If the data item of this customer is marked "invalid", it won't be accessible at all. In this case, it is better to use the old data and when there is a change, send the new data item to the clients in stead of only a report that the old version is not valid anymore.

2.1.3 Replication

Replication is a method that distributes (parts of) a database (or other data storage) to other clients. Replication is usually done at table level, although models exist that use object-based replication. Clients can subscribe to a table and when there are changes in a table, they are sent to the client. This does require a constant connection to the server. When the connection is lost, the whole contents of the replicated database needs to be checked to ensure consistency. It is obvious this isn't an option for Connect-It.

The Bengal Database Replication System [11] is a database system designed for mobile environments. It was designed for traveling people who need to exchange their work with each other. There is no central server and updates are spread among the clients. It does allow different topologies, like a ring, tree or star topology. The latter is effectively a client-server architecture. However, the Bengal Database Replication System is an addition to existing databases and only works on a limited set of data storage mechanisms.

Roam [17], also a replication system for a mobile environment, uses the same principles and the updates are distributed between the clients. Although it is not database-specific, it only supports client-client communication without the addition of different topologies that mimic a client-server architecture.

The advantage of a client-client architecture is the lack of a central server. When a client can't reach one client, but can still contact another one, the updates can still take place. In the case of Connect-It: if a client can't reach the central server, it probably can't reach any other client because of the lack of network coverage.

For Connect-It, it is not preferable to replicate every table in the database or only a selection of tables as this could be more data than a single PDA could store. Some filter mechanism is required that distributes only a subset of the data in one or more tables and all changes done on that subset. Existing replication mechanisms don't support this.

2.1.4 Consistency techniques in a replication system

In a replication system, there are two methods of ensuring consistency between all clients [11]: Conservative (or Pessimistic) replication and Optimistic replication. Conservative replication uses locking or restricting to limit the operations on the data. This can have a tremendous negative effect on the other processes if a lock or restriction is not released. It also requires that all clients receive the lock before a certain operation can take place [12].

Optimistic replication allows all changes to be made locally and during an update consistency is checked and conflicts are resolved. The lack of client to client communication is now a big advantage in the client-server architecture [16]: all updates will pass through the server and all consistency checks can be done at the server. CVS [1] and Subversion [5], both document version control systems, work in a similar way. In general: the data has a record (timestamp) when it was last modified. When a client sends an update to the server and both timestamps match, then no conflict exists and the update can continue. If the timestamps do not match, there is a conflict that needs to be resolved.

2.1.5 Shared Data Spaces

In a shared data space system, the data is distributed over several clients, while no single client needs to contain all available data. An example is GSpace [18]. To the rest of the application, GSpace is a single data storage, although it is distributed over several clients. It is obvious this isn't what Connect-It needs. However, in [19] a mechanism is presented to dynamically change the data distribution policies within the GSpace system. The mechanism can be adapted to work with Connect-It in stead of with GSpace, to allow Connect-It to adapt to a changing environment.

2.2 Challenges

Before the architecture is analyzed in more detail (section 2.3), this section will look into the challenges of designing a distributed system and how they apply to Connect-It. According to [10], there are 7 challenges to deal with:

1. Heterogeneity
2. Openness
3. Security
4. Scalability
5. Failure handling

6. Concurrency
7. Transparency

Each of these challenges will be addressed in the following subsections and their applicability in Connect-It is discussed.

2.2.1 Heterogeneity

Different components in a distributed system can run on different types of computer hardware, use different network protocols, have different operating systems, and even the representation of the data can be different. An integer data type can be signed or unsigned and 16, 32 or 64 bits long. When designing a distributed system, one has to keep these differences in mind. One way to do this is to explicitly define the interfaces between each component and not to make any assumptions as a developer. The use of middleware is a common solution in distributed systems [10] [20]. Middleware is a term that indicates a software layer that abstracts from the underlying hardware, network and operating system.

To some extent, Connect-It uses middleware. The Microsoft .NET Framework and the Microsoft .NET Compact Framework abstract the developer from the underlying hardware and network protocols. The .NET Framework even abstracts from the details of the operating system, but one can assume it will be a Microsoft operating system in the case of Connect-It. In theory, other operating systems could be supported, but no full .NET Framework implementation exists for any non-Microsoft operating system. The Data Access Objects can also be seen as middleware, they are an abstraction of the data storage.

What Connect-It doesn't have, is middleware that abstracts the application logic from the distribution protocols. The application logic directly connects to the server, in stead of using a separate software layer for this task. If another protocol needs to be implemented in the future, it will be a very difficult and complex task to do if it is scattered across the entire application. One of the goals of this thesis is to design this separate layer (see section 1.4).

2.2.2 Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways [10]. An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services [20]. The syntax and semantics are often called the interface of a service.

When looking at Connect-It, several interfaces can be seen:

- The web service interface to the server.
- The DAO interface to the data storage.
- The .NET interface to different hardware and network systems.

However, some limitations can also be identified:

- There is no interface to the data distribution algorithms: changes in the web service result in changes needed in the PDA application logic.
- Changes in database design result in changes in the DAO interface.

The result is a system that is partially open: new application logic can be created that use the services of the lower software layers. However, changes in the lower software layers have a negative impact, as the upper layers also need to be adjusted. In an ideal environment, an interface is independent of specific implementation details and should remain the same. According to the problem statement in section 1.4, this is also a design goal.

2.2.3 Security

There are three security measurements possible [10] in distributed systems:

- Confidentiality (protection against disclosure to unauthorized individuals)
- Integrity (protection against alteration or corruption)
- Availability (protection against interference with the means to access the resources)

Confidentiality and integrity are very important within Connect-It. Since the shared data consists of orders and other financial data, it is important that only authorized individuals can access and change the data. At application level this is done by requiring a valid user name and password to log in to the application. During communication with the server, Secure Socket Layer (SSL) encryption is used to ensure safe data transport. Furthermore, each PDA is given a unique ID and activity using that ID is logged at the server. Requests that use an unknown ID are blocked at the server to prevent unauthorized access.

The availability aspect is tricky. Some preventive measures to for example a Denial Of Service (DOS) attack can be taken, but it requires actions from the network administrator instead of the application itself. The good news is that Connect-It is designed to work in an environment where availability isn't guaranteed, so this aspect is of a less concern to the design and no further effort is necessary to address this aspect.

To summarize, the security in Connect-It is already sufficient, no additional steps need to be taken here as long as the security doesn't degrade.

2.2.4 Scalability

A system can be scalable in three areas [20]:

- Size
- Geography
- Administration

Size

Scalable in size means more users or more resources. If a client-server architecture is being used, more clients usually means more work for the server, while client-client architectures often scale better in that respect. However, when confidential data is used, one secure server is usually preferred over many clients that also act as servers [20]. If needed, the data storage used at the server can be distributed across several other servers, but for the outside world, it looks like one server. When looking at an average working day for an average customer of Connect-It, there are 240 changes in 8 hours, or $\frac{1}{120}$ changes per second. Reading data from the data storage can be done in parallel, while writing requires a lock. This means that for those $\frac{1}{120}$ changes per second, a lock is needed. The time it takes to write 1 change to the data storage is τ, where τ is much lower than 1 (in seconds). This results in $\frac{1}{120}\tau$ being the non-parallelizable part in Amdahl's law about the speedup of parallel computing. The resulting speedup for using N servers (ignoring overhead) is:

$$S(N) = \frac{1}{\frac{1}{120}\tau + \frac{(1-\frac{1}{120}\tau)}{N}} \tag{2.1}$$

With a theoretical maximum (N is infinite) of $\frac{120}{\tau}$, which is a very good speedup.

Geography

Distributed systems designed for a local area network often don't work when used in a wide area network like the internet [20], as the performance of the internet is usually much less than the performance of a local network. Connect-It is already designed for use with network types that have a low performance. It evens keeps functioning when a network connection isn't available for hours, so it can be said that it is scalable enough with respect to the geography.

Administration

The third scalability aspect is administration. Does a distributed system keep functioning when it is used at a larger scale and expands to include different enterprises? Connect-It already deals with this aspect by allowing the administrators to define multiple administrations within Connect-It, each having their own set of users and data. As each administration is independent of other administrations, each administration could have its own separate server and data storage, achieving linear speedup: $S(N) = N$. So Connect-It is also very scalable with respect to Administration.

To summarize, Connect-It is a very scalable system that can be used both in a small and a large scale environment, using whatever network connections are available. As long as the architecture doesn't degrade the scalability, no additional steps need to be taken here as long as the scalability doesn't degrade.

2.2.5 Failure handling

Sometimes computer systems fail. Whether it is a software error or a hardware error that caused the failure, the failure needs to be handled in a correct way. In a distributed system, the failure of one component should not result in failures of other components. In Connect-It, the number of devices can also change over time (see section 1.4): the number of employees change, or PDAs are simply turned off or run out of battery. In that case, the loss of data should of course be minimized. Important data shouldn't be stored in memory alone, but also on a disk or other persistent storage device.

Currently, the failure of one client won't result in any other failures in the system. When the server is down, the client will keep functioning just as if there was no network connection available. However, the changes a client made locally and are waiting to be sent to the server only exist in memory. If a local failure occurs, these changes will never be sent to the server (omission failure). The new software architecture should ensure that changes in the data that are waiting to be transferred to the server will still exists after a local failure occurs.

2.2.6 Concurrency

A server provides resources that can be shared by several clients, so there is a possibility that several clients will attempt to access a shared resource at the same time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent [10].

At the moment, not much can be said about consistency within Connect-It.

When the server receives data from a client, it stores that data without checking for any conflicts. As the communication with the server is initiated at many different places in the application logic of the client, there is no easy way to address this problem. However, when using a new software layer for the distribution of the data, the problem becomes much less complex.

Conflicts can be detected during a *Commit* by comparing the timestamp of each data item received from the client with the timestamp of the corresponding data item at the server (see section 1.3). If a conflict occurs, it needs to be resolved before the *Commit* can successfully complete. If it can't be resolved, a rollback mechanism is required as *Commit* should guarantee that all changes are applied in one atomic operation.

Because a client might be disconnected from the server for an undetermined amount of time, the best consistency model that describes Connect-It is Eventual Consistency: when no updates occur for a long period of time, eventually all updates will propagate through the system and all the replicas will be consistent.

The new architecture should provide means to detect and resolve conflicts to guarantee consistency within Connect-It. As most conflicts will probably occur in order data or other financially sensitive data, ViaData prefers human interaction. This is also easy to implement, alert a user (whether it is at the client side, the server side or at the back office) that there is a conflict and provide an option to select either one version or a merged version. The application logic to send messages to a certain user or a certain user group (for example the administrator group) already exists. It is only a small addition to extend this existing system to include messages about conflicts.

2.2.7 Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components [10].

Connect-It consists of many different components that work together to give a user the illusion that it is one system it is interacting with. Furthermore, the use of the Data Access Object design pattern allows the application developers to access the data without worrying about the specific details that play a role in storing or retrieving data. The DAO layer hides those aspects and can be seen as a single component.

The distribution of the data is still not transparent to the application developers. One of the goals (see section 1.4) is to design a new software layer that hides all the details that play a role in the distribution of the data.

One might ask if 100% transparency is a good thing, as there are several examples where hiding all details is not a good idea [20]. For example: in the case of Connect-It, when a PDA can't contact the server for several days, that information is relevant to the user. Furthermore, it is currently possible for a user to manually trigger the synchronization process for a client. A user might want to do that if he is at the office and has his PDA in a cradle. In most cases, a high degree of transparency is preferable, but hiding everything is often not the best solution.

2.2.8 Summary

In table 2.1 an overview is given of the different challenges and how they apply to Connect-It. For each challenge there are two possible scores:

- When the challenge has been dealt with in an adequate manner, the score is a +
- When the challenge hasn't been dealt with or improvement is necessary, the score is a -

Challenge	Score	Improvement
Heterogeneity	-	Use middleware to abstract the distribution of the data.
Openness	-	Use interfaces that are independent of implementation changes.
Security	+	No improvement necessary.
Scalability	+	No improvement necessary.
Failure handling	-	Persistently store the changes of a client.
Concurrency	-	Add conflict detection.
Transparency	-	Make the distribution of the data transparent to the application logic.

Table 2.1: Challenges within Connect-It

As can be seen, Connect-It can and should be improved for 5 of the 7 challenges and should not degrade for the 2 challenges that already have been dealt with in an adequate manner.

2.3 Architectural analysis

Using the 4+1 views of Kruchten [14] I will describe the current architecture. First, the logical view is shown in section 2.3.1. Section 2.3.2 describes the process view, followed by the development view in section 2.3.3. After that, in section 2.3.4, the physical view of the architecture is shown. The fifth view deals with several use cases in section 2.3.5.

Note that there was no complete or extended architectural documentation available, I performed this analysis by reverse engineering the system as it is.

2.3.1 Logical view

The logical view shows the object model of the design. First the class diagram will be discussed, after that the interaction between the different classes is shown in sequence diagrams.

Class diagram

The class diagram in figure 2.1 only goes into detail about the part of the application that is relevant for the distribution of the data. The following list shows all classes and their purposes.

- **PdaApplication**
  This package represents the rest of the PDA application. It uses the ConnectionManager, the SyncManager and the PDAWebService to communicate with the server.
- **DTO**
  DataTransferObject. This abstract class is the base for all data classes, such as Order and Customer. Each record in the database is represented by a DTO object.
- **Order**
  This class represents order data, it inherits from DTO.
- **Customer**
  This class represents customer data, it inherits from DTO.
- **...**
  Represents other classes that inherit from DTO, just like Order and Customer.
- **ConnectionManager**
  This class can open a new connection to the server.

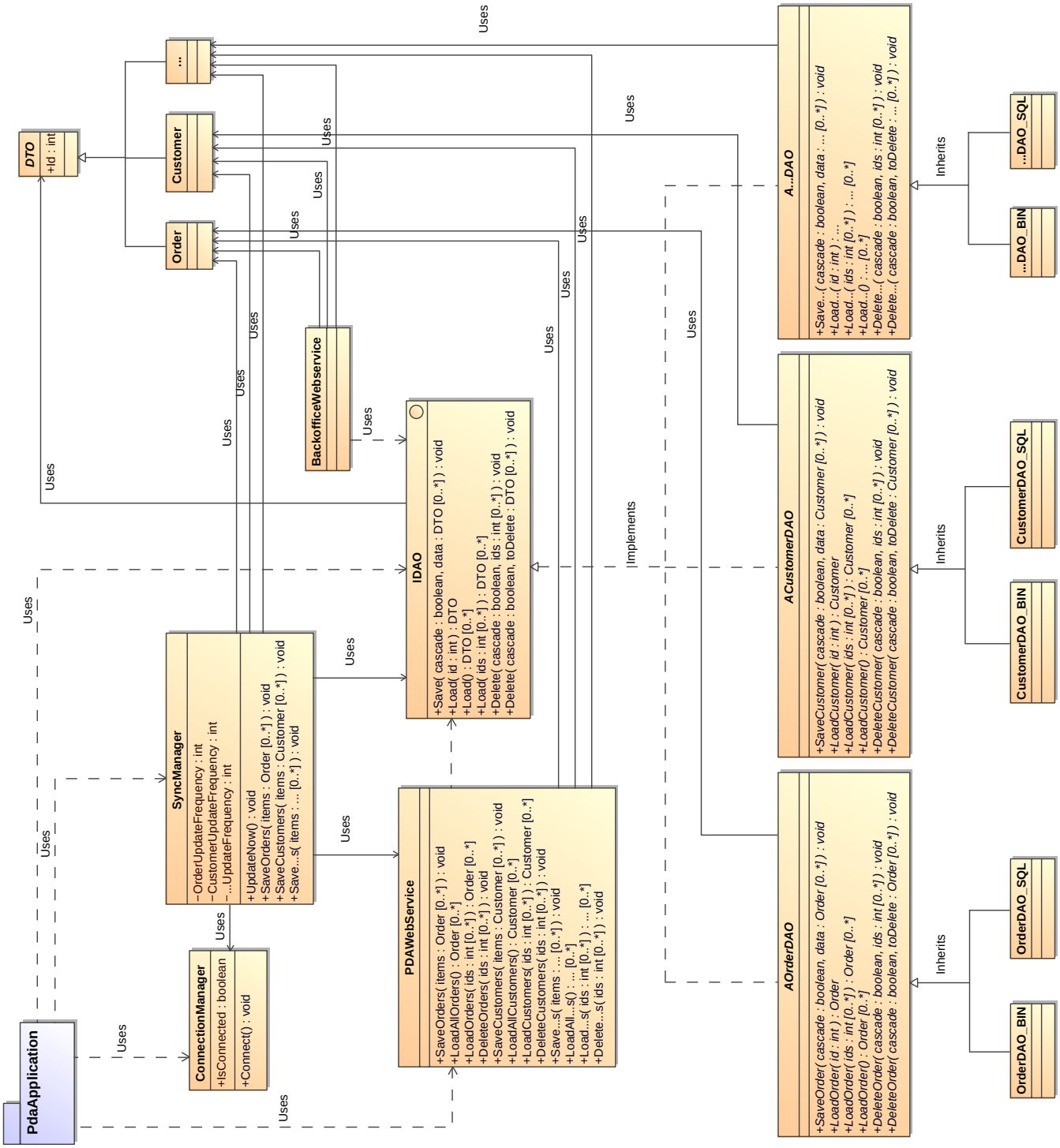

Figure 2.1: Class diagram (analysis)

- **SyncManager**
  This class can periodically download all data for a single type (Order, Customer, ...). It will delete all data of that type that is currently stored in the data storage.

- **PDAWebService**
  This is the PDA webservice running at the server, all PDAs will use this webservice to communicate with the server. It has specific load, save and delete methods for each data type (Order, Customer, ...). It uses the DAO interface to store and retrieve data from the data storage.

- **BackofficeWebservice**
  This is the backoffice webservice running at the server, all backoffice clients will use this webservice to communicate with the server. It has many different methods compared to the PDA webservice which are not relevant for PDA clients.

- **IDAO**
  This is the DAO (Data Access Object) interface. It provides general load, save and delete functions to the data storage used. Note that all methods use the abstract DTO type, it depends on the implementation of a specific DAO what the return values are.

- **AOrderDAO**
  This abstract class represents a DAO for accessing order data. It offers type safe variants of the methods supplied by the DAO interface.

- **ACustomerDAO**
  This abstract class represents a DAO for accessing customer data. It offers type safe variants of the methods supplied by the DAO interface.

- **A...DAO**
  Represents other abstract DAO classes, just like AOrderDAO and ACustomerDAO.

- **OrderDAO_BIN**
  This is the DAO for accessing order data using binary files (in a custom designed format).

- **CustomerDAO_BIN**
  This is the DAO for accessing customer data using binary files (in a custom designed format).

- **...DAO_BIN**
  Represents other DAOs using binary files (in a custom designed format).

- **OrderDAO_SQL**
  This is the DAO for accessing order data using a SQL database.
- **CustomerDAO_SQL**
  This is the DAO for accessing customer data using a SQL database.
- **...DAO_SQL**
  Represents other DAOs using a SQL database.

When looking at the class diagram, three aspects can be observed that need to change:

1. Transparency. The PdaApplication uses two ways to communicate with the server, either by using the PDAWebService directly and via the SyncManager. There is no guideline or documentation when each method is used. It would be better if the connection with the server was transparent to the PdaApplication, so it will be clear when and how the connections are used.
2. Openness. There are a lot of dependencies to the derived DTO classes like Order and Customer, even when it is not relevant what type of data is being transferred. Now, most classes have multiple methods for each derived DTO class. Using generic programming techniques, these classes would only have a dependency to the abstract DTO class and in stead of writing each method separately for each derived class, only one method will suffice. When a new data type is introduced, there is no change necessary in the distribution classes.
3. Openness. Every DAO class has a specific implementation for each derived DTO class. Again, using generic programming techniques, one class per DAO type (Binary text files, SQL database, ...) would suffice. When a new data type is introduced, no new DAO implementations have to be made.

Sequence diagrams

The sequence diagrams show the interaction between the different classes. Figure 2.2 shows a thread of the PDA application that interacts with the underlying DAO and how data items are sent to the Buffer. The Buffer runs a second thread that interacts with the server, this can be seen in figure 2.3. Finally, figure 2.4 shows the sequence of updating the local data storage.

