Transaction integrity in the ING financial services architecture

van Geenen, J.L.

Award date:
2004

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

Transaction integrity in the ING Financial Services Architecture

by

J.L. van Geenen

Supervisors: Prof. dr. ir. J.F. Groote
J. Miedema MIM
ir. A.J. Mooij

Eindhoven, August 2004

Preface

In this thesis we study how to guarantee distributed transaction integrity in the ING Financial Services Architecture (IFSA). IFSA is a service based information-technology architecture recently introduced at ING, a multinational financial institution, headquartered in Amsterdam, the Netherlands.

Chapter 1 introduces the business context and gradually defines the exact problem statement. We also discuss the possible solutions to the problem statement and make a founded choice for a particular solution to be worked out in the subsequent chapters.

In chapter 2 we present a communication protocol as a solution to the problem statement given in chapter 1. We also provide directions for implementation and operation of the protocol.

The protocol is formally verified in two steps: first we verify a simplified version of the protocol in chapter 3. In chapter 4 we generalize the results from chapter 3 such that they apply to protocol as presented in chapter 2.

The main findings in this thesis are presented in chapter 5. Section 5.1 gives a management summary, section 5.2 gives a technical summary and section 5.3 provides directions for further research and development concerning distributed transaction integrity in IFSA.

Appendices A and B have been included only for reference: they are not meant to be read from beginning to end. We link to (sections) of the appendices in the sequel.

Table 1 indicates the intended readership of all sections of this thesis. A '+' indicates 'suitable', a '-' indicates 'unsuitable'.

Chapter	*Sections*	*Managers*	*IT-professionals*	*Computing Scientists*
1	up to 1.1.2	+	+	+
	1.1.2 up to 1.3.2.2	-	+	+
	1.3.2.2	-	-	+
	1.3.3	-	+	+
2	all	-	+	+
3	all	-	-	+
4	all	-	-	+
5	5.1	+	+	+
	5.2	-	+	+
	5.3	+	+	+
A	all	-	+	+
B	all	-	-	+

Table 1: Intended readership

We advise readers to read all the sections suitable for them in the order in which they appear in this thesis.

Acknowledgements

People at ING
I would like to thank the following people (formerly) at ING for helping me find and define a graduation project and for the opportunities offered: Ivo van Geenen, Kees Buis, Evert Himmelreich, Ruud Goudriaan, Raymond Bolten, Dirk Brouwer, Marco Doeland, Rob Moes, Jan Miedema and Dorien Jongeneel.

I would like to thank Rob Moes, Jan Miedema and all people at the former OPS&IT/ NSI/ ECS/ ALS department at ING for helping me to find my way at ING. I would like to thank Jan Miedema, Rudy Wouters and Rob Moes for the valuable input they provided, and for their time and energy spent on this project.

People at the Technische Universiteit Eindhoven
I want to thank Jan-Friso Groote for all his time and energy spent on my graduation project and for bringing me into contact with the right people at the Technische Universiteit Eindhoven, the Centrum voor Wiskunde en Informatica in Amsterdam, and the Laboratory for Quality Software in Eindhoven.

I would like to thank Arjan Mooij for the meticulous precision with which he reviewed my work, and for his countless hints, contributions and support. I realize that my project must have cost you an enormous amount of time: thank you for every minute. I would also like to thank Jaco van de Pol for his time and energy during the early stages of my project, and Cor Hurkens from the Combinatorial Optimization group at the Technische Universiteit Eindhoven for helping me prove a precursor to theorem 1.

People at home
Last but not least I would like to thank my girlfriend, Kristel Maas, and my parents, Josette van Geenen-Hustings and Ivo van Geenen, for continuously supporting me during my studies at the Technische Universiteit Eindhoven.

Contents

Preface **3**

Acknowledgements **5**

1 Problem Description **11**

1.1 Context . . . 11

1.1.1 The ING Financial Services Architecture . . . 11

1.1.2 Communication patterns offered by the bus . . . 12

1.2 Problem statement . . . 15

1.3 Possible solutions . . . 17

1.3.1 Resending requests and replies . . . 18

1.3.2 Compensating transactions . . . 18

1.3.3 Allowing additional F&F-messages . . . 25

2 Description of the n-R/R protocol **27**

2.1 Introduction . . . 27

2.2 The client program . . . 27

2.2.1 High-level description . . . 27

2.2.2 Pseudo-code description . . . 29

2.3 The service program . . . 32

2.3.1 High-level description . . . 32

2.3.2 Pseudo-code description . . . 33

2.4 Example transactions . . . 35

2.4.1 Scenario: a distributed transaction with size 2 . . . 35

2.4.2 Scenario: three distributed transaction with size 1 . . . 36

2.5 Considerations . . . 37

2.5.1 The transaction-identifier . . . 37

2.5.2 Deadlock and suitable values for `MAX_SZ` and `MAX_TIMEOUT` . . . 37

2.5.3 Efficiency . . . 39

2.5.4 Handling Exceptions . . . 40

2.5.5 Limitations . . . 41

3 Verification of the 1-R/R protocol **43**
3.1 Overview 43
3.2 Identifiers and Types 44
3.3 Formal model of the IFSA application-bus 47
3.3.1 Variables 47
3.3.2 Language constructs 49
3.3.3 R/R communication 49
3.3.4 F&F communication 51
3.4 Specification 54
3.4.1 Introduction 54
3.4.2 Clients 56
3.4.3 Services 57
3.4.4 Main program 58
3.4.5 Atomicity 59
3.5 Partial correctness criteria 60
3.5.1 Formalization of requirement 1 61
3.5.2 Other invariants and auxiliary functions 61
3.6 Assertions: local correctness 64
3.6.1 Assertions in the main program 64
3.6.2 Assertions in the client component 64
3.6.3 Assertions in the service component 66
3.7 Assertions: global correctness 69
3.7.1 Introduction 69
3.7.2 Absence of disturbance between clients 70
3.7.3 Absence of disturbance between clients and services 70
3.7.4 Absence of disturbance between services 74
3.8 Invariants: correctness 76
3.9 Progress (requirement 2) 80
3.9.1 Additional assertions in the client component 80
3.9.2 Additional assertions in the service component 80
3.9.3 Absence of Live- or deadlock 81
3.9.4 Conclusion 83

4 Verification of the n-R/R protocol **85**
4.1 Specification 85
4.1.1 Additional auxiliary functions 86
4.1.2 Additional global variables 86
4.1.3 Structure of the client component 87
4.1.4 Clients 88
4.1.5 Atomicity 89
4.2 Partial Correctness criteria 89

4.2.1 Formalization of requirement 1 89
4.2.2 Additional system-invariants and auxiliary functions 90
4.3 Assertions: local correctness 90
4.3.1 Assertions in the client component 91
4.4 Assertions: global correctness 97
4.4.1 Introduction 97
4.4.2 Absence of disturbance between clients 97
4.4.3 Absence of disturbance by clients of assertions of services 98
4.4.4 Absence of disturbance by services of assertions of clients 99
4.5 Invariants: correctness 102
4.5.1 Introduction 102
4.5.2 Proofs 102
4.6 Progress (requirement 2) 107
4.6.1 Additional assertions in the client component 107
4.6.2 Additional assertions in the service component 108
4.6.3 Absence of Live- or deadlock 108
4.6.4 Conclusion 110

5 Main Findings **111**
5.1 Management Summary 111
5.2 Technical Summary 113
5.2.1 Chapter 1 113
5.2.2 Chapter 2 115
5.2.3 Chapter 3 115
5.2.4 Chapter 4 116
5.2.5 Conclusion 116
5.3 Directions for further research and development 116

A The business application interface **119**
A.1 Introduction 119
A.2 Common BAI concepts 119
A.2.1 Connections and Service slots 120
A.2.2 (Un)registering a service 120
A.3 Sending messages or requests 120
A.3.1 The 'Fire and Forget' pattern 120
A.3.2 The 'Request/Reply' pattern 121
A.3.3 Creating requests 122
A.3.4 Creating replies 122
A.3.5 Sending requests and replies 122
A.4 Receiving messages 123
A.4.1 The `IFSAGetMessage` method 123
A.4.2 The `IFSAGetReply` method 123

A.4.3 The IFSAGetAnyReply method 124
A.5 Units of work 125
A.6 Considerations 126

B Functions, assertions, invariants and main-program 129

Chapter 1

Problem Description

In this chapter we introduce the business context, we give a definition of the problem addressed in this thesis and briefly sketch the different possible solutions. Along with each possible solution we discuss its pros and cons, on which we base our choice for a particular solution to be worked out in the subsequent chapters.

Section 1.1 introduces the context of the problem setting and some concepts and definitions used throughout this thesis. Section 1.2 gives a problem statement and section 1.3 treats the different possible solutions.

1.1 Context

ING is a large multinational financial institution headquartered in Amsterdam, the Netherlands. It is active in over 65 countries, has over 110,000 employees and serves over 50,000,000 customers worldwide. Profits in 2002 amounted to 4.3 billion euros.

Historically, ING has been formed as a conglomerate of multiple formerly independent companies active in the financial services industry. All these former companies originally had their own business processes and information-technology (IT). Leveraging opportunities for economies of scale required a new company-wide business-architecture and a corresponding IT-architecture: the ING Financial Services Architecture.

1.1.1 The ING Financial Services Architecture

The pan-European ING Financial Services Architecture (IFSA) is a reflection of ING's business-architecture. IFSA defines organizational building blocks called *domains*. Two examples of such domain are;

- Party Information Management: this domain manages data of customers and relations of ING.
- Payments: this domain manages money-transfers.

Three fundamental design-principles guided the design of IFSA;

- All intra-domain communication between applications is implemented using services (similar to web-services, see [Kay03]) via the central IFSA *application bus*, referred to as *bus* in the sequel.
- Services are logically distributed over the domains of IFSA.

- Services are *loosely coupled* [Kay03]: they can connect to, or disconnect from the network as in, for example, the internet.

The following three reasons led to the definition of the design-principles of IFSA;

- Enforcement of reuse, and concentration of services in domains avoids unnecessary duplication.
- A service based architecture facilitates legacy-dismantling and -unlocking. Computing platforms have been introduced in the company since the 50's. A lot of those legacy systems operate with such efficiency that it is not economical to replace them right away. Special 'adaptor' software makes it possible to hook legacy systems to the bus, thus unlocking their functionality to other IFSA-services. Hence clients depend only on services' interfaces but not on the underlying applications. This makes it possible to gradually replace and dismantle legacy systems when economical, with minimal disruption. Also, because of the historical shaping of ING, there are often multiple legacy-applications with similar functionality. Operation and maintenance required for these applications is expensive whence dismantling those offers great potential for cost savings.
- Loose coupling of services is dictated by the business architecture. Indeed, one of its concepts is that it must be easy for business processes to be defined, changed or terminated.

The main functionality offered by the bus is platform-independent messaging and acting as a service-repository. Basically, a program can connect to the bus and act as a service-requester (client) and/or register a service and act as a service-provider (service). The situation is quite similar to web-services where we also have (loosely coupled) clients and services. In this thesis we only consider programs that act either as client or as service, but not both.

Along with IFSA comes an organizational apparatus that standardizes the way in which clients and services are defined and (inter-)operate. This organizational apparatus governs documentation, security, design-guidelines for clients and services and further development of the bus. The conversion to IFSA was started several years ago and is still a process in motion, even if the official IFSA project group is dismantled in August 2004. Many services have already been defined and implemented. The bus went through different versions, version 2.0 being the latest at the time of writing. This is also the version this thesis is based on, the IFSA-documentation used being [Fit03] and [it03].

1.1.2 Communication patterns offered by the bus

The bus offers two communication patterns: Request/Reply (R/R) and Fire and Forget (F&F). The R/R pattern consists of two messages: typically, a client sends a request to a service upon which the service ideally responds with a reply. The R/R-pattern is similar to the part of the HTTP-protocol most often used in web-browsing: sending a request and receiving of the response. The R/R-pattern is said to be *lossy* because requests and replies can be lost. The R/R-pattern is discussed in section 1.1.2.1.

The F&F-pattern consists of a single F&F-message, referred to as *message* if no confusion is possible. F&F-messaging is one-way, unordered, lossless and asynchronous; messages are guaranteed to arrive, the order in which they arrive may differ from the order in which they were sent. The F&F-pattern is discussed in section 1.1.2.2.

1.1.2.1 The R/R Pattern

We discuss the R/R-pattern in more detail here. The four possible scenarios that can occur when a client sends a request to a service and expects a reply are shown in figure 1.1.