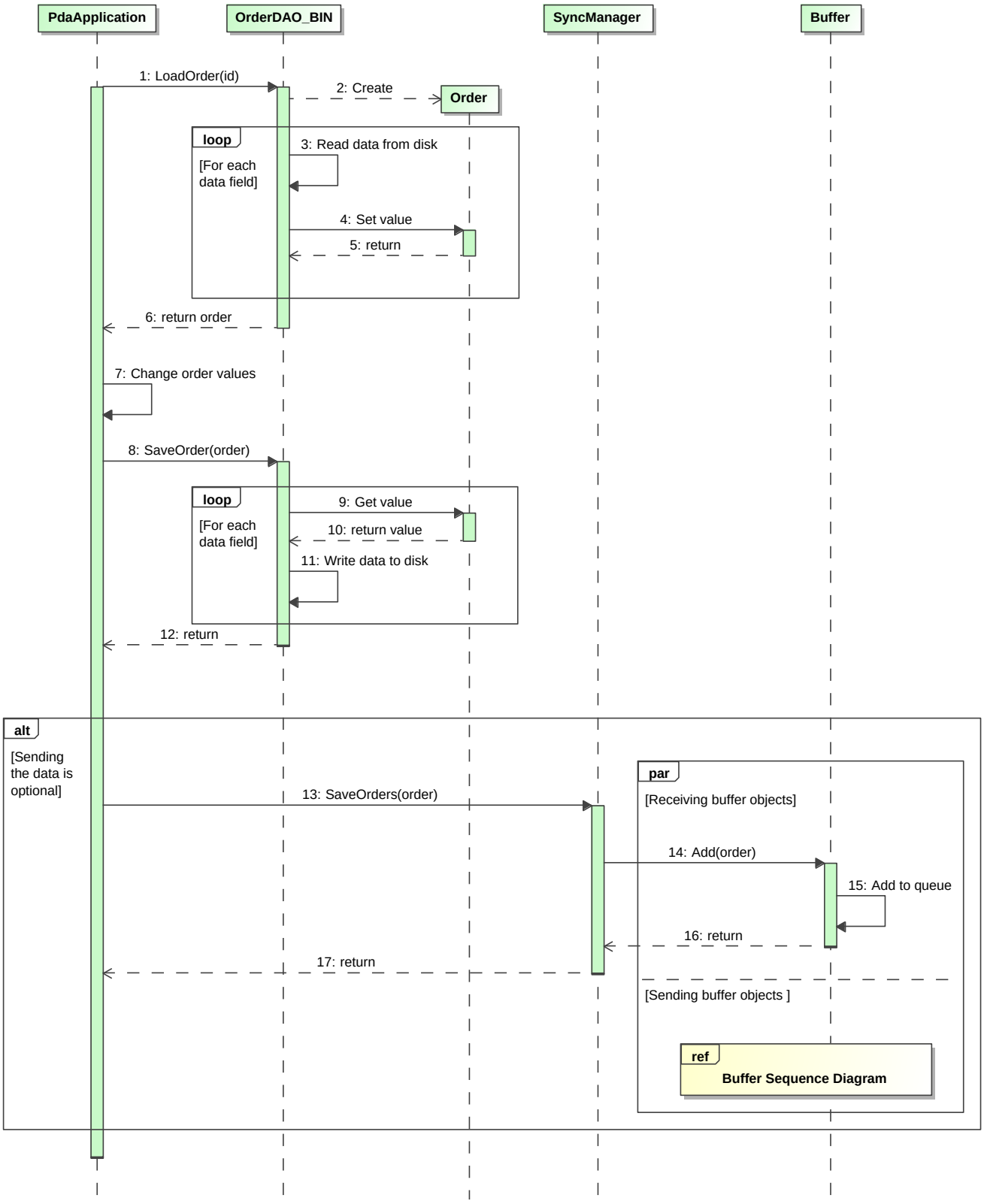

Figure 2.2: Application sequence diagram (analysis)

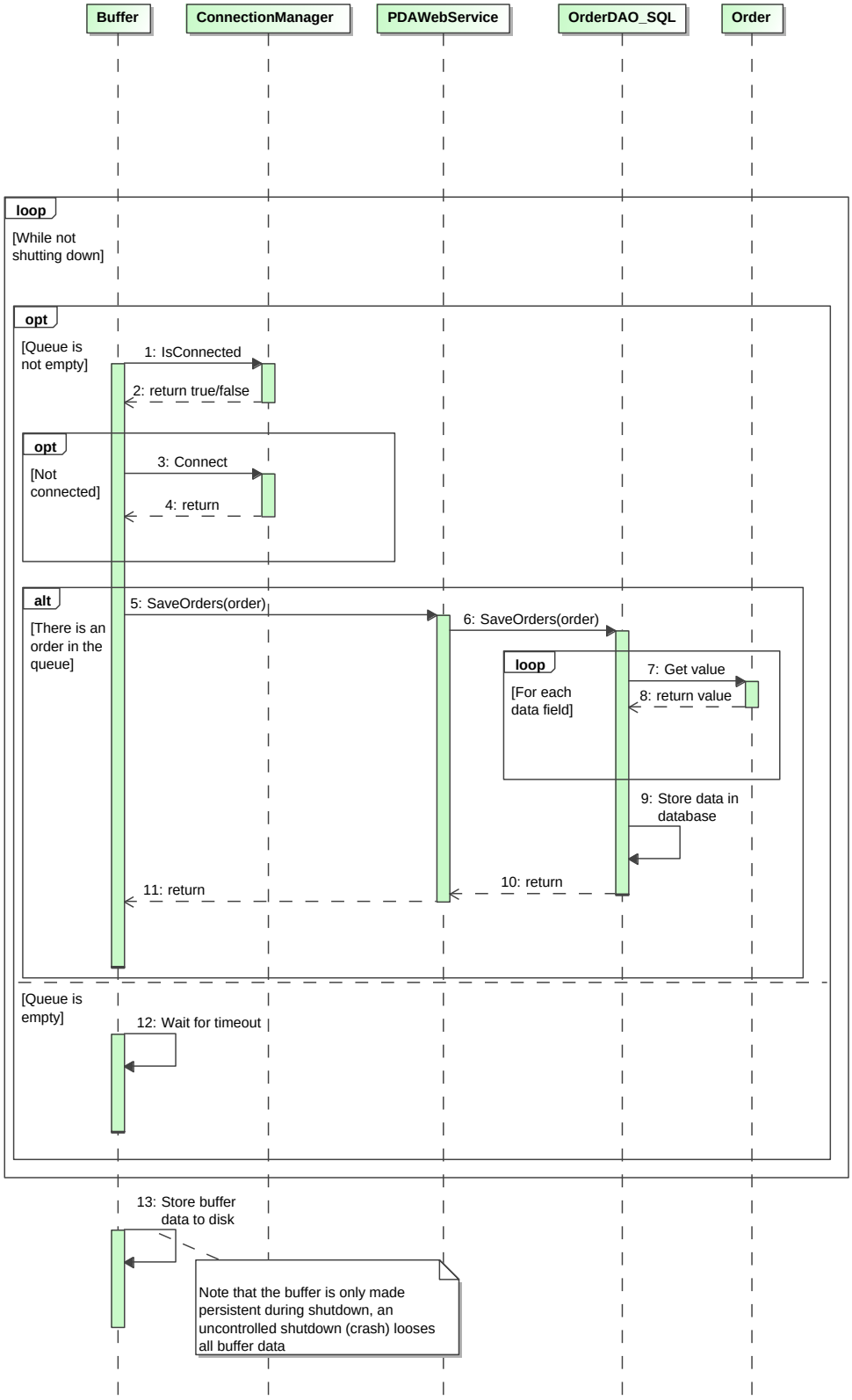

Figure 2.3: Buffer sequence diagram (analysis)

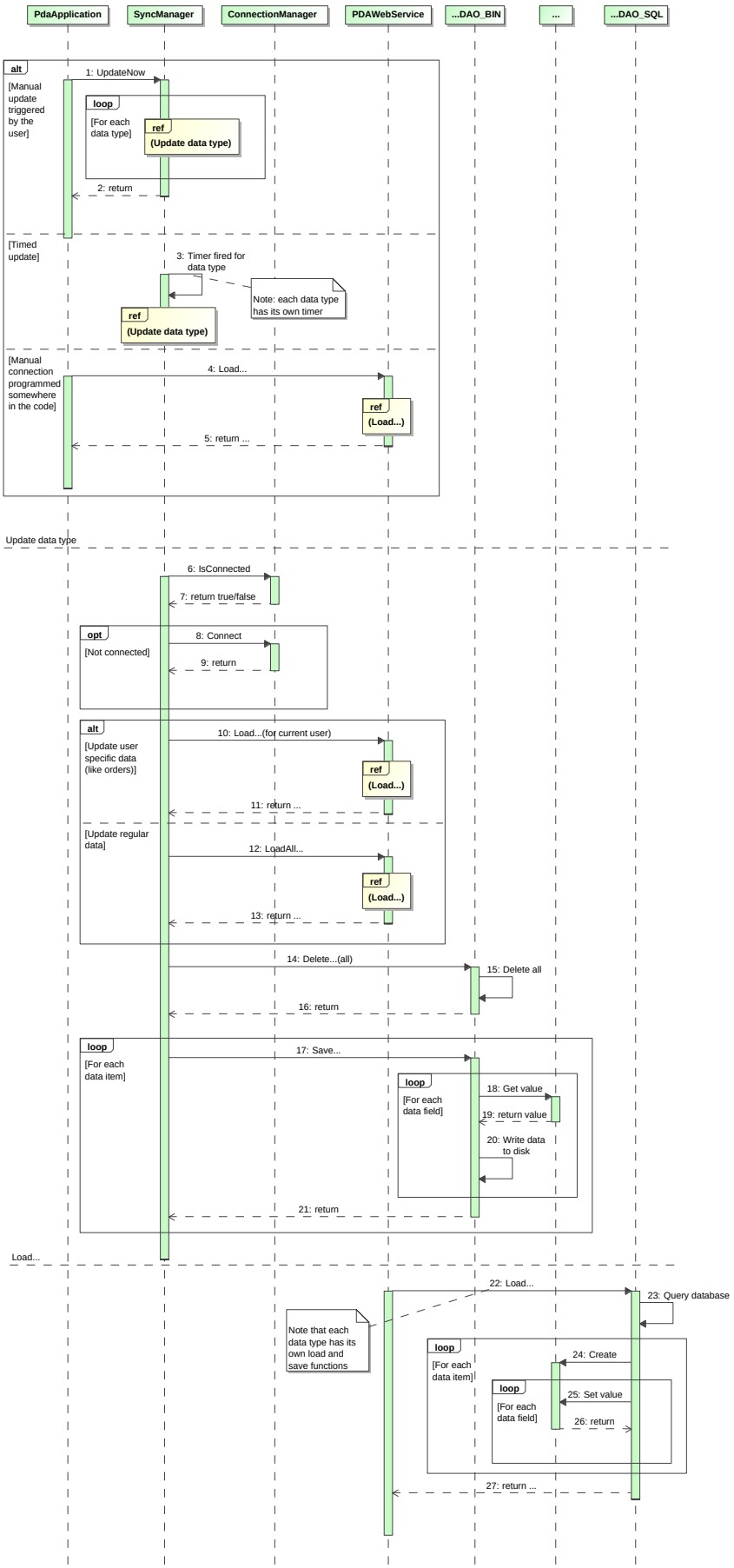

Figure 2.4: Update sequence diagram (analysis)

Each diagram shows aspects that can (or should) be improved:

- **Application sequence**
  The first half of the diagram in figure 2.2 shows the DAO pattern, how data is retrieved and stored again. The second half is optional and this is a big flaw: every change in the data should be transferred to the server (see section 1.3). In most cases the application logic will do this, but sometimes this is forgotten. If the connection with the server was transparent and transfers were done automatically, this problem would be solved.

- **Buffer sequence**
  The buffer component (when used) receives DTO objects on a separate thread (see the application sequence). Another thread is looping and when there is an item that needs to be transfered, it tries to connect to the server. If the connection fails during the SaveOrders call, the call returns and the objects are considered to be transferred successfully, there is no error detection. Furthermore, all buffered data exists in memory alone: only during a controlled shutdown is the data persistently stored to disk. These are two flaws that need to be resolved.

- **Update sequence**
  The biggest performance bottleneck in the update sequence is that all DTO objects are transferred every time a $Refresh$ takes place. The advantage is that the local data storage really is up to date, but that can be achieved in many other ways. The model in section 1.3 gives a solution: apply all local changes to a buffer and don't let the client change anything in its local storage. When updating, only transfer the changes in the data. This can be compared with the IMS messages from section 2.1.2. The key is to design a efficient method of querying for changes in the whole local data set.

2.3.2 Process view

The process view shows the different processes and threads mapped to hardware components. In this case, the PDA consists of one process: *Connect-It SE.exe*. The server runs the webserver that executes the *pda.asmx* webservice. Figure 2.5 illustrates this. Note that the backoffice webservice is not shown here, as the focus of this thesis is the part of the architecture that deals with the PDA clients. What activities take place in each thread can be seen in the sequence diagrams from section 2.3.1.

Looking at the different threads, a problem can be identified. Some PdaApplication threads directly connect to the server to send and receive data. This has the effect that when a thread at the server blocks (for example, because access to the server data storage is synchronized), the PdaApplication thread will

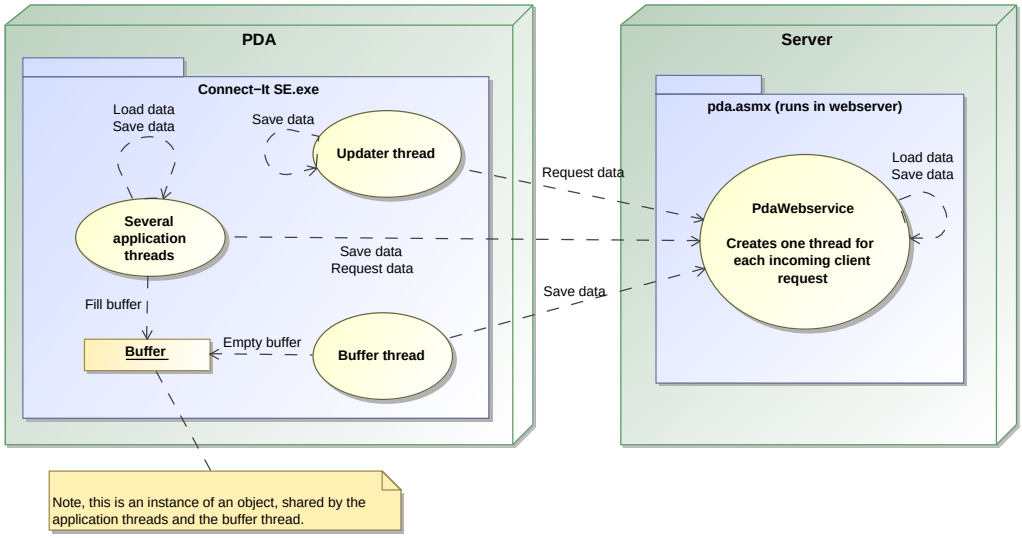

Figure 2.5: Processes and threads (analysis)

block. This could reduce responsiveness at user level. When all server interaction would be done using separate threads, this problem would be solved.

2.3.3 Development view

The development view shows a decomposition of the system in several software components and their relation. Figure 2.6 shows this decomposition for Connect-It in three layers: an application specific layer on top, a middleware layer and a general components layer on the bottom. The lines show the dependencies of the different software components. Note that for each component the classes it contains is shown, see the logical view in section 2.3.1 for more information on each class. Below, the functionality of each component is explained.

- **Connect-It SE.exe**
  The main executable of the PDA, it initializes the program and shows the logon screen as the application starting point for a user. Depends on:
  - Globals.dll
  - PDAGlobals.dll
  - DataObjects.dll
  - Managers.dll
  - FlowManager.dll
  - Schermen.dll
  - pda.asmx

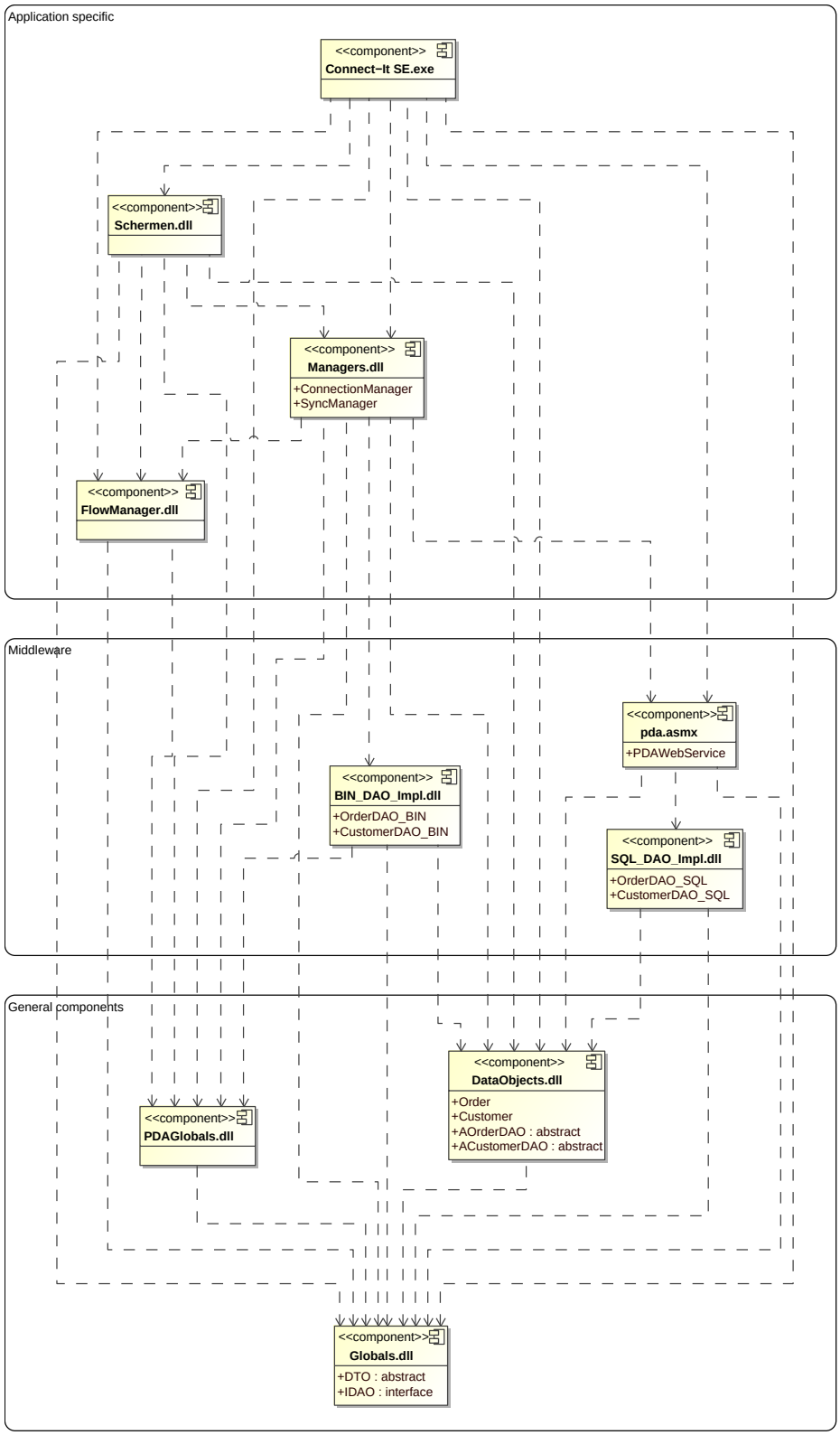

Figure 2.6: Component dependencies (analysis)

- **Schermen.dll**
  All user interface screens are located in this component. At startup, all screens are registered at the FlowManager. When a screen is closed, the FlowManager determines what screen is shown next, depending on the return value of the closing screen. This enabled different actions based on the condition when the screen is closed (i.e. what button was pressed). This component is the largest component, as it holds most of the application logic. Depends on:
  - Globals.dll
  - PDAGlobals.dll
  - DataObjects.dll
  - Managers.dll
  - FlowManager.dll
- **Managers.dll**
  This components holds functionality that doesn't belong to a single screen. It also keeps track of data between several screens, for example the current user that is logged in and the current order that the user is working on. One of the managers in this component serves as access point for the application logic in the screens to connect to the server. Depends on:
  - Globals.dll
  - PDAGlobals.dll
  - DataObjects.dll
  - FlowManager.dll
  - pda.asmx
  - Any DAO layer (implementation uses BIN_DAO_Impl.dll)
- **FlowManager.dll**
  This components determines the flow of the application: what screen is shown next. A script file is read that enables the application to show several screens in a different order. This enables the software to be customized to the working methods preferred by a customer. Depends on:
  - Globals.dll
  - PDAGlobals.dll
- **pda.asmx**
  This is the web service definition, it receives calls from the clients and can access the server database. Depends on:

  - Globals.dll
  - DataObjects.dll
  - Any DAO layer (implementation uses SQL_DAO_Impl.dll)

- **BIN_DAO_Impl.dll**
  This component implements the data layer for binary text files using the Data Access Object design pattern [2]. Depends on:
  - Globals.dll
  - PDAGlobals.dll
  - DataObjects.dll

- **SQL_DAO_Impl.dll**
  This component implements the data layer for a SQL database using the Data Access Object design pattern [2]. Depends on:
  - Globals.dll
  - DataObjects.dll

- **DataObjects.dll**
  This component holds all data objects like "Customer" and "Order". Note that this component is application specific, even though it is located in the general components layer. Depends on:
  - Globals.dll

- **PDAGlobals.dll**
  This component holds global information that is relevant for the PDA application, for example the ip-address of the server. Depends on:
  - Globals.dll

- **Globals.dll**
  This component holds global information that is relevant for both the PDA application and the server application, for example the abstract type "DTO" (see section 2.3.1). Depends on nothing.