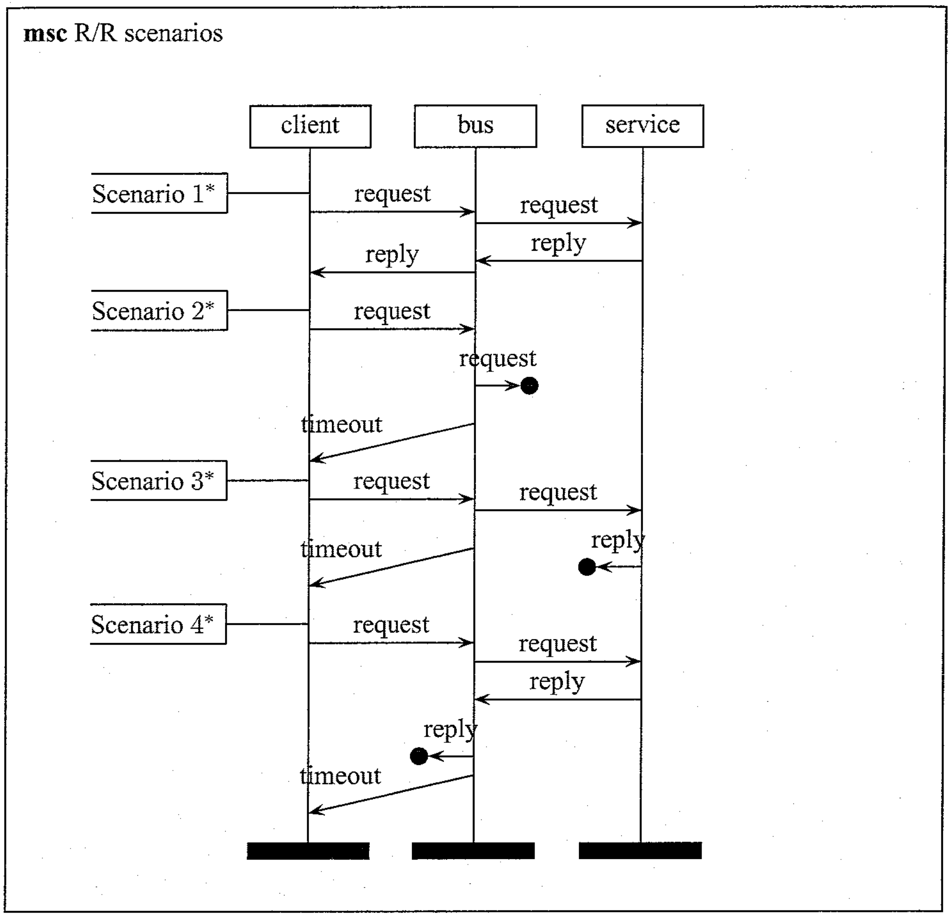

Figure 1.1: Possible scenarios for R/R

We explain the different scenarios shown in figure 1.1 below;

1. Only in scenario 1* we have that no messages are lost: the bus forwards the request sent by the client to the service, the service sends a reply which is forwarded to the client by the bus.

2. In scenario 2*, the service was not ready in time to receive the request sent by the client which the bus tried to forward: the client receives a timeout instead of a reply from the bus. Note that in this scenario, the service cannot detect that the client actually sent it a request.

3. In scenario 3*, the service did receive the request in time but too much time passed before the service sent the reply. Hence the reply is lost and the client receives a timeout instead.

4. In scenario 4*, the service did receive the request and sent a reply in time but too much time passed before the client was ready to receive the reply from the bus. The client receives a timeout instead. In scenarios 2*, 3* and 4*, the client receives

a timeout if a request or reply is lost, although it cannot discriminate between the scenarios. The service cannot discriminate between scenarios 1*, 3* and 4*. These observations allow us both to reduce the number of scenarios and also, to leave out the passing of messages by the bus: figure 1.2 reflects this. Instead of letting the bus pass messages between clients and services, we abstract from the bus and do as if clients and services communicate directly. Also note that a timeout is not an actual message in reality: it is merely an indication for the client that a reply could not be received. We do model them as such for easier representation.

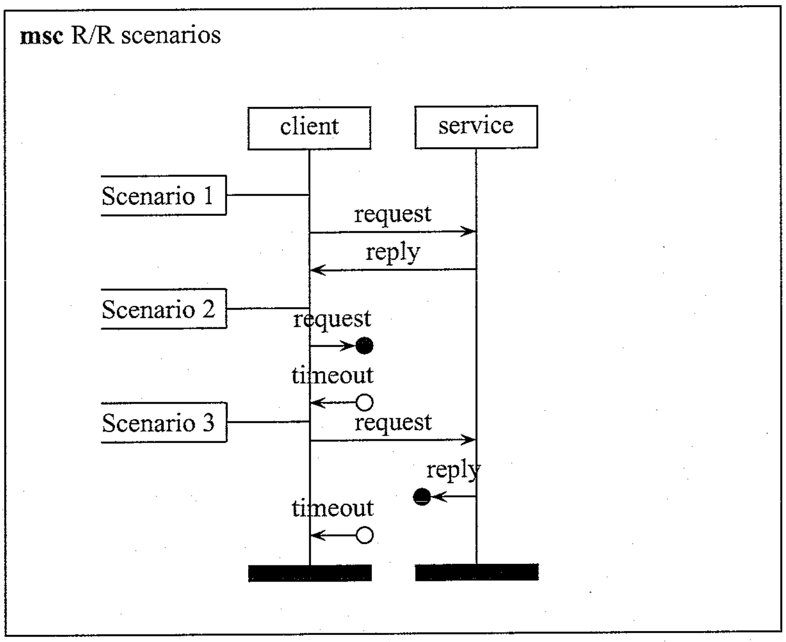

Figure 1.2: Possible scenarios for R/R (reduced)

The R/R-pattern features short-lived *connections*: a request is related to a unique reply or timeout. The connection lasts for at most two messages.

1.1.2.2 The F&F-Pattern

We discuss the F&F-pattern here. Basically, the bus stores a F&F-message upon sending (usually by a client) and deletes it only after it has been successfully received (usually by a service). This phenomenon is called *persistent messaging*. The only scenario possible when a client sends a F&F-message to a service is given in figure 1.3. It was already noted that F&F-messaging is unordered: this is demonstrated in figure 1.4. The client sends two messages to the service. It may happen that the message sent last, is actually received by the service before the first message sent. The F&F-pattern is connectionless: every two F&F-messages are by themselves unrelated.

A more detailed description of the bus and its business application-interface (BAI) can be found in Appendix A: the BAI is the interface that programmers must use when writing clients or services.