Note that these components are spread over the clients and the server. See section 2.3.4 for more details on the deployment.

Two observations can be made:

1. There is no middleware that abstracts the distribution of the data, the application logic directly connects to the pda webservice *pda.asmx*. If there is a change in the web service or a whole new connection type is used, the application specific components need to be changed as well in several locations. Better would be to use an interface that is implementation and connection type independent.

2. The second observation is that the application specific component *DataObjects.dll* is located in the general components layer. Note that I placed it there based on the dependencies in the middleware layer. One could argue that it could be located in the middleware layer as well. However, it is still application specific and should therefore be located in the application specific layer. The middleware layer should have no dependencies whatsoever to this *DataObjects.dll* component. The advantage will be that changes in what data is stored and transferred (for example, adding a new data type) does not result in any change necessary in the middleware layer.

2.3.4 Physical view

The physical view describes the different hardware components and what software they deploy. Figure 2.7 shows the server and a PDA and how the components from section 2.3.3 are deployed on them. Note that Globals.dll and DataObjects.dll are deployed by both the server and the PDAs.

From this point of view, the DataObjects.dll should move to the Application Layer. Furthermore, separate components for distributing the data should be added to the Middleware Layer.

2.3.5 Scenarios

The use case diagram (figure 2.8) shows the mechanics as actors and the backoffice as an actor. Basically, from a user point of view, all actors want to be able to view and make changes to the data. It is the task of the system to make sure that all relevant data gets transferred between the server and the clients. Currently, there is no conflict detection.

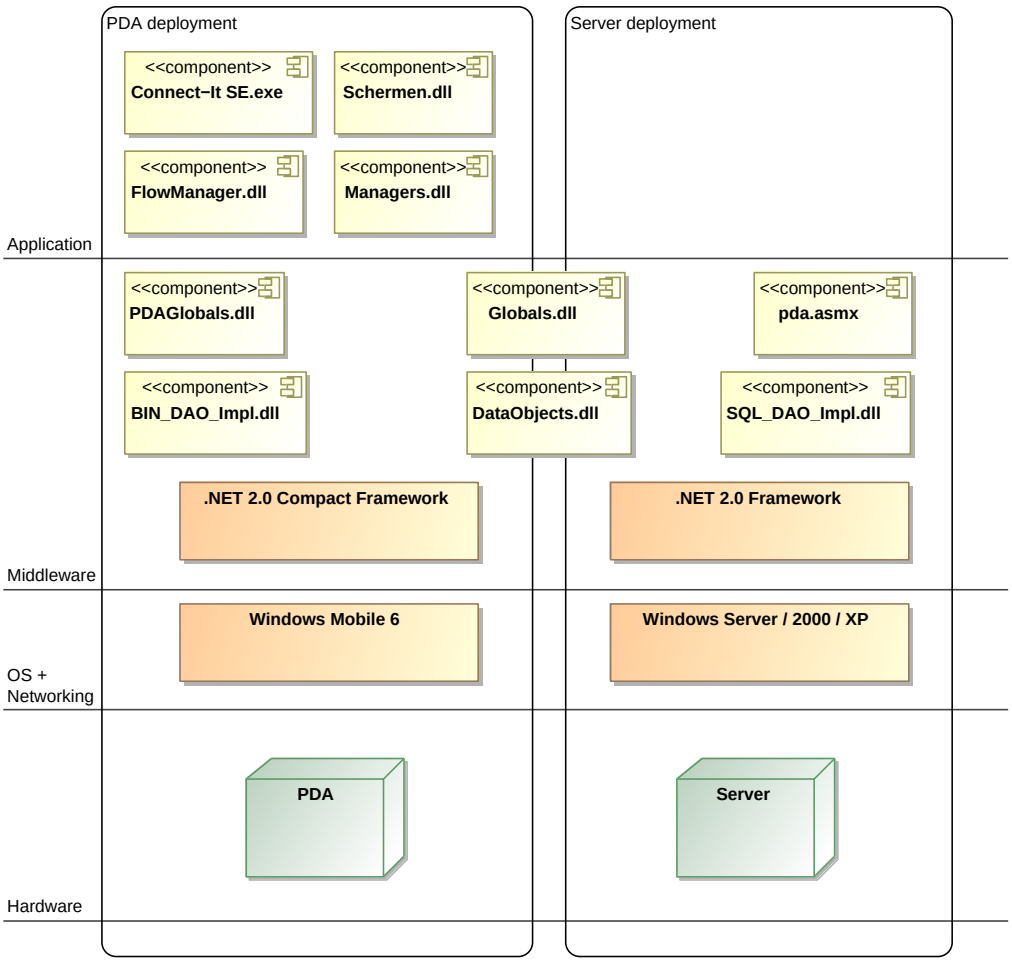

Figure 2.7: Deployment (analysis)

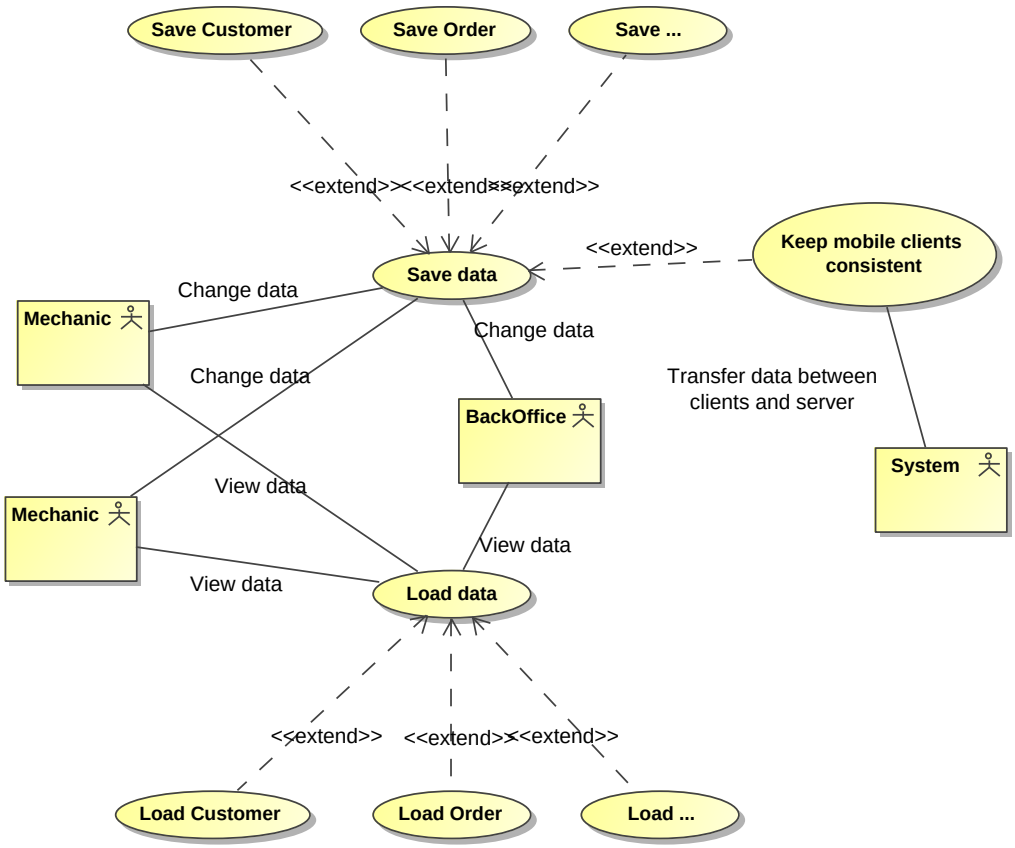

Figure 2.8: Use case diagram (analysis)

2.4 Cost functions

The use of cost functions allows us to quantify certain aspects, such as communication, processing time and consistency. With the costs quantified it is possible to compare different distribution protocols. The protocol with the lowest cost would be the best protocol. When only one protocol will be implemented, the optimal protocol given the requirements can be determined. When an adaptation mechanism is built, the parameters can be calculated while the program is running and with the use of the cost functions the optimal protocol given the requirements and given the current environment can be determined. If the current protocol isn't optimal, the system can switch to another protocol. The adaptation mechanism that is required for this behavior is described in [19].

In subsection 2.4.1 the parameters that play a role in the cost functions are discussed. The subsections 2.4.2, 2.4.3, 2.4.4 respectively introduce the cost function for communication, client processing time and consistency. In subsection 2.4.5 a trade off can be seen between the different parameters and finally in subsection 2.4.6 the three cost functions are combined.

2.4.1 Parameters

There are several parameters that influence the quality (cost) of a protocol. These parameters can be split into two groups: environmental parameters and protocol parameters.

Environmental parameters

Environmental parameters are parameters that are not affected by the protocol choice. For example: how often does the data change and what is the average size of a change that needs to be transferred?

The changes in the data will probably not occur in a uniform distribution during the day. When an employee changes one thing, it is likely that some other changes will follow very soon after the first change. This behavior of bursts of changes can be modeled with a compound Poisson distribution. To do this, two assumptions need to be made:

1. The amount of changes in each burst is not of any influence on the number of bursts that take place.

2. The expected number of changes in a burst is the same for every burst.

These assumptions can be justified by looking at how the changes are entered into the system. In the case of Connect-It, this can be done manually or by

using a connection to an ERP system. In both cases, these changes can include a new order, a change in an existing order, changed customer data, a change in the inventory of a mechanic, etc. These are all bursts (events) coming from the environment, so the specific amount of changes for the system in one of these bursts is not of any influence to the number of bursts (events) that take place, justifying the first assumption. Most bursts include a very limited number of changes and the expected number of changes in each burst is the same, justifying the second assumption.

To summarize: a burst is an event that generates a number of changes in the data. The number of bursts that will happen during a certain interval Δt is $X_{\Delta t}$ and has a Poisson distribution with an expected value $\lambda_{\Delta t}$:

$$P(X_{\Delta t} = k) = \frac{e^{-(\lambda_{\Delta t})}(\lambda_{\Delta t})^k}{k!} \tag{2.2}$$

The expected value scales linearly with Δt: there will probably be twice as many changes if the interval is twice as long. The number of unique changes within burst x is a random value Y_x. The sum of all changes within interval Δt, say $N_{\Delta t}$, has a compound Poisson distribution:

$$N_{\Delta t} = \sum_{x=1}^{X_{\Delta t}} Y_x \tag{2.3}$$

Changes in different bursts don't have to be unique, multiple changes can occur in one data item. For example, if a new order is submitted and there was a typing error, correcting the error is a second change on the same data item (in a new burst). If both changes occur within interval Δt, $N_{\Delta t}$ will contain more changes than there are data items that need to be distributed. However, this fact does not change the theoretical worst case scenario where all changes are unique. In most cases, most changes in interval Δt will be unique, so the worst case scenario is the best scenario to work with.

According to the compound Poisson distribution, the expected value of $N_{\Delta t}$ is:

$$E(N_{\Delta t}) = E(X_{\Delta t})E(Y) = \lambda_{\Delta t}E(Y) = \gamma \times \Delta t \tag{2.4}$$

Here, γ is the interesting part: the expected number of changes per second. Note that the distribution of Y is not important, only its expected value, which is the expected number of changes in a burst as mentioned in the second assumption earlier in this section. The distribution of $X_{\Delta t}$ is a Poisson distribution, which is ideal for expressing the probability of a number of events in a fixed time interval if these events occur with a known average rate. The interval Δt is not necessarily the same as the interval between two successive checks, but can be linearly scaled

with respect to $E(N_{\Delta t})$. When the application is running and an adaptation mechanism is used, this mechanism can calculate the number of changes during a certain interval to get realistic estimates for the parameter γ.

Another environmental parameter is the average size of each data item (in bytes). In Connect-It, most data items are records in a database. As mentioned before, there is a chance that not all changes in one interval are changes on unique data items, but most changes will affect unique data items. The shorter the interval Δt, the likelier the changes will affect unique data items.

Protocol parameters

Protocol parameters are parameters that are affected by the protocol choice. This can be the frequency of checks for changes, the amount of communication overhead needed when checking for changes (in bytes) and the amount of communication needed to keep the data up to date (in bytes). The last one might be trivial: equal to the size of the changes, but not all protocols transfer only the changed items, so this is a protocol parameter and should not be ignored.

2.4.2 Communication

It is possible to define the communication needed per client (in bytes per second):

$$CF_{Comm}(F_c, C_o, S_d) = F_c \times C_o + \gamma \times S_d \tag{2.5}$$

Where F_c is the frequency of the checks, C_o is the communication overhead needed when checking for updates, S_d is the average size of the data that is being transferred for each change and γ is the expected number of changes per second (system wide).

An observation can be made from this formula: every change has to be transmitted to the clients. It will probably be the case that not all data items are needed by all clients, so the formula is based on the worst case scenario that all data items are needed by every client.

An average working day for an average customer who uses Connect-It will consist of around 240 changes per 8 hours. As seen in section 2.4.1, the average size of a single data item will be 100 bytes. The behavior of CF_{Comm} can plotted to study its characteristics using these averages. In all figures γ is set to $\frac{1}{120}$ changes per second, this equals 240 changes per 8 hours. Changing γ is the same as changing S_d, they are both a linear factor. In figure 2.9, S_d is set to 100 bytes. The result is a quadratic behavior between F_c and C_o. In figure 2.10 C_o is set to 150 bytes. Here it shows that S_d is only a linear factor. Figure

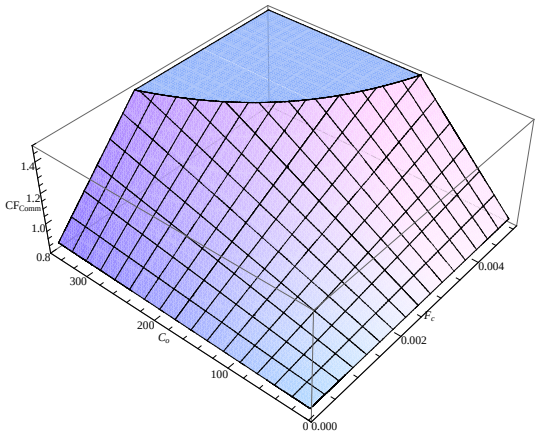

Figure 2.9: Communication cost using F_c and C_o. $\gamma = \frac{1}{120}$, $S_d = 100$.

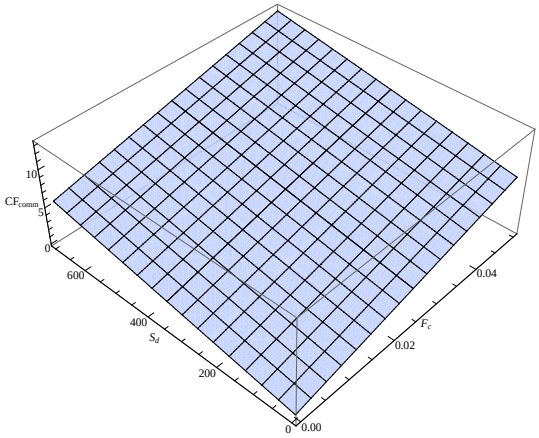

Figure 2.10: Communication cost using F_c and S_d. $\gamma = \frac{1}{120}$, $C_o = 150$.

2.11 show the relation between C_o and S_d with F_c set to $\frac{1}{300}$. Again, this is linear.

Another interesting aspect is not the frequency of the checks, but the time between two successive checks: $\frac{1}{F_c}$. Figure 2.12 is the result, where S_d is again set to 100 bytes. The result is an asymptotic curve for $\frac{1}{F_c}$. To be complete $\frac{1}{F_c}$ is plotted against S_d in figure 2.13, while keeping C_o at 150 bytes. The result is very different in top-down view. In figure 2.12 this is a straight line, the result from the quadratic relation between F_c and C_o, while in figure 2.13 this is another asymptotic curve, emphasizing the linear relation of S_d to F_c.

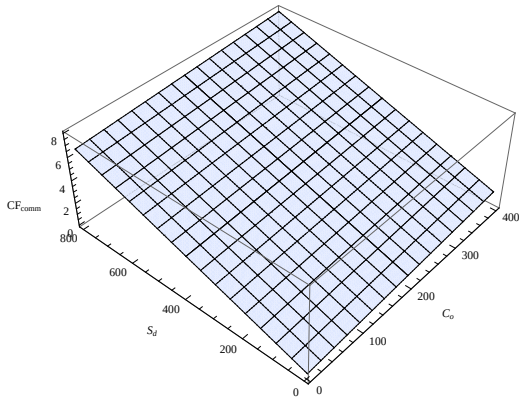

Figure 2.11: Communication cost using C_o and S_d. $\gamma = \frac{1}{120}$, $F_c = \frac{1}{300}$.

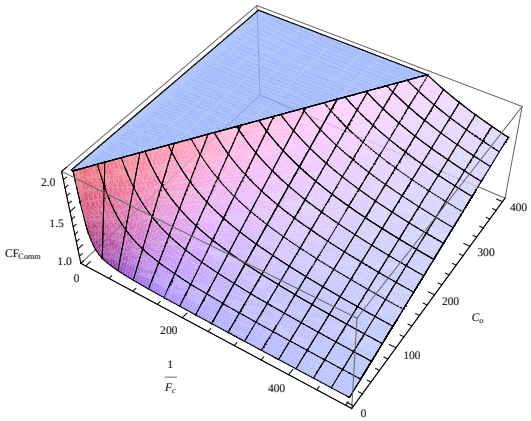

Figure 2.12: Communication cost using $\frac{1}{F_c}$ and C_o. $\gamma = \frac{1}{120}$, $S_d = 100$.

2.4.3 Client processing time

The total processing time at the client (in operations per second) can be defined as:

$$CF_{Proc}(S_d) = \gamma \times W(S_d) \tag{2.6}$$

Where S_d is the average size of the data that is being transferred for each change, γ is the expected number of changes per second and $W(x)$ is a function that determines the amount of processing a client needs to store (write) the amount of data x. Every time data is being transferred, it needs to be stored. W is monotonic, in other words: W has the property that if and only if $a \leq b$ then $W(a) \leq W(b)$.

Note that F_c is not relevant here. This is based on the worst case scenario that all changes are unique. In reality, the higher the time between two checks, the higher the chance that two changes are affecting the same data item, reducing

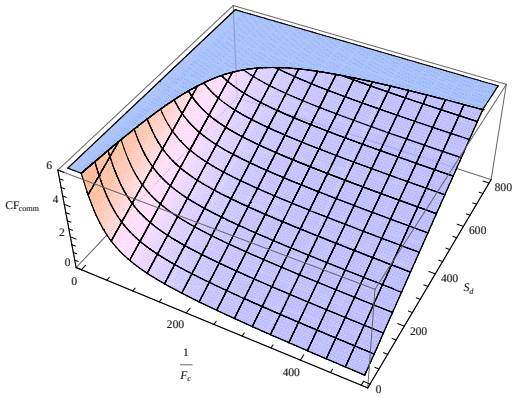

Figure 2.13: Communication cost using $\frac{1}{F_c}$ and S_d. $\gamma = \frac{1}{120}$, $C_o = 150$.

the processing cost. In my opinion it is best to work with the worst case scenario, especially for realistic values of F_c instead of theoretical values such as one check every year.

2.4.4 Consistency

Consistency cost can be represented by the percentage of the time that the client state is not consistent with the server state. According to [21] the inconsistency ratio of a system I^* is defined by the average inconsistency ratio of all possible traces Z that can happen in a system:

$$I^* = \sum P(Z) \times I(Z) \tag{2.7}$$

Where $P(Z)$ is the probability that trace Z occurs and $I(Z)$ is the inconsistency ratio of that trace and is defined by:

$$I(Z) = \frac{1}{D(Z)} \times \int_0^{D(Z)} ic(x_Z(\tau))d\tau \tag{2.8}$$

Where $D(Z)$ is the duration of trace Z, where $x_Z(\tau)$ defines a client state c_i at time τ given a trace Z and where $ic(c_i)$ defines whether state c_i is consistent:

$$ic(c_i) = \begin{cases} 0 & if\ \forall n \in dom(c_i) : c_i(n) = s(n) \\ 1 & if\ \exists n \in dom(c_i) : c_i(n) \neq s(n) \end{cases} \tag{2.9}$$

To approximate I^*, one can find a large enough $D(Z)$ and assume that the resulting $I(Z)$ is close enough to I^* [21].

To approximate $I(Z)$ where $D(Z) \to \infty$, one can look at the ratio between the expected number of changes per second γ and the frequency of the checks F_c, this can be used as a cost function for consistency:

$$CF_{Con}(F_c) = \frac{\gamma}{F_c + \gamma} \tag{2.10}$$

It is obvious that the higher the frequency of the checks is, the higher the consistency and the lower the consistency cost. This is shown in figure 2.14. Another interesting aspect is the time between two successive checks: $\frac{1}{F_c}$, shown in figure 2.15. Both figures have $\gamma = \frac{1}{120}$.

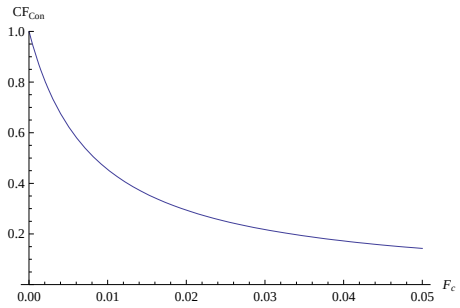

Figure 2.14: Consistency cost using F_c. $\gamma = \frac{1}{120}$.

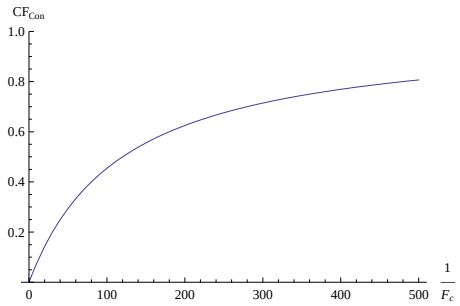

Figure 2.15: Consistency cost using $\frac{1}{F_c}$. $\gamma = \frac{1}{120}$.

2.4.5 Trade off

Looking at the cost functions, a trade off can be made: trade communication costs against consistency. Checking more often for changes (a higher F_c) results in a better consistency at the cost of more communication needed. It is also possible to use different frequencies on PDAs who use different subsets of the data items, based on the expected number of changes that will occur on that subset. However, this is not possible in the worst case scenario that all clients need all changes in the data.

2.4.6 Total cost

When all cost functions are known, a total cost function can also be defined:

$$\begin{aligned} CF_{Tot}(F_c, C_o, S_d) = \quad & w_1 \times CF_{Comm}(F_c, C_o, S_d) \\ + \quad & w_2 \times CF_{Proc}(S_d) \\ + \quad & w_3 \times CF_{Con}(F_c) \end{aligned} \tag{2.11}$$

It is not easy to do a minimization of this function to get the lowest cost, as there is a lot of uncertainty about all variables. Still, it might be interesting to see some results when default values are chosen (table 2.2).

Parameter:	Default value:	Description:
γ	$\frac{1}{120}$ changes/second	The expected frequency at which changes occur in the data.
S_d	100 bytes	The average size of a single data item.
F_c	$\frac{1}{300}$ checks/second	The frequency at which a client checks for changes in the data.
C_o	150 bytes	The communication overhead used when checking for changes.

Table 2.2: Parameters and their default values

Note that in the next figures, $\frac{1}{F_c}$ (the time between two checks) is plotted instead of F_c. The function $W(x)$ (see equation 2.6) simply returns x, $C_o = 150$, $S_d = 100$, $\gamma = \frac{1}{120}$ and all weights w_1, w_2 and w_3 are set to 1. The result is figure 2.16.

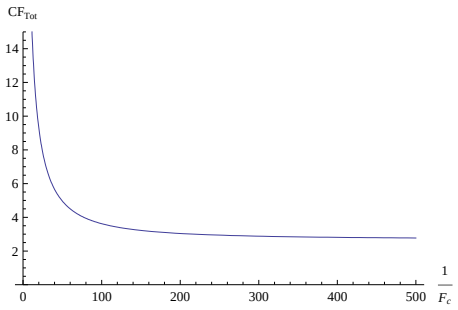

Figure 2.16: Total cost using $\frac{1}{F_c}$. $\gamma = \frac{1}{120}$, $C_o = 150$, $S_d = 100$, $W(x) = x$, $w_1 = 1$, $w_2 = 1$, $w_3 = 1$.

However, the weights are not very realistic, as the minimum of CF_{Tot} with these values is reached when $\frac{1}{F_c} \to \infty$. If we emphasize on consistency and set $w_3 = 10$,

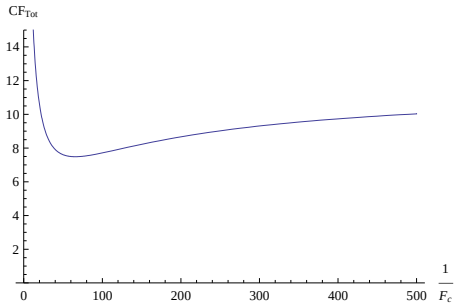

Figure 2.17: Total cost using $\frac{1}{F_c}$. $\gamma = \frac{1}{120}$, $C_o = 150$, $S_d = 100$, $W(x) = x$, $w_1 = 1$, $w_2 = 1$, $w_3 = 10$.

the result is figure 2.17 with a minimum cost at $\frac{1}{F_c} \approx 66$.

It would be interesting to see the relation between the three weights. If all other variables had their default values (including $W(x) = x$), the cost function would become:

$$CF_{Tot}(w_1, w_2, w_3) = \frac{4}{3}w_1 + \frac{5}{6}w_2 + \frac{5}{7}w_3 \tag{2.12}$$

Finally, the relation between $\frac{1}{F_c}$ and w_3 is plotted in figure 2.18, where $w_1 = 1$ and $w_2 = 1$. The corresponding cost function is:

$$CF_{Tot}(F_c, w_3) = \frac{5}{3} + 150F_c + \frac{w_3}{1 + 120F_c} \tag{2.13}$$

To find the most realistic values for w_1, w_2 and w_3 is outside the scope of this thesis, but I would recommend a high w_3 compared to w_1 and w_2.

2.5 Comparison

The various methods to ensure consistency (IMS, TTL, IRs and Lease, see section 2.1.2) are hard to compare using a cost function. It is dependent on the way they are implemented. Often other protocols use the features of one or more of these protocols. They also have a poor consistency in the Connect-It environment. This combination makes it uninteresting to look further into them.

The regular replication protocols also can't guarantee consistency in a mobile environment, or require much more communication to do this. The Bengal database replication system doesn't fulfill the requirement that it is data storage independent. They also will not be looked into any further. A shared data space system is also not an option as clients need to be able to access the information when they need to and not only when both clients are connected to the internet.

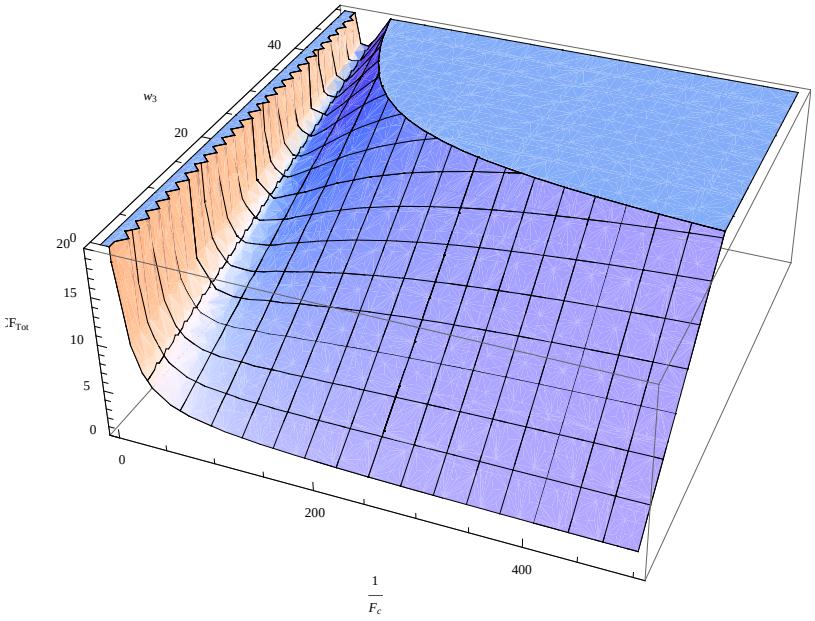

Figure 2.18: Total cost using $\frac{1}{F_c}$ and w_3. $\gamma = \frac{1}{120}$, $C_o = 150$, $S_d = 100$, $W(x) = x$, $w_1 = 1$, $w_2 = 1$.

What remains is the currently implemented protocol and the Roam replication system. The subsections 2.5.1, 2.5.2 and 2.5.3 will respectively compare the communication cost, the client processing time and the consistency between the currently implemented protocol and Roam.

2.5.1 Communication

The protocol that is currently implemented transfers everything every time it checks for updates. The time between two checks is very high, ranging between 5 minutes and 4 weeks. On average, this will be 1 day. Note that the frequency is fixed per datatype, but different data types can have different frequencies. The communication overhead is very low, no communication except the transfer of the data is needed, say 10 megabytes. However, as the protocol doesn't transfer any data per change, all transfers can be seen as overhead while no data is transfered per change. This results in:

$$CF_{Comm}(\frac{1}{86400}, 10485760, 0) = \frac{16384}{135} \approx 121,36 \tag{2.14}$$

Note that this is highly dependent on the total size of the data on the pda, but even if it was only 1 megabyte, it would still result in a much higher cost than when an optimal protocol would be used.

The Roam replication system uses client-client communication. This can be

translated as a double amount of communication needed when compared to an optimal protocol in a client-server architecture:

$$CF_{Comm}(F_c, 2C_o, 2S_d) = 2(F_c \times C_o + \gamma \times S_d) \tag{2.15}$$

It is obvious that the cost for this is twice the amount of an optimal protocol in a client-server architecture. However, it still is much better than the currently implemented protocol. Let's assume that the time between two checks is 5 minutes, the communication overhead is 150 bytes, there are 240 changes in 8 hours and the average size of 1 data piece is 100 bytes. For Roam this results in:

$$CF_{Comm}(\frac{1}{300}, 300, 200) = \frac{11}{3} \approx 3,67 \tag{2.16}$$

These values are realistic, but even if they are 10 times bigger and the total data on a pda would be only 1 megabyte, it would still result in a major cost reduction compared to the currently implemented protocol. However, it still is double the cost of what could be possible.

2.5.2 Client processing time

For the currently implemented protocol we have to rewrite equation 2.6, as it is independent of the number of changes but on the frequency of the checks (it transfers everything every time, even when nothing has changed):

$$CF_{Proc}(F_c, S_d) = F_c \times W(S_d) \tag{2.17}$$

The total amount of processing at the client using this protocol:

$$CF_{Proc}(\frac{1}{86400}, 10485760) = \frac{W(10485760)}{86400} \tag{2.18}$$

But what is the value for W? It is only known that it is monotonic, but an assumption can be made that it is a linear function: writing twice the amount of data will probably result in roughly twice the amount of processing needed:

$$W(x) = ax + b \tag{2.19}$$

To make it easy, say a is 1 and b is 0. If we apply this to all calculations, the costs can be compared relative to each other. For the currently implemented protocol:

$$\frac{W(10485760)}{86400} = \frac{16384}{135} \approx 121,36 \tag{2.20}$$

For Roam, equation 2.6 can be used, where γ is set to $\frac{1}{120}$ changes per second, and S_d to 100 bytes. Roam uses twice the amount of data and this results in:

$$CF_{Proc}(2S_d) = CF_{Proc}(200) = \frac{5}{3} \approx 1,67 \tag{2.21}$$

Again, twice the cost of what could be possible, but still a major performance gain compared to the currently implemented protocol. Note that the same W is used for both protocols. In reality, Roam will use more client processing power as it uses a more complicated method of distributing the data. The clients also have to determine whether there are changes etcetera. But again, even if the cost would be ten times bigger, it would still be a big improvement.

2.5.3 Consistency

As mentioned in section 2.5.1 the currently implemented protocol uses an average frequency of one check each day. Again, γ is set to $\frac{1}{120}$ changes per second.

$$CF_{Con}(\frac{1}{86400}) = \frac{720}{721} \tag{2.22}$$

This represents a very inconsistent system, where $c_i \not\subseteq s$ holds almost all the time, but is it fair to use the 24 hour delay in consistency when most changes will probably occur only during the 8 hours of a working day? Also, most data items that change regularly have a higher frequency, only the data items that change infrequently have frequencies of one check every few days or weeks. So the real or average consistency will be better than is shown above, but in the worst case scenario, there are data items that are indeed inconsistent for days or weeks. Still, it might be interesting to see if there was an average check for everything each hour, this would result in:

$$CF_{Con}(\frac{1}{3600}) = \frac{30}{31} \tag{2.23}$$

For Roam, which would be checking for changes about every 5 minutes, we get:

$$CF_{Con}(\frac{1}{300}) = \frac{5}{7} \tag{2.24}$$

Again we see a big improvement over the currently implemented protocol, but there is still room for improvement. If we check every minute (the optimum in figure 2.17), the cost would be:

$$CF_{Con}(\frac{1}{60}) = \frac{1}{3} \tag{2.25}$$

2.6 Summary

This chapter has analyzed the problem from chapter 1. It is clear that the currently implemented architecture does not fulfill the model described in section 1.3 and that the distribution protocol is not the optimal protocol. The best protocol is a replication method that replicates objects, not tables of a database. However, no such replication method exists that is suitable for a mobile environment like

Connect-It while fulfilling all requirements of section 1.4, although I think it is possible.

There are replication methods that work efficiently in a client-server architecture and in a mobile environment. There are replication methods that are data storage independent. There are data distribution methods that use very limited client processing time: only to store the data and no big overhead. Combine the best of these methods and the protocol for Connect-It is the result.

In chapter 3 I will present a design of a new architecture for Connect-It that deals with the challenges from section 2.2, fulfills the model from section 1.3 and has the lowest cost on the cost functions from section 2.4. Chapter 4 will analyze the results and check if the new architecture really is a solution for Connect-It.

Chapter 3

Design

In this chapter the design and implementation of the new architecture is described in detail. First, in section 3.1, the new architecture is described and the differences with the old architecture from section 2.3 are discussed. After that, section 3.2 goes into detail about the data filter mechanism. Section 3.3 gives some insight on the implementation details of the new architecture. Section 3.4 gives a summary of this chapter.

Note that everything in this chapter is my own work, including the new architectural design and the data filter mechanism. Where needed, I implemented everything myself.

3.1 Architecture

The new architecture will be shown using the 4+1 views of Kruchten [14]. For each view, the differences with the old architecture (see section 2.3) will be discussed.

3.1.1 Logical view

First the class diagram will be discussed, the sequence diagrams follow after that.

Class diagram: problem

The analysis in section 2.3.1 showed that three aspects needed to change:

1. The connection with the server and how the data is transmitted should be transparent to the PdaApplication.

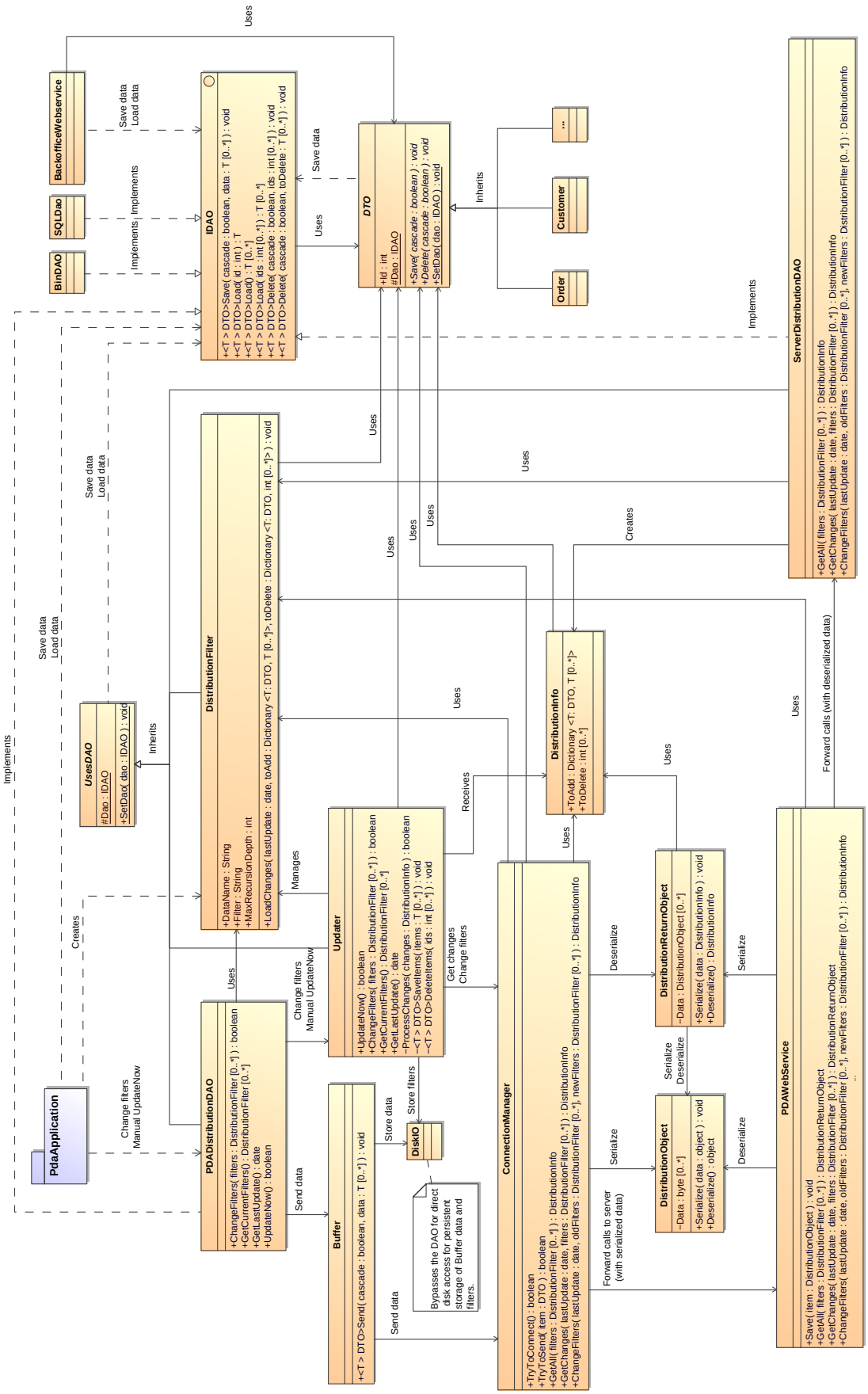

Figure 3.1: Class diagram (design)

2. There should be no dependencies to the derived DTO classes (like Order and Customer).

3. There should be only one DAO implementation per type (binary text files, SQL database, ...).

Class diagram: solution

Figure 3.1 shows the new class diagram. Note that only the relevant classes within Connect-It are shown. These are the major changes that have been made to address the above aspects:

- A new DAO has been introduced: PDADistributionDAO. It follows the same DAO interface as the other DAOs, but behaves a little bit different. Instead of storing the data in a datasource, it forwards all calls to another DAO. A call to *Save* will not only be forwarded, but also triggers a transfer to the server.