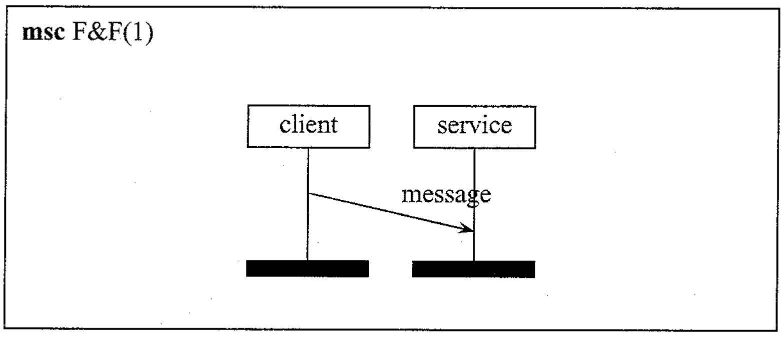

Figure 1.3: Possible scenarios for F&F

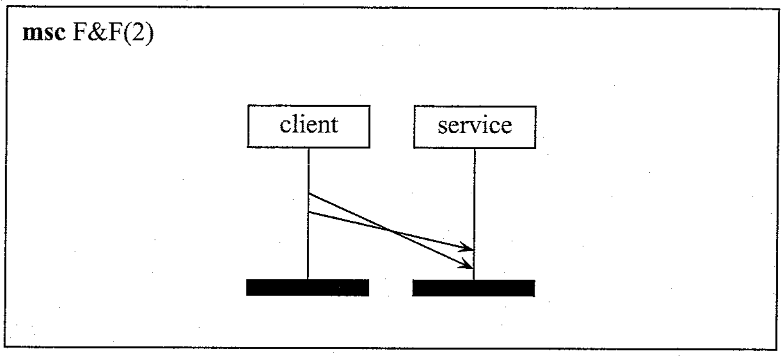

Figure 1.4: Unordered nature of F&F-messaging

1.2 Problem statement

In this section we gradually define the problem-statement. To this end we must introduce some concepts and definitions used throughout this thesis.

We refer to the ensemble of clients and services within IFSA as the *transaction system*. The nature of the services offered by the services can differ enormously. In order to concretize the use of a service by a client we model it as a *distributed database transaction*. To this end we assume that each client and each service has its own *local database*, referred to as *database* if no confusion is possible. A database can only be read or updated by its owner by means of a *local database transaction*, referred to as *transaction* if no confusion is possible.

A distributed transaction between a client and a single service typically involves two local transactions: one executed by the client, one by the service. The conceptual (non-existent) *distributed database* is simply the multiset of all databases of all machines in the transaction system. In this way we can easily describe global correctness criteria and distributed transactions.

To ensure consistency of the distributed database, it is generally required that the transaction system maintains the following standard ACID properties of distributed transactions:

Atomicity. Either all operations of a distributed transaction are reflected properly in the distributed database, or none are.

Consistency. Execution of a distributed transaction in isolation preserves the consistency of the distributed database.

Isolation. Even though multiple distributed transactions may execute concurrently, it is guaranteed that for every pair of distributed transactions $\mathcal{T}_i, \mathcal{T}_j$, it appears to $\mathcal{T}_i$ that $\mathcal{T}_j$ finished execution before $\mathcal{T}_i$ started or it appears to $\mathcal{T}_j$ that $\mathcal{T}_i$ finished execution before $\mathcal{T}_j$ started.

Durability. After a distributed transaction completes successfully, the changes it has made to the distributed database persist, even if there are system failures.

All definitions above were taken from [SKS98], which gives a more elaborate treatment of the subject. Throughout this thesis we assume that durability is guaranteed by the implementors of clients and services: we do not concern ourselves with it.

A fundamental problem in ensuring atomicity, consistency and isolation of distributed transactions is ensuring that either each party involved *commits* its local transaction, or, each party *aborts* its local transaction. We refer to this problem as the *commitment problem*. The commitment problem is especially present when performing *distributed update-transactions* while using only R/R-messaging for communication. In such distributed transactions, at least one client or service must update its database.

To illustrate this, consider again the scenarios given in figure 1.2. Now assume the request to contain an instruction for the service to update its database by means of a local database transaction. Also assume the reply to contain information with which the client must update its database, also, by means of a local database transaction.

Obviously if the client receives a timeout, it must abort its local database transaction (scenarios 2 and 3). In scenario 2 this is no problem: the service never received the client's service-request and hence never started a local database transaction of its own. In scenario 3 this is a problem though: the service cannot tell that the reply it sent was not received by the client.

The impossibility for the client to discriminate between scenarios 2 and 3 and for the service to discriminate between scenarios 1 and 3 is in fact the core of our problem. If the client aborts its local database transaction, the service must also abort its local database transaction.

The commitment problem has been analyzed and described as the *coordinated attack problem* in [Lyn96]. It is also shown in [Lyn96] that there exists no deterministic algorithm that can solve the coordinated attack problem if communication between the machines involved is lossy. R/R-messaging is also a form of lossy communication, in view of this one could advise against using R/R-messaging in distributed transactions in which the commitment-problem must be solved.

R/R-messaging though, has some advantages over (reliable) F&F-messaging;

- R/R-messaging has much lower overhead than F&F-messaging: the reliability of persistent messaging with guaranteed delivery comes at a price.
- A single reply or timeout is related to a unique request: R/R-messaging provides a (short-lived) connection between the sender of the request and the addressee. The bus does not provide such functionality for F&F-messages: every two F&F-messages are unrelated. The behavior of R/R-messaging can in part be simulated with F&F-messaging by adding identifiers to the *payload* (contents) of messages to relate them. This was actually one of the solutions formerly considered within ING to dealing with the lossy nature of R/R-messaging. However, protocols featuring R/R-messaging have a danger of deadlock if all R/R-messaging is replaced by F&F-messaging. We explain why this is so;
  - Consider figure 1.2 again. As before, assume the request to contain an instruction for the service to perform an update to its database by means of a local

database transaction. Also assume that the reply contains information with which the client must update its database, also, by means of a local database transaction. Now in addition, assume that we have managed to replace the request and reply by F&F-messages such that a 'F&F-request' and a 'F&F-reply' are related by means of payload-variables (variables present in the contents of messages).

Because F&F-messaging is reliable, we have that only scenario 1 applies. However, because IFSA is a loosely coupled architecture, it may be so that the service is down. Hence it does not sent a 'F&F-reply' to the 'F&F-request' sent by the client, or, it may take hours before it does. If the client simply waits for the 'F&F-reply' to arrive, it is stuck for hours also. Of course we could also introduce a timeout mechanism here, but then we would gain nothing by replacing requests and replies by F&F-messages!

- In section 2.5.2 we give another more subtle example of a protocol that may deadlock clients and services if R/R-messaging is replaced by F&F-messaging. At this time it is not yet appropriate to discuss the details.

In summary, despite its lossy nature, R/R-messaging has important advantages over F&F-messaging, especially in a loosely coupled architecture, but its use in distributed update-transactions is problematic;

Problem Statement: given the communication primitives provided by the IFSA-bus, find a solution for the problems connected to using R/R-messaging when performing distributed update-transactions between one client and any number of services that follows the IFSA design principles. To this end, ensure that;

1. Each distributed transaction always ends such that all parties involved commit their local database transaction, or all parties involved abort their local database transaction;
2. Distributed transactions always terminate;
3. It is impossible that the transaction system enters a state wherein each transaction is always aborted.
4. A good balance exists between price and performance of the protocol.