  A big advantage of using the same DAO interface is the ease of integration. The code only has to be rewritten in the previous parts that dealt with the distribution of the data, the rest stays the same.

- Using generic programming techniques, it was possible to remove all dependencies to the derived DTO classes. Note that for programmers, everything is still typesafe. When needed, the abstract DTO objects are inspected at runtime to determine what data they contain. Sections 3.3.1 will go more into detail about this.

- Using the same generic programming techniques, it was possible to rewrite the existing DAOs to a single class per type, without changing their behavior.

Below, each class is addressed to discuss its purpose and if applicable, the changes that have been made to it:

- **PdaApplication**
  This package represents the rest of the PDA application. It no longer directly communicates with the server, instead, data is automatically scheduled to be sent when saved to disk using the regular DAO interface calls. The PDA application is able to set certain filters for the DistributionDAO and all data that passes the filter will automatically be synchronized to the PDA data storage.

- **IDAO**
  This is the DAO (Data Access Object) interface. It provides generic load,

save and delete functions to the data storage used. Note that the generic methods provide typesafety while they are not dependent on the specific data types.

- **BinDAO**
  This is the DAO for accessing data using binary files. Because of the generic implementation, it is no longer necessary to implement this DAO for each data type.

- **SQLDAO**
  This is the DAO for accessing data using a SQL database. Because of the generic implementation, it is no longer necessary to implement this DAO for each data type.

- **DTO**
  DataTransferObject. This abstract class is the base for all data classes such as Order and Customer. Each record in the database is represented by a DTO object. There is a static DAO that all child classes will use.

- **Order**
  This class represents order data, it inherits from DTO.

- **Customer**
  This class represents order data, it inherits from DTO.

- **...**
  Represents other classes that inherit from DTO, just like Order and Customer.

- **UsesDAO**
  This abstract class is the base for four other classes: PDADistributionDAO, Updater, DistributionFilter and ServerDistributionDAO. It has a static DAO that all child classes will use. Note that the ServerDistributionDAO is located at the server and can therefore have a different DAO. The DistributionFilter will only use its DAO at the server.

- **PDADistributionDAO**
  This is the DAO that the PdaApplication will use to load, save and delete data. When saving data, that data is also passed to the Buffer for transfer to the server. All DAO calls are simply forwarded to the next DAO.

- **DistributionFilter**
  This class represents a filter and is used in the update process. It can check for changes in the data that passes the filter since the last update. See section 3.2 for more details on this.

- **Buffer**
  The buffer receives data that must be send to the server. It will persistently store that data on disk using DiskIO, and tries to send the data as soon as possible to the server.

- **Updater**
  The updater periodically checks for changes at the server, given the filters which are stored on disk.

- **DiskIO**
  Provides persistent storage for the Buffer and the Updater. This class directly writes to disk and doesn't use any other DAO.

- **ConnectionManager**
  Manages the connection to the server. Because a webservice can't use generics, all data is serialized into DistributionObjects and the return values are deserialized from DistributionReturnObjects.

- **DistributionInfo**
  This class holds the changes that need to be applied to the PDA data storage. It contains new data and information about what data to delete.

- **DistributionObject**
  Generic objects that are sent from the PDA to the server are serialized into an array of bytes by this class.

- **DistributionReturnObject**
  DistributionInfo objects that are sent from the server to the PDA are serialized into an array of DistributionObjects.

- **PDAWebService**
  This is the PDA webservice running at the server, all PDA's will use this webservice to communicate with the server. This webservice no longer supports the DAO calls, but acts as a connector between the ConnectionManager and the ServerDistributionDAO.

- **ServerDistributionDAO**
  This class runs at the server and can use the DistributionFilters from a PDA to get information about what that PDA needs to do to be synchronized again.

- **BackofficeWebservice**
  This is the backoffice webservice running at the server, all backoffice clients will use this webservice to communicate with the server. It has not been changed in the new architecture.

Sequence diagrams: problem

The analysis of the class interactions in section 2.3.1 showed four necessary changes:

1. Sending data to the server is optional. This presents a big risk: some changes might never reach the server while other changes might be dependent on them.

2. Connection failures during a transfer are not handled properly.

3. All knowledge about what data to transfer to the server exists in memory alone. This should be persistently stored on disk, so an uncontrolled shutdown does not result in data loss.

4. For performance reasons, only the changes in the data should be transferred. It is not necessary to transfer data that has not been changed and already exists in the clients local storage.

Sequence diagrams: solution

The use of the DistributionDAO changes the way the processes behave. The PDA application doesn't need to send the files explicitly, each call to the Save function will have that as a result (see figure 3.2). Notice that when a data item arrives at the Buffer, the Buffer will also persistently store the data item so changes are never lost.

The Buffer sequence diagram (figure 3.3) shows how the generic data items are serialized and deserialized during transfer. It also shows the conflict detection when there is a conflict. The actual resolve algorithm will trigger a conflict event and a subscriber to that event can store both conflicting items in a separate place while sending a message to a specific user or user group.

The Update sequence has also been drastically changed (figures 3.4 and 3.5). It is no longer possible for a programmer to manually connect to the server to request data. Furthermore, instead of deleting all data and downloading it again, only the changes are transferred now. The result will be less data transfer and all data is kept up to date with the same high check frequency, improving consistency. The datafilters play a big role in this, see section 3.2 for more information.

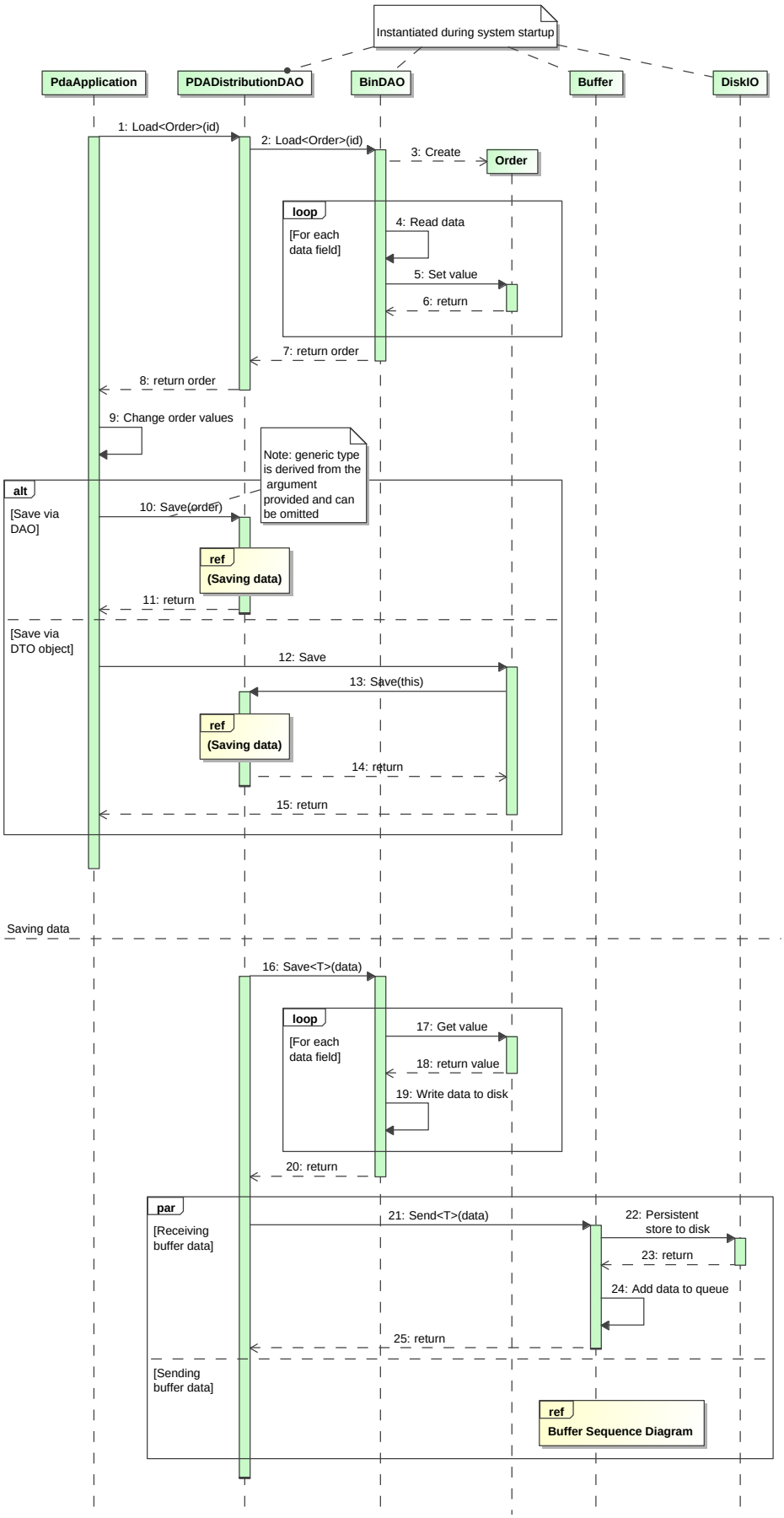

Figure 3.2: Application sequence diagram (design)

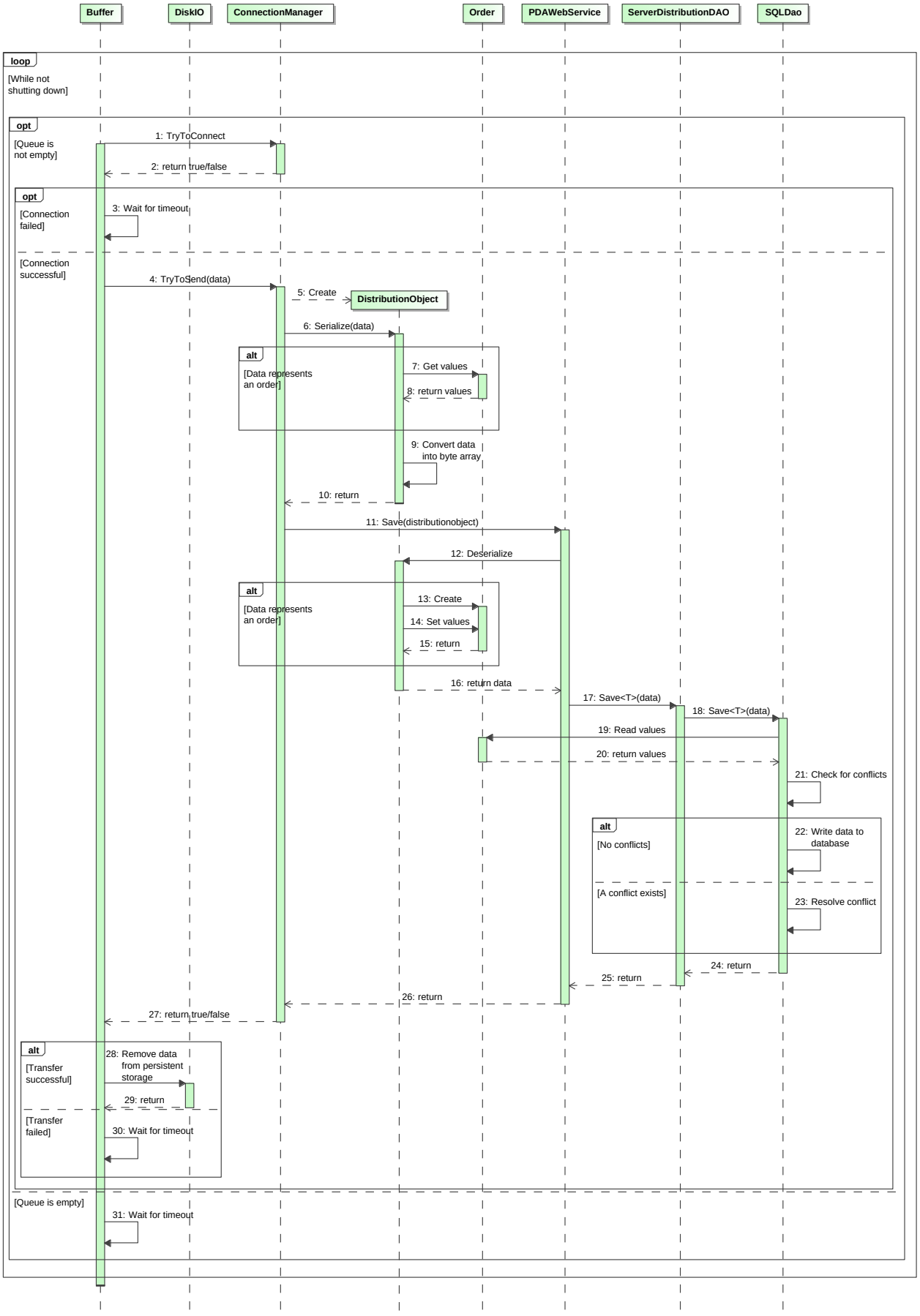

Figure 3.3: Buffer sequence diagram (design)

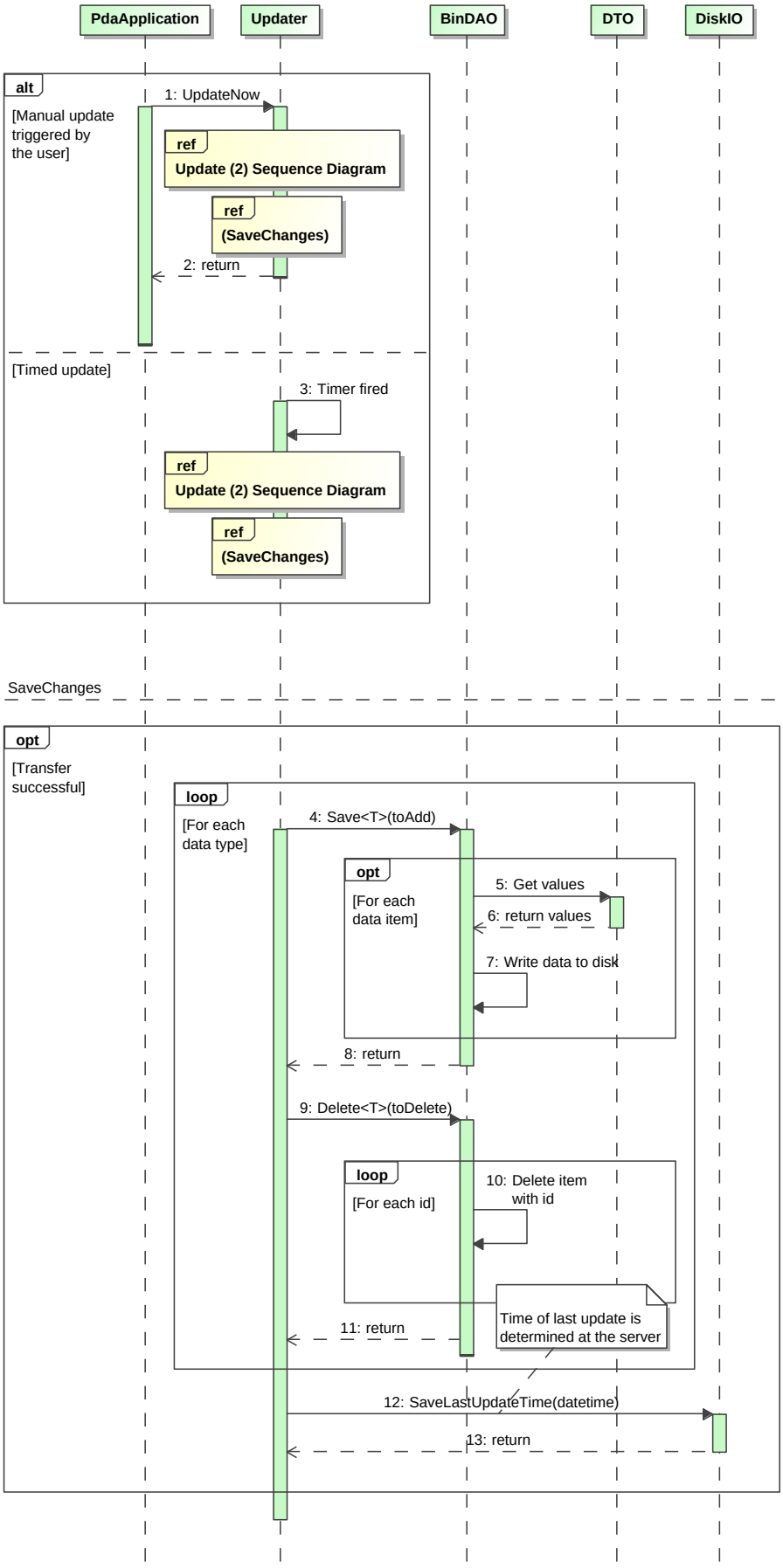

Figure 3.4: Update (1) sequence diagram (design)

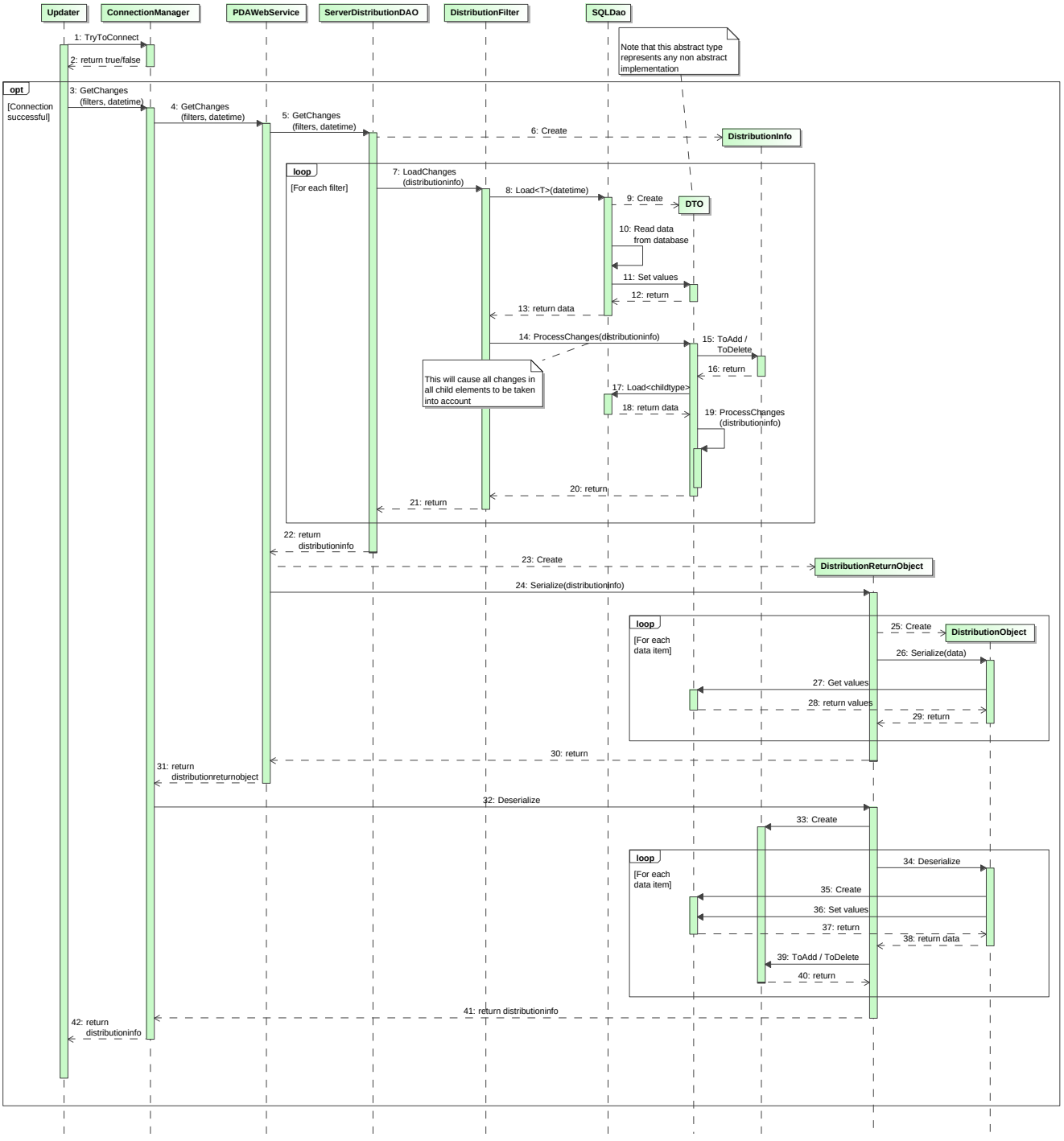

Figure 3.5: Update (2) sequence diagram (design)

3.1.2 Process view

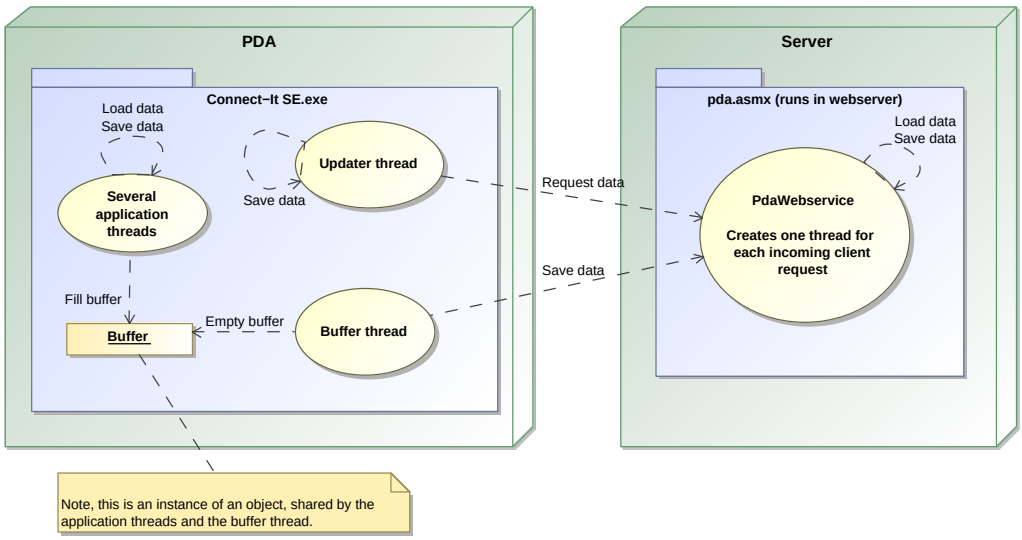

Figure 3.6: Processes and threads (design)

The analysis in section 2.3.2 showed a potential problem where responsiveness might be reduced at the user level because application threads could block when a server thread would block. The new architecture solves this because direct connections to the server are not allowed or possible anymore. Figure 3.6 shows the new threads and their interaction.

3.1.3 Development view

Problem

The analysis in section 2.3.3 showed two aspects that needed to change:

1. Use middleware to abstract the distribution of the data.
2. The component DataObjects.dll is application specific, but there are many (unnecessary) dependencies to it from the middleware layer.

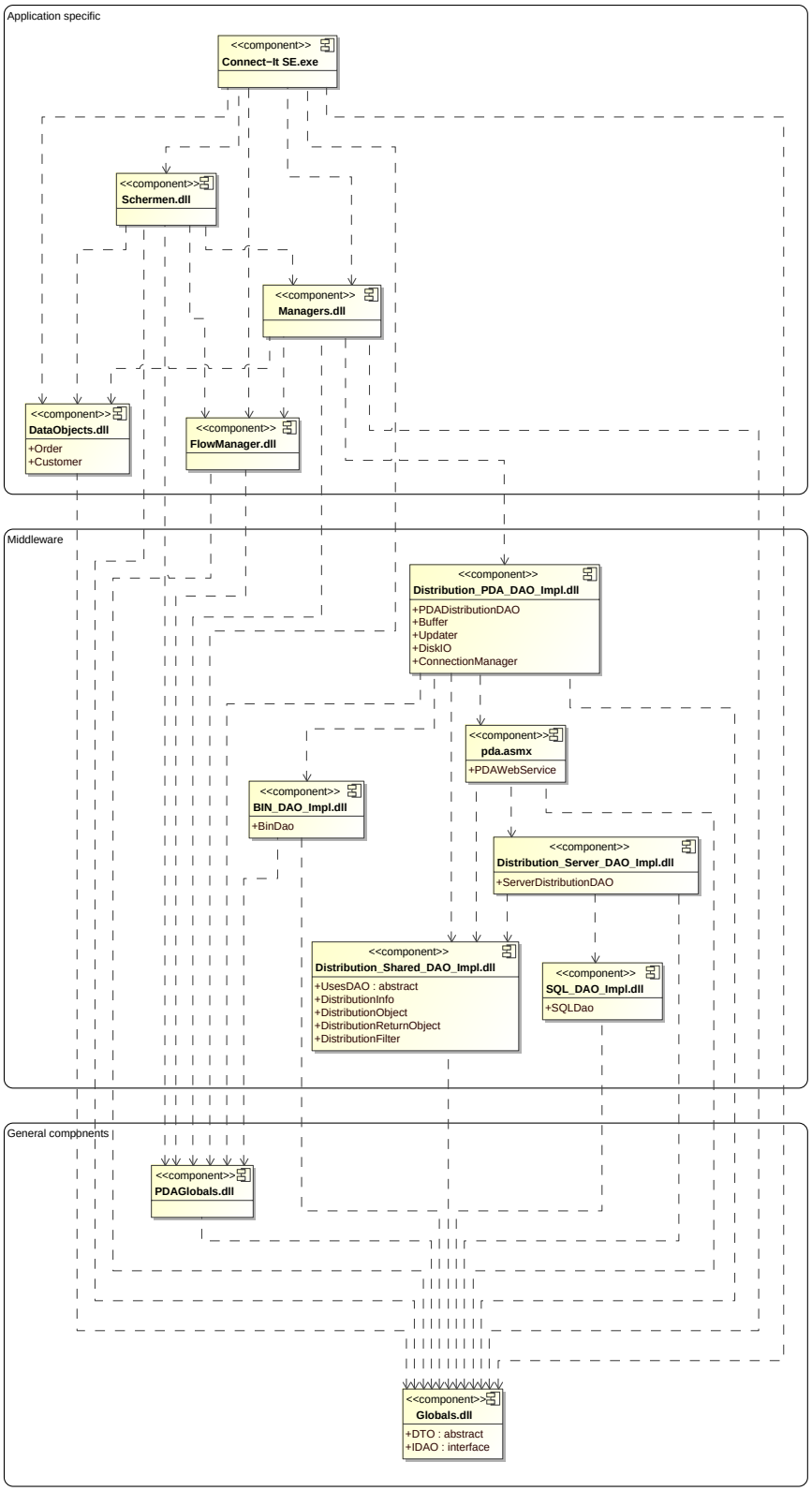

Figure 3.7: Component dependencies (design)

Solution

Figure 3.7 shows the components and their relation in the new architecture. In the middleware layer, there are three new components: Distribution_PDA_DAO_Impl .dll, Distribution_Server_DAO_Impl.dll and Distribution_Shared_DAO_Impl.dll. These components implement the behavior as described in section 1.3. All communication between a PDA and the server is done using these components (requirement 2 from section 1.4). Because these components are in the middleware layer, they are application independent (requirement 3 from section 1.4). Furthermore, the component DataObjects.dll has been moved from the general components layer to the application specific layer. There are no more dependencies from other layers to DataObjects.dll. The result is that DataObjects.dll can be changed, without affecting the other layers (requirement 4 from section 1.4). This has been accomplished by using generic programming techniques, see section 3.3.1 for more information.

3.1.4 Physical view

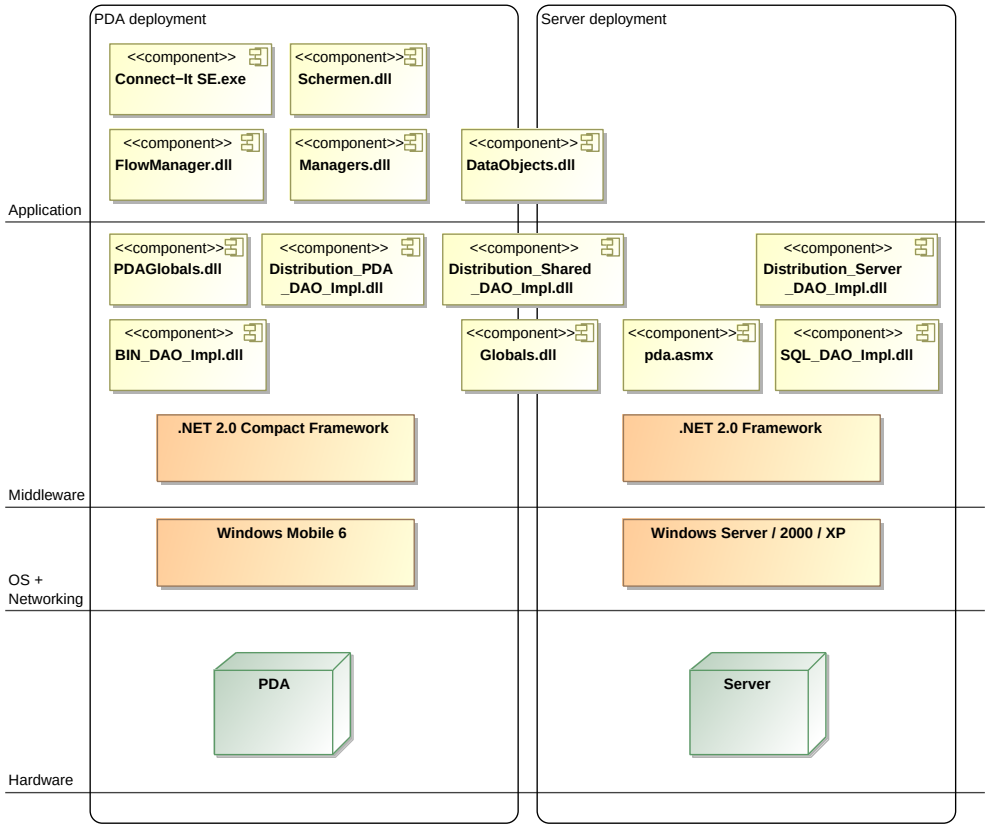

Figure 3.8: Deployment (design)

Figure 3.8 shows the new deployment diagram. The component Distribution_PDA_DAO_Impl.dll will be deployed on the PDAs, the server will deploy Distribution_Server_DAO_Impl.dll and they both deploy Distribution_Shared_DAO_Impl.dll.

3.1.5 Scenarios

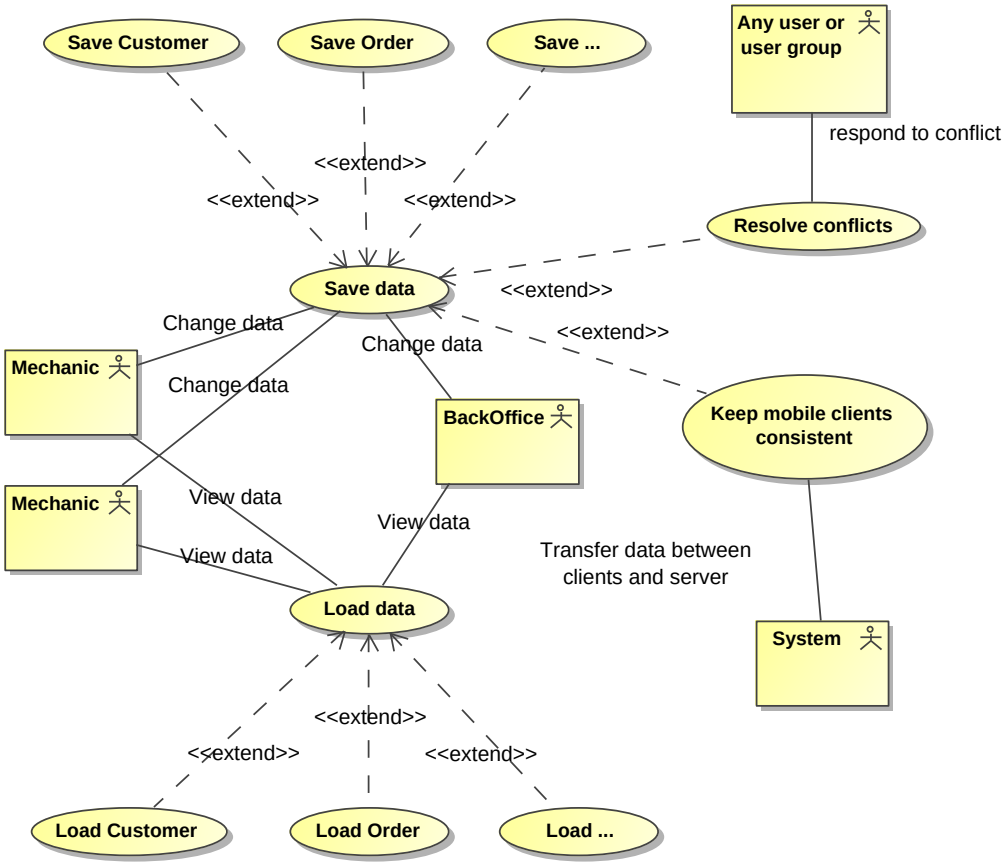

Figure 3.9: Use case diagram (design)

From a use case point of view, no changes were done to the system as a whole, except that it is now possible to respond to a conflict. This is illustrated in figure 3.9.

3.2 Data filters

3.2.1 Introduction

The problem I encountered during the design of the architecture in section 3.1 was that it was required (for performance reasons) to only transfer changes in the data during the *Refresh* operation (see section 1.3). However, the server does

not contain the knowledge of what data each client has stored in its local storage. One could argue that the server might keep track of this, but a quick analysis showed that it would generate too much overhead at the server side, reducing scalability.

What I came up with was a filter mechanism where each client could define a set of filters by which the server could identify what the changes in the data are (both new or changed data, as well as deleted data).

Data filters make sure only the minimum set of data - given the filters - is transferred to the clients. They prevent whole database tables to be distributed and allow clients to specify what data is relevant for them, using a flexible query language.

What these data filters accomplish is basically a mix of object replication (see section 2.1.3) and IMS (see section 2.1.2). The difference with regular object replication is that the filters don't work on a fixed set of objects, as a client doesn't need to have knowledge of the existence of certain objects. Furthermore, it is possible to define filters based on relationships between objects, creating a very flexible system.

I explicitly chose not to use SQL as a query language for two reasons. First, the language should be implementation independent, not based on the knowledge that the server currently uses a SQL database. The fact that the server uses a SQL database is abstracted by the use of the DAO layer, therefore the query mechanism should be abstracted to work with any DAO implementation. Secondly, I wanted to keep the query language as simple as possible for the clients and at the same time allow much more expressiveness in a single query. One simple query in this language can represent a very complex SQL query because all related data is also queried; see the examples below for more information.

Note that a single query could result in a large amount of data: the query "Administration,true" will return all data in all administrations. This is the mechanism that ViaData prefers over a regular query language where only the explicit data that is queried for is returned.

3.2.2 Syntax

The syntax of a data filter is as follows (in Backus-Naur Form):

$$
\begin{array}{lll}
<filter> & ::= & <datatype> \text{","} <condition> \\
<datatype> & ::= & <text> \\
<condition> & ::= & \text{"}true\text{"} \mid \text{"}child\text{("} <filter> \text{")"} \mid <expression> \\
 & & \mid \text{"}exclude\text{("} <filter> \text{"),"} <condition> \\
<expression> & ::= & <and> \mid <or> \mid <value> \\
<and> & ::= & \text{"}and\text{("} <expression> \text{","} <expression> \text{")"} \\
<or> & ::= & \text{"}or\text{("} <expression> \text{","} <expression> \text{")"} \\
<value> & ::= & \text{"}value\text{("} <property> \text{","} <operator> \text{","} <text> \text{")"} \\
<property> & ::= & <text> \\
<operator> & ::= & \text{"}==\text{"} \mid \text{"}!=\text{"} \mid \text{"}<\text{"} \mid \text{"}<=\text{"} \mid \text{">"} \mid \text{">="}
\end{array}
$$

Some remarks:

- $<text>$ represents a text string without white spaces.
- The data type text must be the name of an existing data type, such as "Order" or "Customer".
- The condition "true" means that all data of that data type will pass the filter.
- The child filter must apply to a data type that is a child of the parent data type, like Order is a child for Customer.
- When an exclude filter is specified, no data that passes the exclude filter (no related child recursion) will pass the original filter.
- The property text must be the name of a property of the data type to which the condition applies, such as "Name" for Customer.
- The value text must be of a proper format for the property it applies to, like a formatted date string for OrderDate.
- When an object passes a filter, all children (and their children) of that object automatically pass that filter as well. So when a customer passes a filter, all orders of that customer pass that filter. This works for all 1-many relations in the data. To stop recursing over children, use the exclude filter or specify a max recursion depth (property of the *DistributionFilter* object).
- The filters can be generated at runtime, depending on which employee is using the PDA and what orders he is working on.

Some examples of filters:

- **Filter:** Employee,true
  Description: Passes all employees, and all data related to those employees (such as assigned orders, all data related to those orders, etc).

- **Filter:** Order,value(EmployeeId,==,4)
  Description: Passes all orders assigned to employee with id 4, and all related data to those orders.

- **Filter:** Customer,exclude(Order,value(OrderDate,<,"1-1-2000")),true
  Description: Passes all customers, and all data related to those customers, except orders with an *OrderDate* before 1-1-2000.

- **Filter:** Customer,child(Order,and(value(State,==,"open"),value(EmployeeId,==,-1)))
  Description: Passes all customers that have open orders that are not yet assigned to an employee, and all data related to those customers.

3.2.3 Algorithms

Initial request

Initially, the client storage is empty and all data needs to be requested. The *Updater* calls the method *GetAll* on the *ConnectionManager* together with one or more filters as argument. If there is a connection possible, the call is then forwarded via the *PdaWebService* to the *ServerDistributionDAO*. The *ServerDistributionDAO* will trigger each *DistributionFilter* to load all data that passes the filter. When all filters are done, the server will return the resulting data set to the *Updater* who stores it in the local storage.

Requesting changes

Every Δt the *Updater* wants to receive changes in the data. The *Updater* calls the method *GetChanges* on the *ConnectionManager* together with one or more *DistributionFilter*s and a date/time the last update took place as arguments. If there is a connection possible, the call is then forwarded via the *PdaWebService* to the *ServerDistributionDAO*. The *ServerDistributionDAO* will trigger each *DistributionFilter* to load all changes in the data that passes the filter (based on the provided time stamp). These changes can be newer versions of the data or a message that certain data should be deleted (compare it with an invalidation report in caching schemes [22, 8]). The changes are then returned to the *Updater* who stores them in the local storage. The sequence diagrams are shown in figures 3.4 and 3.5.

Changing filters

Once every while the PDA application needs to change his filters, for example when a new employee logs in. The local storage then needs be updated with the data that passes the new filters. The *Updater* calls the method *ChangeFilters* on the *ConnectionManager* together with one or more new filters, the old filters and a date/time the last update took place. If there is a connection possible, the call is then forwarded via the *PdaWebService* to the *ServerDistributionDAO*. Now the *ServerDistributionDAO* has to compute the difference between the filters.

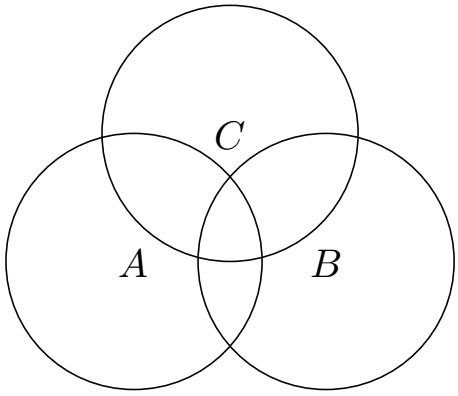

Figure 3.10: *ChangeFilters*: situation at the server

Figure 3.10 illustrates the situation at the server, where A represents the data currently at the client, B represents the changes since the last update (this can be changes in existing data, deletion of existing data or addition of new data) and C represents the data that passes the new filters and this is what the client is interested in.