The reason for the first requirement is clear: it requires the commitment-problem to be solved between the parties involved in the distributed transaction (one client and any number of services). The second requirement ensures that whenever a client or service starts a transaction, it cannot get stuck executing its protocol. The reason for the third requirement is somewhat technical: a transaction system in which every transaction aborts from some point onwards is of little practical use but may still satisfy requirements 1,2 and 4. The reason for the fourth requirement also seems clear: ideally, a solution should be fast (in terms of the number of distributed transactions processed per hour) and cheap (for example, in terms of the number and type of messages required).

1.3 Possible solutions

In this section we sketch some solutions to the problems caused by the unreliability of R/R-messaging when performing distributed database transactions. We ultimately choose a single solution to work out. Note that simply replacing R/R-messaging by F&F-messaging

was ruled out already in section 1.2. Hence we consider the following potential solutions, the first two of which were actually considered within ING;

- Resending requests and replies, discussed in section 1.3.1.
- Compensating transactions [KLS90], discussed in section 1.3.2.
- Allowing additional F&F-messages, discussed in section 1.3.3.

1.3.1 Resending requests and replies

A common solution to make unreliable communication reliable is the use of (a variation of) the alternating-bit protocol (see for example [FvG99]). Basically, such protocols involve re-sending messages until an acknowledgement from the recipient is received. In many cases, the reason for a client receiving a timeout when using R/R-messaging is that either the recipient is too busy serving other clients, or, that the service is temporarily disconnected from the network ('down'). Either way, the continuous re-sending of requests until a reply (instead of a timeout) is received is likely to only lengthen the service's response-time if it is not down, or worse, takes at least as long as the service's downtime. Hence the client's response-time is also lengthened.

Another related problem is the additional administration needed that ensures that services process a transaction associated with the request only once. For it may be the case that a reply to such a request was sent by the service but lost, causing the client to resend its request. Hence we reject this approach.

1.3.2 Compensating transactions

In this section we first discuss compensating transactions in a rather informal manner. The technical details needed to substantiate some of the claims made are given in subsection 1.3.2.2.

1.3.2.1 Discussion

The use of compensating local transactions is an important topic in the world of web-services [Kay03] because of the unreliable, R/R-like, HTTP-protocol often used. It was also one of the solutions considered within ING until we advised against this approach.

The basic idea is as follows: whenever a party in a distributed transaction commits its local transaction while it should have aborted it because another party aborted, the effects of the local transaction are undone by executing a compensating transaction later on.

For an example, reconsider the example distributed transaction discussed in section 1.2 in connection to figure 1.2. Recall that the client aborted in scenarios 2 and 3. Hence the service should execute a compensating transaction after scenario 3 took place. In scenario 2 the service's database needs no compensation because it never started a transaction corresponding to the reply that was lost.

Figure 1.5 is a flow diagram of a typical compensation scenario. The first transaction executed is the transaction that must be compensated for, followed by K *dependent-transactions*. Basically, those are transactions that read - but not update - database variables called *entities* that were updated by the compensated-for transaction. The compensating-transaction must somehow 'undo' the effects of the compensated-for transaction such that the state of the database is consistent, we return to the problem of defining a compensating transaction in the sequel.

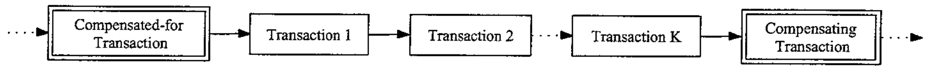

Figure 1.5: Typical compensation scenario

The first objection to compensating transactions as a general solution to the problem statement is that many real-world local transactions cannot be compensated for, because they involve *real actions* [Gra81]: irreversible actions such as dispensing money from an automated teller machine. Hence the use of compensating transactions is limited to local database transactions for which a compensating transaction exists.

For the compensation mechanism to work, we must solve the two following problems;

1. *Is consistency of the distributed database restored after executing the compensating transaction?* A sub-problem of this is: *is consistency of the local database on which the compensating transaction is executed restored after compensation?*

   Both of these problems must be solved by the auditors of the transaction system.

   It is important to understand what is meant by consistency. Even if a compensated-for transaction can be compensated for, it can be the case that the database is inconsistent due to the dependent-transactions. Indeed, the dependent transactions read entities that were changed by the compensated-for transaction. Hence it may be that the dependent transactions have written values that are invalid after compensation. It follows those dependent-transactions may have to be compensated as well. Note that this danger is present for every client or service in the transaction-system that read values witten by a compensated-for transaction. Hence a single compensating transaction may trigger *cascading compensating-transactions* similar to cascading aborts in databases [SKS98].

   In section 1.3.2.2 we prove theorem 1, which states that the problem of determining the consistency of the *local*-database on which the compensating transaction must be executed is intractable.

2. *How can we find a compensating transaction?* Typically, it is the responsibility of architects to design client- or service-programs such that a compensating transaction can be defined and executed. In many applications, the only compensating transaction allowed is the classical 'undo'. For example, if a transaction transfers money from account X to account Y, the compensating transaction would simply transfer the same amount back to account X (and not some other amount).

   In other applications one could define compensating transactions that do more than a classical undo. Such compensating transactions could for example repair inconsistencies that result from the invalid data written (after compensation) by the dependent transactions.

   In section 1.3.2.2 we prove theorem 2. This theorem states that the problem of defining a compensating transaction that undoes the compensated-for transaction *and* restores the integrity of the database is intractable.

It was already noted that in general, both problems given above are intractable. In practice this means that the amount of time needed to solve either problem is generally exponential in the variable K and/or the number of entities of the local database. This is so *unless* all transactions of the transaction-system have specific 'nice' properties. We return to such nice properties later. Also note that without resorting to cascading compensating transactions, it may be the case that even *if* we can quickly decide whether or not compensation can restore the consistency of a database, it may be that consistency can *not* be restored even if none of the transactions involves real actions! Of course this is a problem, hence we reject compensation for transactions for which we cannot prove that compensation is tractable *and* possible.

We give an example of a class of transactions for which compensation is tractable and possible. For this we need the following definition: two transactions T_i, T_j are said to *commute* [KLS90] if execution on a local database of T_i followed by T_j from initial state S yields the same end-state as execution of T_j followed by T_i from initial state S, for *all* reachable states S. In the next subsection we prove that compensation is possible and tractable if all transactions - including compensating transactions - in a transactions-system commute and every transaction has a compensating transaction.