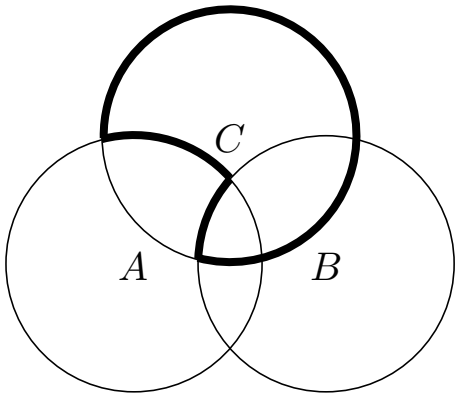

Figure 3.11: *ChangeFilters*: data that needs to be transferred "*ToAdd*"

Because one of the main goals is to minimize the communication overhead, we don't want to delete everything at the client and send all new data again, because much can overlap. What we do want to transfer, is everything that is new, or existing data that will still be used but has changed since the last update, as shown in figure 3.11.

In a more formal way, the data the client has to add to its storage is:

$$toAdd = C \backslash (A \backslash B) \tag{3.1}$$

But the server doesn't know what data the client currently has (A), it only knows what data should be on the client now using the old filters, and it knows what changes have taken place since the last update. This is not a big problem, because what the server knows (using the old filters) is $A \cup B$, and there is no difference in $A \backslash B$ and $(A \cup B) \backslash B$.

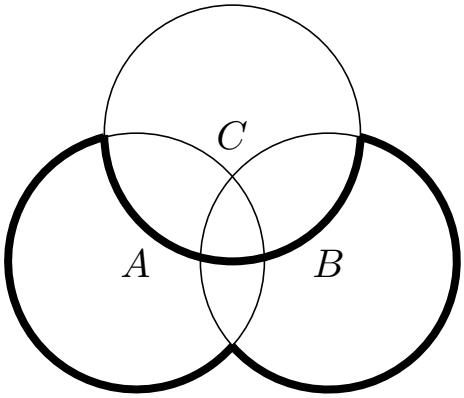

Figure 3.12: *ChangeFilters*: data that needs to be deleted "*ToDelete*"

The data that needs to be deleted is everything that is in the client storage what has become obsolete. The problem is that the client doesn't know what is obsolete or not, so the server has to specify this: delete $A \backslash C$. Here we see the same problem again: A is unknown. So the message will become:

$$toDelete = (A \cup B) \backslash C \tag{3.2}$$

Figure 3.12 illustrates this. The resulting *toAdd* and *toDelete* are returned to the *Updater* who can store everything in the local data storage.

3.3 Implementation

3.3.1 Generics and reflection

The implementation was done using Microsoft Visual Studio 2005 in C# .NET 2.0. One of the big advantages of C# .NET 2.0 (over C# .NET 1.1) is the use of generics and reflection. This allows code to be written that uses parameters or return values of generic types in stead of explicit types. The difference with C++ templates, is that the generics from C# are more flexible and readable. Furthermore, while the application is running, these objects of generic types can be observed using the reflection techniques to gather information about them and call methods that were dynamically found. For example, the save function in the DAO Layer might look as follows:

```
public void Save<T>(bool cascade, params T[] data) where T : DTO, new()
{
    SaveItems<T>(cascade, data);
}
```

Here T is a generic type, with the requirements that it inherits from the abstract *DTO* type (Data Transfer Object, see [2]) and it has an empty default constructor: it can be instantiated using

```
T obj = new T();
```

This prevents abstract types to be passed as a type parameter to this method. A possible call to this save method can be:

```
Order o = ... ;
Save<Order>(true, o);
```

Note that these methods are all typesafe; a *Save* call with type parameter *Order* only accepts orders as a parameter. A *Load* call with type parameter *Customer* will only return a customer object (or *null*). A compiler error will occur if a programmer tries otherwise. Because all methods require a type parameter, it is no longer necessary to write a DAO for each datatype, now there is only one DAO per data storage. Currently there is a BIN DAO for binary files and a SQL DAO for a SQL database. Creating a new DAO, for example for XML files or another database type, should be fairly easy.

Because an object can be saved using different public method calls, the actual save implementation is done in a private method called *SaveItems*. There T is analyzed to discover which fields and properties it contains that need to be saved. A simple version of the actual implementation (using C# reflection techniques) is:

```
foreach (T element in data)
{
    foreach (string field in element.GetDBFields())
    {
        object value = typeof(T).GetProperty(field).GetValue(element, null);
        switch (element.GetSqlType(field))
        { . . . save value . . . }
    }
}
```

The only thing known is that an object of type T has a method called $GetDBFields$ which returns a list of field names that need to be saved and a method called $GetSqlType$ that returns the SQL type of the specified field name. Even in a non SQL environment, this SQL type information can be useful, for example to determine the maximum allowed length of a string. In a SQL environment this information can be used to dynamically build the necessary queries.

Generics combined with reflection is a very powerful programming method; the example above is just the beginning. A $Customer$ object has a property called $Child\_Orders$ which contains zero or more orders belonging to this customer. A cascaded save of a customer will also save these orders (and their children). Determining if an object of type T contains such properties, saving them and saving every child is done dynamically, without any knowledge of what types of objects are being processed. At compile time, it is not known in the DAO that Customer has a property called $Child\_Orders$. In the same way, methods belonging to an object of type T can be found and called. The only requirement is that the data objects follow certain rules, see section 3.3.2. These rules are enforced in the abstract DTO type which is located in the $Globals.dll$ component.

Implementing the Distribution Layer with the use of generics and reflection allows the programmers of Connect-It to change the database without changing anything to the Distribution Layer or the DAO Layer. In the previous version of Connect-It the DAO Layers and the PDA WebService had to be rewritten after every change in the database, as each type had its own DAO methods.

The fixed interface of the DAO Layer enables the use of different DAO Layers depending on the data storage used. Switching from a set of XML files to a SQL database is nothing more than instantiating another DAO Layer in the program start-up routine, as long as the DAO Layer for that data storage exists of course. Writing a new DAO Layer using generics has become much less work, as there is only one method for each possible call (save/load/delete/etc).

3.3.2 Code generator

First an introduction to the code generator will be given, after that the syntax is explained.

Introduction

Because of the generic implementation, the Distribution layer and DAO layers don't have to be adjusted anymore if there is a change in the database design. The data types like *Customer* still require that the changes are also applied to them, or errors will occur in the DAO layer. To make things easier a code generator has been developed. This code generator generates the C# .NET 2.0 implementation code for each database table. Every table becomes an object type, like *Customer* and *Order* (see the class diagram in figure 3.1).

Some notes about the code generator:

- **Use case:** The database has changed: tables have been added, removed or changed.
- **Input:** A SQL database and a file containing a code template.
- **Activity:** Parse the template and apply it to every table in the database. Displays a warning if database design errors could result in runtime errors in the DAO layer.
- **Output:** The code for all data objects in the *DataObjects.dll* component.
- **Openness:** Changing all classes in the *DataObjects.dll* component is now a matter of changing one template, the generator will do the rest.
- **Security:** Improves data integrity: database design errors are detected during the generation of the code.
- **Scalability:** Improves the scalability of Connect-It: code will be generated instead of manually written every time the database design changes.
- **Transparency:** Once the template is written, the programmer who uses the generator doesn't need to know all specifics, extending the *DataObjects.dll* component with new classes has become transparent.

Besides the obvious advantage of reducing the work for the programmers, the code generator has another big advantage. In the database, an order table for example, has a foreign key to a customer record. The naming of these keys follows certain rules required by the DAO layer. To ensure all of these rules, the code generator generates the implementation code, including all cross references to its parents and children. This way, programmers can use:

```
Customer c = ... ;
Orders[] o = c.GetChildOrders();
```

This is a very easy way of loading all orders that belong to a certain customer. All database relations are translated this way, making it very easy to work with the data as objects.

If something is encountered after a change in the database that might result in errors in the DAO layer, the code generator displays a warning and the programmers can correct the error in the database design. An example: some fields in the database might need the requirement that they cannot be *null* as the corresponding types in C# are not allowed to be *null*. The code generator can give a warning to notify the programmers that an error might occur if the database tables are used like this. It is unlikely that an error will occur, as no *null* values can be written by the DAO layer if C# doesn't allow them for that type, but somebody might manually edit the database and forget about this detail.

The code generator follows a template that it applies to every table it encounters. The template uses tags that can't exist in the programming code. The code generator replaces the tags with the corresponding information. These tags can be anything from class information to specific field information. For example:

```
public partial class <$classname> : DTO
{ ... }
```

Besides simple tags, advanced scripting can be done, such as a foreach statement:

```
public override string[] GetDBFields()
{
    string[] fields = new string[<$nrofmembers>];
    <$foreach#member>fields[<$counter>] = "<$membername>";
    <$end_foreach>return fields;
}
```

Or If statements:

```
<$foreach#member>
  <$if#is_not#membername#Id>
    <$if#is#membertype#int>
      . . .
    <$else>
      . . .
    <$end_if>
  <$end_if>
<$end_foreach>
```

This allows every possible construction of the class data as required. The code generator parses the template and executes the corresponding scripts and replaces the data tags with the correct information. Writing the template is basically writing a generic class, with options for every possible type. Once the template is finished, it doesn't need to change often, only when a new DAO layer requires new functionality that doesn't fit in the abstract *DTO* type.

Type specific information, such as a method that can calculate the order total given its items and values, can be done in separate partial classes, also a new feature of C# .NET 2.0. A partial class allows the code of one class to be split among several files. As seen above, every class generated is a partial class. All generated code can be put in one file as nobody should need it. If a custom method needs to be added to a type, a separate file just for that type can be written and it only needs to declare the extra methods that are needed. When a change in the database is made, the code generator can overwrite the previously generated code without overwriting the custom designed methods. This gives the programmers a lot of flexibility to work with, and they can focus on the important work in stead of the trivial work.

Syntax

The syntax used for a template for the code generator is as follows (in Backus-Naur Form):
<*template*> ::= <*text*> <*template*> | <*tag*> <*template*> | ""
<*tag*> ::= "<$" <*tagname*> <*tagoption*> ">"
<*tagname*> ::= <*text*>
<*tagoption*> ::= "#" <*tagoptionname*> <*tagoption*> | ""
<*tagoptionname*> ::= <*text*>

Below is a list of valid tag names and their options:

- **classname**
  Prints the name of this class. Options: none.
- **foreach**
  Start a "for each" loop: repeats everything between this tag and the end_foreach tag for each item specified. Requires one option:
  - **member**
  - **parent**
  - **child**

  Note that for each loops can't be nested.

- **end_foreach**
  Indicates the end of a "for each" loop. Options: none.
- **if**
  Starts a conditional statement which will be executed if the operator returns true. Requires two or three options. The first option is an equality operator:
  - **is**
    Returns true if the evaluation returns true, returns false otherwise.
  - **is_not**
    Returns false if the evaluation returns true, returns true otherwise.

  The second option can be an unary evaluation, or a binary evaluation. Possible unary evaluations:
  - **classhaschildren**
    Returns true if the current class has any children.
  - **memberisfk**
    Returns true if the current member (in a foreach member loop) is a foreign key to another table.
  - **parentallowfknull**
    Returns true if the current parent (in a foreach parent loop) allows null values for this child foreign key.
  - **first**
    Returns true if this is the first loop of a foreach loop.
  - **last**
    Returns true if this is the last loop of a foreach loop.

  Binary evaluations perform an operation based on the third option. Possible binary evaluations:
  - **membername**
    Returns true if the member name of the current member (in a foreach member loop) matches the specified value.
  - **membernamestartswith**
    Returns true if the member name of the current member (in a foreach member loop) starts with the specified value.
  - **membernameendswidth**
    Returns true if the member name of the current member (in a foreach member loop) ends with the specified value.
  - **membertype**
    Returns true if the member type of the current member (in a foreach member loop) matches the specified value.

  - **memberSQLtype**
    Returns true if the member SQL type of the current member (in a foreach member loop) matches the specified value.

  Note that it is possible to nest several if statements.

- **else**
  Must be placed between an if tag and end_if tag. Starts a conditional statement which will be executed if the original if equality operator returns false. Options: none. Note that each if tag has at most one else tag. In the case of nested if blocks, the else tag always belongs to the last opened if tag.

- **end_if**
  Indicates the end of an if tag. Options: none.

- **counter**
  Prints a counter for the current foreach loop, starting at 0 for the first loop. Options: none.

- **membername**
  Prints the current member name (in a foreach member loop). Options: none.

- **membertype**
  Prints the current member type (in a foreach member loop). Options: none.

- **memberSQLtype**
  Prints the current member SQL type (in a foreach member loop). Options: none.

- **memberSQLlength**
  Prints the size of the current member SQL type (in a foreach member loop). Options: none.

- **nrofmembers**
  Prints the number of members for the current class. Options: none.

- **membercanbefknull**
  Prints "true" if the current member (in a foreach member loop) can be a null value. Prints "false" otherwise. Options: none.

- **parentname**
  Prints the name of the current parent (in a foreach parent loop). Options: none.

- **parentnameinchild**
  Prints the name of the current parent (in a foreach parent loop) as it is known from the childs point of view. Options: none.

- **parentfkname**
  Prints the name of the foreign key of the current parent (in a foreach parent loop). Options: none.

- **childname**
  Prints the name of the current child (in a foreach child loop). Options: none.

- **childtype**
  Prints the type of the current child (in a foreach child loop). Options: none.

- **childfkparentname**
  Prints the parent foreign key name to the current child (in a foreach child loop). Options: none.

3.4 Summary

In this chapter a design has been made (section 3.1) for a new architecture for the problem described in chapter 1 which was analyzed in chapter 2. The data filter mechanism has been discussed (section 3.2) which allows Connect-It to use a replication method based on (properties of) objects and the relation of different objects in the database instead of the more common replication techniques that replicate on whole database tables. Some remarks addressing the implementation details were done in section 3.3.

Chapter 4 will analyze the results from this chapter to check if all requirements from section 1.4 are met.

Chapter 4

Results

In this chapter the results are analyzed. First, in section 4.1 several measurements are done. After that the new architecture is evaluated in section 4.2. Section 4.3 will give a summary of the results.

4.1 Measurements

In this section the real communication cost and client processing time are measured.

4.1.1 Communication

Overhead

When using a web service to transfer data, everything is turned into XML messages. This means that there is an XML header and everything gets an opening and closing tag. At first this might not seem relevant, but when you look at a data field called NumberOfItemsSold that normally consists of 4 bytes of data (one 32 bits integer), it will become very inefficient to use a web service. In XML, this data field would translate to:

```
<NumberOfItemsSold>123</NumberOfItemsSold>
```

So to transfer the number 123, one would be transferring 42 bytes: one byte per character. This is where the DistributionObject (see section 3.1) has a double function. The first one is to be able to make use of a web service when using generic objects, the second one is to reduce the amount of communication needed. When serializing the data into a byte array, only the real data is used, not the meta data such as field names. This greatly reduces the overhead when transferring data.

Setup

All measurements were performed using a HTC Touch Dual smartphone as a client, which has a 400MHz Qualcomm MSM7200 processor, 128MB RAM and Windows Mobile 6 Professional as operating system. It can connect to the internet using a HDSPA connection. The server was a PC consisting of an Intel Core 2 Duo E6600 2.4 GHz, 6GB RAM and Windows Vista Ultimate 64 as operating system and a glass fiber 100Mbit internet connection.

The client requests data from the server and at the server it is measured how much data is transferred over the internet connection using NetLimiter [3]. This tool allows the monitoring of the network usage of specific software applications. At the server side, specific scenarios were programmed so the measurements could be taken multiple times with the same data being transferred.

To measure the communication needed per data object, four scenarios are defined:

- A small data object with a minimal amount of data in it.
- A small data object with a maximal amount of data in it.
- A large data object with a minimal amount of data in it.
- A large data object with a maximal amount of data in it.

The minimum amount of data is where all string values are empty and all number values are 0. The maximum amount of data is the opposite: all string values are filled to their maximum and all number values contain the largest possible value. Table 4.1 illustrates this. The first data type represents an object which contains 5 data fields which can contain only fixed length numbers. The second data type represents an object which contains 40 data fields with a mix of numbers and strings. The minimum and maximum size of the data object is also shown. Note that most data objects with strings will probably never be the maximum size when they are commonly used as most string values will be much shorter than their maximum length.

Data type	Nr of fields	Min size (bytes)	Max size (bytes)
1	5	36	36
2	40	90	3590

Table 4.1: Data types used in the communication

Results

The first thing that is measured is the amount of bytes received at the client when checking for updates (figures 4.1 and 4.2). The second figure is the same as the first except the high values of data type 2 are not shown. This gives a better view of the other values. Each update contains 5 changes that are transferred to the client. Note that these changes are not unique, updates 4, 9, 15 and 16 contain an item that was changed twice, this explains the dips in the figures: there was 20% less data to transfer.

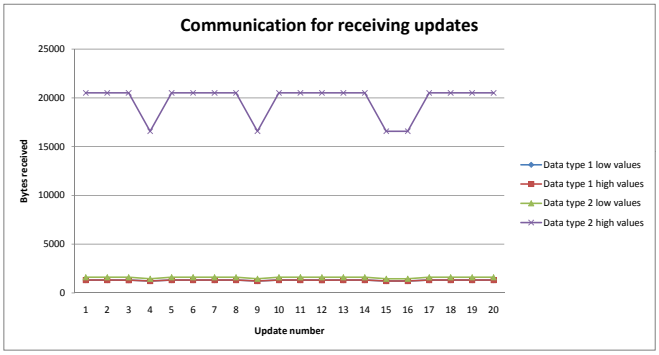

Figure 4.1: Communication for receiving data

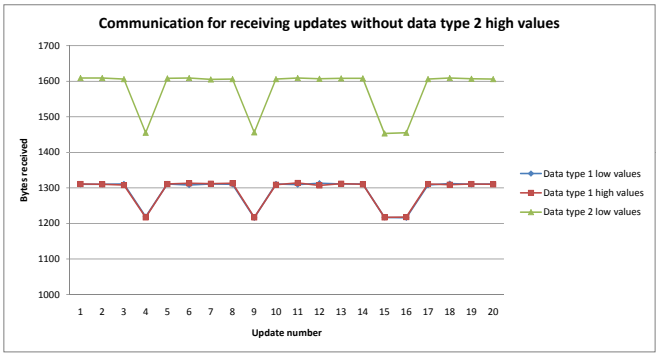

Figure 4.2: Enlargement of figure 4.1 (without datatype 2 high values)

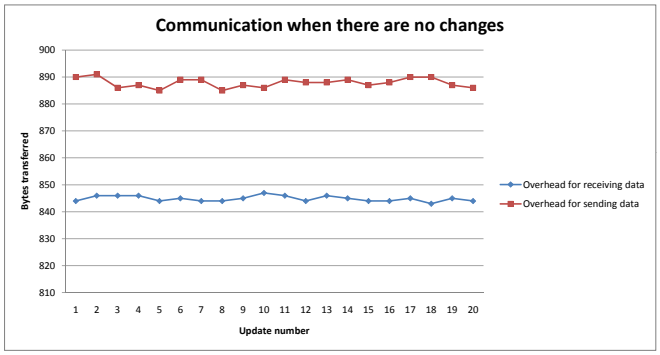

Figure 4.3: Communication overhead

Next, the communication overhead is measured in figure 4.3. Here, the client checks for updates while there are no changes to send back. The figure shows both the outgoing communication for the client to ask if there are changes and the incoming changes when there are no changes. Together, the 844 bytes it receives when there are no changes and the 888 bytes for checking if there are updates, are the overhead.

Scenario	1	2	3	4	5	6	7	8
Data type	1	1	2	2	1	1	2	2
Changes (λ)	5	5	5	5	4	4	4	4
Size (bytes) (D)	36	36	90	3590	36	36	90	3590
Fields (F)	5	5	40	40	5	5	40	40
Communication	1310	1311	1607	20507	1217	1217	1455	16574

Table 4.2: Communication measurements

The results can be seen in table 4.2. The last four scenario's are the same as the first four, except the number of changed data items is four in stead of five. As their might be some overhead per data field and per byte of data, we can adapt the cost function to:

$$CF_{Comm} = x_1 F_c + \lambda(x_2 + x_3 F + x_4 D) \tag{4.1}$$

Table 4.2 has the result of (the average of 20 measurements of) one update per scenario, so F_c can be replaced by 1. If all overhead is renamed to x_i, the formula becomes:

$$\begin{aligned} CF_{Comm} &= x_1 + \lambda(x_2 + x_3 F + x_4 D) \\ &= x_1 + \lambda x_2 + \lambda F x_3 + \lambda D x_4 \\ &= a x_1 + b x_2 + c x_3 + d x_4 \end{aligned} \tag{4.2}$$

Now we have a linear equation that we can solve using the least squares fitting method. Note that x_1 should be between 844 and 845: the measured amount of incoming communication when there were no changed data items. This results in the following linear problem (using non rounded values at the right hand side):

$$\begin{array}{rcrcrcrcr}
x_1 & + & 5x_2 & + & 25x_3 & + & 180x_4 & = & 1310,44 \\
x_1 & + & 5x_2 & + & 25x_3 & + & 180x_4 & = & 1310,69 \\
x_1 & + & 5x_2 & + & 200x_3 & + & 450x_4 & = & 1607,38 \\
x_1 & + & 5x_2 & + & 200x_3 & + & 17950x_4 & = & 20506,94 \\
x_1 & + & 4x_2 & + & 20x_3 & + & 144x_4 & = & 1217,00 \\
x_1 & + & 4x_2 & + & 20x_3 & + & 144x_4 & = & 1217,25 \\
x_1 & + & 4x_2 & + & 160x_3 & + & 360x_4 & = & 1454,75 \\
x_1 & + & 4x_2 & + & 160x_3 & + & 14360x_4 & = & 16574,25
\end{array} \tag{4.3}$$

With the constraints:

$$\begin{array}{rcrcr}
844 & \leq & x_1 & \leq & 845 \\
0 & \leq & x_2 & & \\
0 & \leq & x_3 & & \\
0 & \leq & x_4 & &
\end{array} \tag{4.4}$$

Least squares fitting method:

$$\sum_{i=1}^{8}(f_i(x_1,x_2,x_3,x_4)-v_i)^2 \tag{4.5}$$

Where f_i is $ax_1+bx_2+cx_3+dx_4$ from equation 4.3 and v_i is the corresponding right hand side value. The results (in bytes, rounded):

$$\begin{array}{rcl}
x_1 & = & 844 \\
x_2 & = & 54,3 \\
x_3 & = & 0,03 \\
x_4 & = & 1,08
\end{array} \tag{4.6}$$

In stead of looking at the error in bytes, one can look at the error as a percentage of the corresponding value. The function we need to minimize is:

$$\sum_{i=1}^{8}(\frac{100(f_i(x_1,x_2,x_3,x_4)-v_i)}{v_i})^2 \tag{4.7}$$

The results are the same when rounded:

$$\begin{array}{rcl}
x_1 & = & 844 \\
x_2 & = & 54,3 \\
x_3 & = & 0,03 \\
x_4 & = & 1,08
\end{array} \tag{4.8}$$

What does this mean? The general overhead (x_1) is 844 bytes, every time the client checks for changes this is the minimum he receives. For each data item there is an additional 54,3 bytes overhead (x_2), which is the result of serializing the DistributionObjects to XML. The value of x_3 is so low we can ignore it. Finally, x_4 is the amount of data transferred for each byte of data: 1,08 bytes, which results in 8% overhead per byte of data. That 8% is most likely the overhead for using xml to transfer integer numbers over a webservice, as the number 123456 requires more characters to transfer it than the number 123. Strings with a length of 50 simply require 50 bytes. To verify this, let's look at the percentage of numbers in the data. For the 4 scenarios, there is in total 252 bytes of number data and 3752 bytes of total data, which leads to 6,7% numbers in the data. If the above assumption is correct, that would mean that each byte of number data requires over 2 bytes to transfer. For a 32 bit integer number consisting of 4 bytes, the maximum value is 4294967295, in other words: 10 characters (bytes) to transfer.

Note that network latency has no influence on the amount of communication that takes place. The measurements were the same using the wireless internet connection on the client, as using a dongle for a direct connection with the server.

4.1.2 Client processing time

Setup

The hardware used in these measurements is the same as for the Communication in section 4.1.1. Furthemore, the same four scenarios were used to measure the time it takes a client to process a change:

- A small data object with a minimal amount of data in it.
- A small data object with a maximal amount of data in it.
- A large data object with a minimal amount of data in it.
- A large data object with a maximal amount of data in it.

The measurements were taken at the client side, the time the data is received is recorded as well as the time the data was processed. The results are the difference between these two times.

Results

First I measured the time it took when the client received new data it had to add to its local storage (figure 4.4). The client received five new data items each time it checked for updates. Several observations can me made from these results.

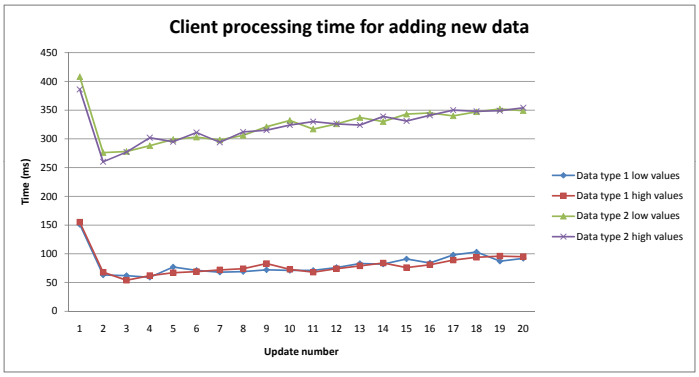

Figure 4.4: Client processing time for adding new data

First of all, it doesn't really matter if a data object has the maximum or minimum size: in both cases the times are in the same order of magnitude. However, it does matter how many data fields it contains: a large data object with many data fields takes significantly more time to process than a small data object with only a few data fields.

Furthermore, the graph shows a climbing trend when the update number rises. The data is saved in binary files, sorted by their Id field. A new data item usually has the highest Id and can therefore be stored at the end of the file. As the position is searched using a binary search, it will not take long to find the correct position. However, every time one or more data items are saved, a check is done to ensure the data file is still consistent (free of errors). The larger the file, the more time this check will take. This explains the climbing trend. Note that the cause is in the used DAO layer, not in any part of the distribution layer. One could simply remove the extra consistency check, but ViaData prefers it because of the sensitivity of the data. Personally, I would recommend to skip this integrity check or switch to a format that doesn't need these checks.

The last thing we can observe from this figure is the initial spike in the first update. This is probably due to the initialization of the data file and some internal objects that are created when using a certain data type for the first time.

The second measuring was the time it took for the client to process changes in data that already existed in its local storage (figure 4.5). The local storage con-

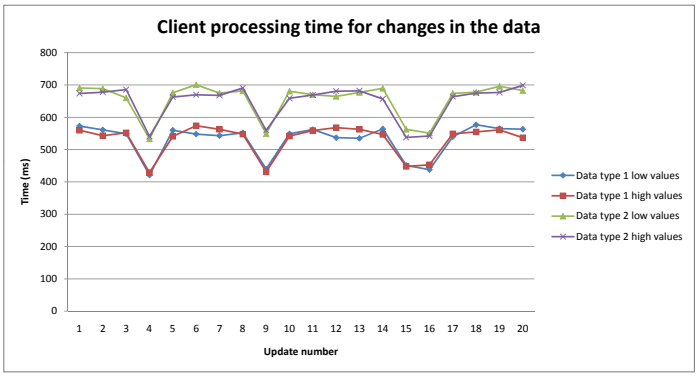

Figure 4.5: Client processing time for changes in the data

sisted of 50 data items. Each time the client checked for updates, he received five random changes (the same as used in figure 4.1). The first question that arises, is why it took the client so much longer to process these changes than when adding new data? As the data is spread randomly across the data storage, the client needs to perform five binary searches (one for each change) to locate the exact position in the file instead of searching for the end and then appending the data. Again we see that the size of an individual data item is not very important, but the number of data fields it contains. Note that there is no climbing trend as the number of items stored didn't change.

The time it took a client to process the changes when there were no changes at all is shown in figure 4.6. As expected, it doesn't matter what data type it was asking for, each time the client came to the conclusion that it didn't receive any changes in about 17,4 ms.

As a final note I want to point out that these measurements are very dependent on the type of data storage used. In these measurements the data storage consists of a set of binary files. When using a set of XML files or a SQL database for a mobile device, the results will probably be different. Because Connect-It currently uses binary files as a data storage on a PDA, only those measurements were done.

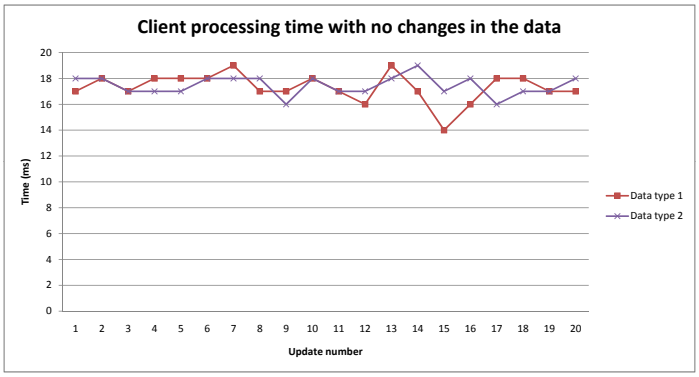

Figure 4.6: Client processing time when there are no changes in the data

4.1.3 Consistency

There were no consistency measurements done, as all parameters are controlled in this simulated environment. The ratio between F_c and γ is known and there are no unknown variables to be measured.

4.2 Evaluation

In chapter 3 an architecture was described to distribute data from a server to several mobile clients. First the requirements from the problem statement (section 1.4) are reviewed in section 4.2.1. After that, section 4.2.2 will review the challenges of section 2.2 to see how the new architecture holds up.

4.2.1 Requirements

Below are the requirements from section 1.4 and for each requirement a discussion why it is met.

1. **Design a new model of the Connect-it system.**

   No model previously existed, the new model is shown in section 1.3.

2. **Change the architecture in such a way that all communication is being done in separate components.**

   The Distribution_PDA_DAO_Impl.dll, Distribution_Server_DAO_Impl.dll and Distribution_Shared_DAO_Impl.dll (see section 3.1) are responsible for all

communication that takes place. The PDA web service (pda.asmx) is a connector between these two components.

3. **Change the architecture in such a way that changes in the application logic don't require changes in the new communication components.**

   Because the communication components are independent on any application specific components (see the dependencies in figure 3.7), all changes in the application specific parts have no influence on these components.

4. **Change the architecture in such a way that changes in the type of data storage used or changes in what data is stored don't require changes in the new communication components.**

   The communication components make use of the IDAO interface (see figure 3.1), no hardcoded dependencies exist between the communication components and a specific data storage. Furthermore, the component DataObjects.dll has been moved to the application specific components and all lower layer components no longer have a dependency to that component. The result is that there is no knowledge about what the data represents and the middleware components can be reused in other applications as well.

5. **Minimize the bandwidth usage for synchronizing the clients with the server:** $(bu(Commit) + bu(Refresh))$ **where** bu **is a function that determines the bandwidth usage of a certain operation.**

   When looking at the communication cost comparison (section 2.5.1), the new architecture has half the cost of the Roam replication system [17] as the new architecture uses a client-server topology. Note that this figure does not include the fact that Roam (as most other replication methods) replicate database tables and not data that passes a filter (see the DataFilter in section 3.2). This means that the new architecture is even more efficient with respect to communication.

   Compared to the old architecture there is a major reduction in communication that takes place (up to 98,5%, see section 2.4.2). This is possible because: (i) only the changes are transferred when checking for updates and (ii) the DistributionObject serializes only the data and not the meta data (see section 4.1.1). Note that a lower cost would be possible if no web service is used, but a direct connection. Currently this is not an option as the use of a web service is required from a commercial point of view (see section 1.2).

6. **Impact on battery usage should be minimal: minimize the client processing time for synchronizing the clients with the server: $(pt(Change) + pt(Commit) + pt(Refresh))$, where pt is a function that determines the processing time of a certain operation.**

   As with the communication cost, the use of a client-server topology halves the cost compared to Roam. Compared to the old architecture we also see a major cost reduction (up to 98,5%, see section 2.5.2) because only the changes are transferred.

7. **The number of clients can change over time.**

   In the old architecture this requirement was met and there were no changes in this respect.

4.2.2 Challenges

Below are the challenges from section 2.2 and for each challenge a discussion why the new architecture has overcome that challenge.

1. **Heterogeneity**

   The new architecture now uses middleware for abstracting the distribution of the data. When Connect-It needs to use a different method to communicate between the clients and the server, for example a direct connection instead of a web service, there are no changes required at application level.

2. **Openness**

   Because all interfaces to the communication components and the data storages are implemented using generic types, these components no longer need to be changed when the database itself changes. Adding new data types or changing existing ones has no longer any influence on these generic components.

3. **Security**

   Connect-It had already overcome the security challenge, but the new architecture is even more secure. When transferring a data item, that data item was directly serialized into a XML message which is readable for the human eye because all meta data such as data field names is included in it. The new architecture uses a DistributionObject that contains one byte array with all the data of a data item. There is no meta data and it is harder to decipher the meaning of those numbers being transferred.

4. **Scalability**

   Connect-It was already very scalable and nothing has changed in this respect.

5. **Failure handling**

   When a client wanted to send data to the server, that data was kept in a buffer in memory alone. A crash resulted in those changes never being transmitted. The new architecture persistently stores every item in the buffer the moment it is added to the buffer, so a crash no longer has the result that data is lost.

6. **Concurrency**

   At the server side there is conflict detection that alerts a user (for example, the administrator) or a user group (for example, the administrator group) that there is a conflict. The user to resolve the conflict can choose to keep one of both versions or merge them manually. Furthermore, the client checks for changes in the data much more often which results in much less conflicts (see section 2.5.3).

7. **Transparency**

   Connect-It already had a good amount of transparency and from a user point of view, nothing has changed in this respect.

4.3 Summary

In this chapter the design from chapter 3 has been compared to the requirements from chapter 1 and analysis from chapter 2. In all aspects the new architecture is a big improvement over the old architecture. In the next chapter a final conclusion is given and some thoughts about what can be done in the future.

Chapter 5

Conclusion

In this chapter a final conclusion is given in section 5.1. Section 5.2 gives some thoughts about what can be done in the future.

5.1 Conclusion

As predicted in section 2.6, it was possible to design a new architecture that fulfilled all requirements from section 1.4 while overcoming all challenges from section 2.2. The new architecture reduces both the amount of communication needed and processing time at the client by up to 98,5% (see section 4.2). Furthermore, the work for the programmers has been simplified because there is no longer a direct dependency between application specific components and the middleware: changes in the application have no influence on the distribution or storage of the data and vise versa. The result is a more heterogeneous, open, secure and concurrent architecture.

5.2 Future Work

When I started to work on this thesis, Connect-It was still using the Microsoft .NET 1.1 framework. The roadmap predicted that by the time this thesis was finished, Connect-It would be using the .NET 2.0 framework. Therefore, the implementation of the architecture was done using the .NET 2.0 framework. However, changes in the roadmap have postponed the migration of Connect-It to .NET 2.0, therefore no real world testing and measurement could be done. As a result, the conclusions in this thesis are based on estimates. In the future, real world data can be used to evaluate the architecture.

Something that has been left out of the architecture is the automatic adjust-

ment to the measurements of the environmental parameters of the cost functions (see section 2.4.1). In [19] a method is introduced to dynamically adapt some parameters to a changing environment. Given the new architecture it is very easy to implement this mechanism to keep the cost functions at a minimum. For now, ViaData didn't see the use of this mechanism but this could change in the future if the performance results in a real world environment are known.

Another aspect that was out of the scope of this thesis was automatic conflict resolution. A lot of research has already been done in this field and there are a lot of good solutions. Which solution is the best is very dependent on the details of the application. In the current architecture, conflicts can begin to queue up if they are not resolved quickly, which reduces the scalability of the system. It might be interesting to look into an algorithm that can deal with the conflicts within Connect-It.

When looking at the cost function in section 2.4.2 it might be interesting to think about the relation between γ and S_d. Using larger units of data, the expected number of unique changes might become smaller which could result in less communication needed. In the case of Connect-It, the size of the data items is fixed, but the data items might be bundled into larger data units. The opposite can also be possible: using smaller units of data, the expected number of unique changes might increase, but because only very small data units are used, for example only one database field in stead of a complete record, the overall communication needed might be lowered even more.

Bibliography

[1] Concurrent versions system. *http://www.nongnu.org/cvs/*. [cited at p. 14]

[2] Data access object (dao) design pattern. *http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html*. [cited at p. 3, 32, 68]

[3] Netlimiter. *http://www.netlimiter.com*. [cited at p. 78]

[4] Simple object access protocol (soap). *http://www.w3.org/TR/soap/*. [cited at p. 3]

[5] Subversion. *http://subversion.tigris.org/*. [cited at p. 14]

[6] Web services. *http://www.w3.org/2002/ws/*. [cited at p. 3]

[7] Jun Cai, Kian Tan, and Lee. Energy-efficient selective cache invalidation. *Wireless Networks*, 5:489–502, 1999. [cited at p. 12]

[8] Guohong Cao. On improving the performance of cache invalidation in mobile environments. *Mobile Networks and Applications*, 7:291–303, 2002. [cited at p. 12, 65]

[9] Boris Y. Chan, Antonio Si, and Hong V. Leong. A framework for cache management for mobile databases: Design and evaluation. *Distributed and Parallel Databases*, 10:23–57, 2001. [cited at p. 11]

[10] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems, concepts and design*. Addison-Wesley, fourth edition, 2005. [cited at p. 14, 15, 16, 18, 19]

[11] Todd Ekenstam, Charles Matheny, Peter Reiher, and Gerald J. Popek. The bengal database replication system. *Distributed and Parallel Databases*, 9:187–210, 2001. [cited at p. 13, 14]

[12] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Disconnection modes for mobile databases. *Wireless Networks*, 8:391–402, 2002. [cited at p. 14]

[13] Qinglong Hu and Dik Lun Lee. Cache algorithms based on adaptive invalidation reports for mobile environments. *Cluster Computing*, 1:39–50, 1998. [cited at p. 12]

[14] Phillipe Kruchten. Architectural blueprints – the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, november 1995. [cited at p. 21, 49]

[15] George Liu and Jr Maguire. A mobility-aware dynamic database caching scheme for wireless mobile computing and communications. *Distributed and Parallel Databases*, 4:271–288, 1996. [cited at p. 12]

[16] Esther Pacitti, Pascale Minet, and Eric Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9:237–267, 2001. [cited at p. 14]

[17] David Ratner, Peter Reiher, and Gerald J. Popek. Roam : A scalable replication system for mobility. *Mobile Networks and Applications*, 9:537–544, 2004. [cited at p. 13, 86]

[18] G. Russello, M. Chaudron, and M. van Steen. Customizable data distribution for shared data spaces. *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, 26 June 2003. [cited at p. 14]

[19] G. Russello, M. Chaudron, and M. van Steen. Dynamic adaptation of data distribution policies in a shared data space system. *Proc. Int'l Symp. On Distributed Objects and Applications (DOA)*, 25 October 2004. [cited at p. 14, 36, 90]

[20] A.S. Tanenbaum and M. van Steen. *Distributed systems, principles and paradigms.* Pearson Prentice Hall, second edition, 2007. [cited at p. 15, 17, 20]

[21] Melissa Tjiong and Johan Lukkien. On the consistency of soft-state based service registration. *Proceedings of GLOBECOM Workshops*, 2008. [cited at p. 41]

[22] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. *Computer communication review : a quarterly publication of the Special Interest Group on Data Communication*, 29:163–174, 1999. [cited at p. 12, 65]

List of Figures

List of Tables