It is evident that many transactions do not commute. For example consider two transactions: one transfers money to account X, the other transfers money from account X. Now suppose that the account-balance is the maximum credit allowed. Obviously we cannot first transfer money from the account in this state: the two transactions do not commute.

Even if we can compensate transactions, one can ask how a client or service can be reliably instructed to compensate a transaction if communication is lossy? It may be so that human intervention is required to achieve this.

In summary, compensation is possible and tractable only for specific transactions. Even then it may have adverse effects on performance (i.e. the number of transactions processed per hour) both because of the danger of cascading compensating transactions and the possible requirement of human intervention. It is unlikely that compensation is a feasible mechanism to solve the problems connected with R/R-messaging and update transactions. Hence we do not investigate this approach any further.

1.3.2.2 Proofs

Most of our definitions regarding compensating transactions in this section are (only slightly) adapted from [KLS90], which gives a more formal treatment of compensating transactions in (local) database transactions. We ourselves prove theorems 1 and 2 which state that finding a compensating transaction is hard, as is determining whether or not the execution of a compensating transaction results in a consistent state of the distributed database. We also prove theorem 3 from which it can be inferred that that compensation is feasible for commutative transactions.

We model distributed database transactions as small multi-programs (i.e. parallel programs). Each component of the multi-program - a *local transaction* or simply *transaction* if no confusion arises- is executed on either a unique client or a unique service and communicates by use of the bus with other components of the same multi-program. We specify components in the Guarded Command Language (GCL) [FvG99].

We use the notation T_i to identify local database transaction i of a distributed transaction. We represent a database as a set of variables and constants: only variables can be modified by transactions. To identify distributed transaction i we use the notation $\mathcal{T}_i$. For example, we could have $\mathcal{T}_0 = T_0 \| T_1$: a distributed transaction $\mathcal{T}_0$ consisting of (local-) transactions T_0 and T_1. We use the convention that the local transaction executed by the client always has the lowest index: T_0 is the local transaction executed by the client.

We use the letters $c, c', c'', \ldots$ to identify clients and $s, s', s'', \ldots$ to identify services. In order to identify which client or service must execute a local transactions we subscript it: ${T_j}_c$ indicates that T_j must be executed by client c.

We identify a compensating transaction simply by prefixing the identifier of the compensated-for transaction by a capital C. For the sake of unity, we summarize and slightly adapt the definitions of histories, soundness of a history and the dependency relation *dep* as defined in [KLS90] below;

Histories: a history H of a set of local transactions ST is the catenation of some permutation of all transactions in ST. We represent a history by means of a list of transactions with list-separator ';', i.e. $T_2; T_0; T_1$ is a history of the set of transactions $\{T_0, T_1, T_2\}$. Histories are used to denote the execution order of local transactions on the database of a client or service. Associated with a history are often an initial state S_0 of the database and an end-state S_1 of the database. We give states as predicates over the database which allows us to capture the relation between begin-state, end-state and history by means of a Hoare-triple $\{S_0\}\ H\ \{S_1\}$.

Transaction-dependency: we say that local transaction T_j *depends* on local transaction T_i if there exists at least one entity e in the database (i.e. e is a 'database-variable') such that;

- T_j reads e after T_i has updated e;
- T_i does not abort before T_j reads e; and,
- every transaction (if any) that updates e between the time T_i updates e and T_j reads e, is aborted before T_j reads e.

The set of all dependent transactions of T is given by $dep(T)$.

Soundness: Let H be the history of $dep(T)$ where T is the compensated-for transaction, CT its compensating transaction. Let S_0 and S_1 be the initial- and end-state respectively such that we have;

$$\{S_0\}T; H; CT\{S_1\}$$

The history $T; H; CT$ is *sound* if there exists a history H' of $dep(T)$ such that $\{S_0\}\ H'\ \{S_1\}$ is a valid Hoare-triple.

Using these definitions, we formally define the decision problem $HistSound(DB, H, T, CT, S_0, S_1)$ as follows;

Definition 1. *Given a local database DB, compensated-for transaction T, compensating transaction CT, a history H of $dep(T)$, an initial state S_0 and end-state S_1: is $T; H; CT$ sound?*

Given the definition of soundness, this amounts to answering the question '*does there exist a history H' of only the transactions in $dep(T)$ such that $\{S_0\}\ H'\ \{S_1\}$ is a valid Hoare-triple?*'

In view of distributed transactions, soundness of histories of the transactions performed on all local databases in the network is a rather weak consistency criterion;

- Let $T_i \in dep(T)$. Execution of T_i in the history H' with initial state S_0 may lead to T_i returning a result to its environment (a user, program or other transaction) different from that produced in the history $T; H$ with initial state S_0. It depends on the application whether or not this is acceptable.

- Soundness of all local databases does not guarantee consistency of the distributed database.

Even still, we prove theorem 1 which states that if $HistSound$ is in $\mathcal{NP}$, then it is intractable. If $HistSound$ is not in $\mathcal{NP}$ then it is probably even harder.

Before proving the theorem, we first discuss why almost any real-life instance of $HistSound$ is in $\mathcal{NP}$. In order for $HistSound$ to be in $\mathcal{NP}$, it must be that a certificate of soundness of a given history H - some other history H'- can be checked in polynomial time. A straightforward way to check the certificate is to execute H' from initial state S_0 and check if S_1 holds. Below we argue that checking a state and executing the transactions can usually be done in an amount of time polynomial in the size of the inputs;

- Let $|DB|$ denote the number of entities in the database DB. We assume that initial- and end-states S_0 and S_1 are predicates of the form $e_1 = v_1 \wedge e_2 = v_2 \wedge \ldots \wedge e_{|DB|} = v_{|DB|}$, where e_i is an entity with value v_i, $1 \leq i \leq |DB|$. If the state of the database does not satisfy S_0, we must first initialize DB by performing at most $|DB|$ assignments: one for each entity occurring in S_0. It follows we can usually check S_1 in $\mathcal{O}(|DB|)$ time.
- Transactions usually have have an execution-time polynomial in $|DB|$, $\mathcal{O}(|DB|^L)$ say, for some constant L. Let $|H'|$ denote the number of transactions in the history H' of the certificate. Obviously $|H'| = |H|$. Hence we can execute H' in $\mathcal{O}(|H||DB|^L)$ time.

Hence the total amount of time needed to check the certificate will in most cases be bounded by a polynomial in the size of the problem.

Theorem 1. *If $HistSound \in \mathcal{NP}$ then $HistSound$ is $\mathcal{NP}$-hard.*

Proof. Assume $HistSound \in \mathcal{NP}$. We have three proof obligations;

- *Prove that $HistSound$ is in $\mathcal{NP}$.* This follows from our assumption.
- *Provide a polynomial-time reduction of a known $\mathcal{NP}$-hard problem to $HistSound$.*

  We provide a polynomial-time reduction to $HistSound$ of one-processor scheduling with release-times and deadlines which was shown to be NP-hard in [LKB77].

  This particular scheduling problem is defined as follows. Given K tasks labeled i, $1 \leq i \leq K$, each task has a release time rt_i, deadline dl_i and processing-time pt_i, all positive reals. Let A be any positive amount of time: the global deadline. Then the questions is;

  > *Does there exist a schedule that allows processing of all tasks within A time-units, such that each task i is started after its release-time rt_i and completed before its deadline dl_i ?*

  In our database DB, we put for each task i, $1 \leq i \leq K$, the real constants rt_i, dl_i and pt_i. In addition we add the real constant A, the integer constant K, the integer variable ntp (initially 0), a single boolean variable $failed$ (initially $false$) and a single real variable $time$ (initially 0).

  For task i, we define transaction T_i given below, which simulates the execution of task i after ntp tasks have been successfully processed in $time$ time-units. The variable $failed$ indicates if any task scheduled before task i failed to be processed.

```
$T_i = |[$ var $b : \mathbb{B}$;
        $b := (rt_i \leq time \wedge time + pt_i \leq dl_i \wedge time + pt_i \leq A)$;
        $b := b \vee (time \leq rt_i \wedge rt_i + pt_i \leq dl_i \wedge rt_i + pt_i \leq A)$;

        if  $failed \rightarrow$ skip;
        $[]$ $\neg failed \wedge \neg b \rightarrow failed := true$;
        $[]$ $\neg failed \wedge b \rightarrow time, ntp := pt_i + MAX(rt_i, time), ntp + 1$;
        fi

        if $ntp = K \rightarrow time, ntp := 0, 0$;
        $[]$ $ntp \neq K \rightarrow skip$;
        fi
     $]|$
```

The variable b is initialized in two steps and has the value $st_1 \vee st_2$ where st_i is the righthand-side of the assignment in step i. An explanation is given below;

- st_1 expresses that task i may be started immediately and that it then meets its own deadline and the global deadline A.
- st_2 expresses that task i may be started only after waiting $rt_i - time$ time-units and that it then also meets its own deadline and the global deadline A.

The first **if**-statement extends the schedule with task i only if $\neg failed \wedge b$ holds. If not, $failed$ is assigned the value $true$ only if $\neg failed$ holds as precondition (we explain the necessity of this precondition in the sequel). Note that there is no point in waiting after the release-time of a task has passed if the processor is idle. Hence we refrain from implementing this.

The last **if**-statement sets $time$ and ntp to zero if T_i was the last in a history of all transactions corresponding to a feasible schedule of all tasks. We explain the use of this. Let H be any history of $\{T_i \mid 1 \leq i \leq K\}$, define S_0 and S_1 as follows;

$$\begin{array}{lcl} S_0 & : & \neg failed \wedge\ time = 0\ \wedge\ ntp = 0 \\ S_1 & : & \ \ failed \wedge\ time = 0\ \wedge\ ntp = 0 \end{array}$$

If we execute H from S_0, then the end-state satisfies S_0 if and only if H corresponds to a feasible schedule of the corresponding tasks. If we execute H from S_1, then the end-state satisfies S_1 because none of the tasks can be processed. The only difference between S_0 and S_1 is the value of $failed$. We exploit this in our definition of the compensated-for transaction T_0 and compensating transaction CT_0;

$$T_0 = CT_0 = [\![\ failed := \neg failed\ ]\!]$$

Clearly the entire transformation can be made in $\mathcal{O}(K)$ time, and we have for all initial states S that CT_0 corresponds to the classical undoing of T_0:

$$\{S\}\ T_0; CT_0\ \{S\}$$

Also, for all histories H of $dep(T_0)$, we have that $dep(T_0)$ equals $\{T_i | 1 \leq i \leq K\}$ in the history $T_0; H$. Indeed, recall that $failed$ is assigned the value $true$ only if $\neg failed$ holds as precondition. Suppose we would allow this assignment from a state in which $failed$ already holds. Then by definition of dep, the first transaction in H, T_j say, would destroy the dependency relation between all other transactions in H and T_0.

- *Prove that the answer to an instance of the problem to be reduced to $HistSound$ always equals the answer to its transformation to $HistSound$.*

  Note that the following Hoare-triple holds for all histories H of $\{T_1 \mid 1 \leq i \leq K\}$:

  $$\{S_0\}\ T_0; H; CT_0\ \{S_0\}$$

  Indeed, we have;

  $$\{S_0\}\ T_0;\ \{S_1\}\ H;\ \{S_1\}\ CT_0\ \{S_0\}$$

  Given an oracle that solves $HistSound$, we let it solve $Histsound(DB, H, T_0, CT_0, S_0, S_0)$.

  If the oracle answers 'yes', then it must be the case that $\{S_0\}\ H'\ \{S_0\}$ is valid for the certificate H', some history of $\{T_i \mid 1 \leq i \leq K\}$. Hence, upon execution of H' it must be that during each transaction, the third alternative is chosen in its first `if`-statement. This corresponds to each task i being started after rt_i and completed before dl_i: H' indeed corresponds to a valid schedule of all tasks.

If the oracle answers 'no' then the state S_0 cannot be reached by any history H' of $\{T_i \mid 1 \leq i \leq K\}$, i.e. $T_0; H; CT_0$ is not sound. Hence there exists no feasible schedule of the tasks i that allows processing of all tasks within A time-units.

□

Until now we have not discussed exactly what a compensating transaction should accomplish. Compensation is treated in [KLS90] as an application-dependent activity. The same paper defines three constraints for compensating transaction, some or all of which may or may not be applicable, depending on the application. Below we define Constraint 1*, somewhat similar to Constraint 1 defined in [KLS90].

Definition 2. Constraint 1*: *let S_0 be the state of the database before transaction T is started. Then CT is a compensating transaction if $\{S_0\}T; CT\{S_0\}$ holds*

Constraint 1* seems applicable in for example, transactions that transfer money: it is desirable that CT simply undoes the effects of T.

We formalize the connection between soundness and the defining of a compensating transaction that satisfies Constraint 1* by means of the decision problem
$CompSound_{C1^*}(DB, T, H, S_0, S_1)$;

Definition 3. *Given a local database DB, compensated-for transaction T, a history H of dep(T), an initial state S_0 and end-state S_1: does there exist a compensating transaction CT such that;*

- $T; H; CT$ is sound, and,
- $\{S_0\}\, T; CT; \{S_0\}$ holds ?

We prove theorem 2 which states that in its most general form, $CompSound_{C1^*}$ is intractable.

Theorem 2. *If $CompSound_{C1^*} \in \mathcal{NP}$ then $CompSound_{C1^*}$ is $\mathcal{NP}$-hard.*

Proof. We use almost the same reduction as given in the proof of theorem 1. The only modification is that our oracle must now solve $CompSound_{C1^*}(DB, T, H, S_0, S_0)$: it must find CT_0 by itself. If the oracle answers 'yes' it must be able to provide a certificate containing both a history H' of dep(T_0) and CT_0 such that $\{S_0\}T_0; CT_0\{S_0\}$ and $\{S_0\}H'\{S_0\}$ hold.

Note that by our definitions of T_0 and S_0, we have that if the oracle answers 'yes', the compensating CT_0 contained in the certificate must be semantically equivalent to T_0. Indeed: transaction T_0 only negates $failed$ and the state of DB is uniquely determined by S_0. Hence we have as in the proof of theorem 1 that the oracle can answer 'yes' if and only if a feasible schedule of the K tasks exist that allows them all to be executed within A time units. □

In section 1.3.2.1 it was claimed that compensation is possible and tractable for transactions that commute and all have compensating transactions. Two transactions T_i, T_j commute [KLS90] if the following holds for all states S_0, S_1 we have: $\{S_0\}T_i;T_j\{S_1\}$ implies $\{S_0\}T_j;T_i\{S_1\}$. Let $HistSound_{comm}$ be the restriction of $HistSound$ to commutative transactions.

Theorem 3. $HistSound_{comm} \in \mathcal{P}$

Proof. We provide an algorithm that solves $HistSound_{comm}(DB, H, T_0, CT_0, S_0, S_1)$ in polynomial time. Here H is a history of *dep*(T_0), where *dep*$(T_0) = \{T_i | 1 \leq i \leq K\}$. The transactions are on the database DB, the initial state is S_0 and end-state is S_1. Also, all transactions, including compensating transactions, commute. The algorithm is as follows: return 'yes' and provide as certificate: H.

To see why this is correct note that by commutativity, we may permute the order of the transactions without affecting the end-state;

$$\begin{aligned}
&\{S_0\}\ T_0;T_1;\ldots;T_K;CT_0\ \{S_1\}\\
\equiv\ &\{\text{ commutativity, } K \text{ times }\}\\
&\{S_0\}\ T_0;CT_0;T_1;T_2;\ldots;T_K\ \{S_1\}\\
\equiv\ &\{\text{ definitions of } T_0, CT_0\ \}\\
&\{S_0\}\ T_1;T_2;\ldots;T_K\ \{S_1\}
\end{aligned}$$

Our algorithm takes $\mathcal{O}(1)$ time, i.e. a polynomial in the size of our problem. □

1.3.3 Allowing additional F&F-messages

In the previous sections it became clear that the commitment problem present in update-transactions cannot be easily solved if we allow R/R-messaging only. Replacing all requests and replies by 'F&F-requests' and 'F&F-replies' was already rejected in section 1.2. Hence the only viable option left to investigate is the use of F&F-messages *in addition* to request and replies.

Reconsider the example distributed transaction discussed in section 1.2 in connection to figure 1.2. Recall that the client aborted in scenarios 2 and 3, while the service only 'aborted' in scenario 2. We can let the client send an extra F&F-message (after it received a reply or timeout) containing an instruction for the service to commit or abort its local database transaction. Of course the client may instruct the service to commit only in scenario 1, in scenarios 2 and 3 the instruction should be to abort. By reliability of F&F-messaging, this extra message arrives eventually and can be used to let the client and service agree on their commit decisions.

In chapter 2 we present the n-R/R protocol which allows for ACID distributed transactions between one client and n different services, $1 \leq n$, thanks to the use of additional F&F-messages as described above. The integer n is called the *size* of the distributed transaction. The n-R/R protocol allows the client and services involved in a distributed transaction to vote whether or not to commit. Distributed transactions are only committed if the client and all services unanimously vote to commit and are aborted otherwise.

The number of extra F&F-messages sent during a single distributed transaction is only n, preceded by n requests and n replies from the R/R-pattern. Hence the total number of messages sent in a distributed transaction equals $3n$. Given the drawbacks of the solutions proposed in sections 1.3.1 and 1.3.2 and the small number of (F&F-) messages needed in the protocol, it seems by far the cheapest, fastest and most universal solution possible, especially in a loosely-coupled setting. Indeed, the protocol is such that clients and services may disconnect from the bus as long as they are not in the middle of a transaction without causing deadlocks or a loss of integrity of the distributed database.

The protocol is verified in chapters 3 and 4. Chapter 3 verifies the protocol for transactions with size 1: the 1-R/R protocol. Chapter 4 generalizes the results from chapter 3 to transactions with any positive size n: the n-R/R protocol. The distinction between the two protocols was made only in order to ease the verification proofs.

In the proposed solution we assume clients and services to be single-threaded: at any time, a client or service can take part in only one distributed database transaction. Also, we assume that neither clients nor services crash, or put differently, backup and recovery has been implemented such that crashes cannot be noticed by other clients or services different from the one that crashed (and recovered). The extension to multi-threaded services or clients is not difficult, we omitted it because of time constraints of this project.

Chapter 2

Description of the n-R/R protocol

2.1 Introduction

In this chapter we describe the n-R/R protocol. The protocol allows a single client and any number of services to agree on the outcome of a distributed transaction (see chapter 1 for a more detailed problem description).The n-R/R protocol can be used to solve other consensus problems as well: we present the protocol as a distributed database transaction protocol because most IT-professionals are familiar with this subject.

The client-program of the n-R/R protocol is discussed in section 2.2, that of the service in section 2.3. In section 2.4 we give some example runs of the protocol by means of message sequence charts.

Some important technical details are discussed in section 2.5 that must be considered by implementors of the protocol. We also argue that requirements 3 and 4 of the problem statement given on page 17 are met (requirements 1 and 2 are covered in chapter 4)

2.2 The client program

In section 2.2.1 we give a high-level description of the client-program. The purpose of section 2.2.1 is twofold;

1. to provide a basic, operational understanding of the workings of the client program;
2. to serve as an introduction for the pseudocode version of the client program presented in section 2.2.2.

The pseudo-code version of the client-program removes the ambiguity present in the high-level description.

2.2.1 High-level description

A flow diagram of the client program is shown in figure 2.1. The boxes in figure 2.1 contain the actual labels used in the pseudo-code of the client-program. The flow-diagram contains a directed path from the box labeled 'Input & Begin DB-Transaction' to the inverted triangle 'Stop?' (a choice). The statements corresponding to the labels on that path correspond to the client's processing of a distributed transaction. If the client must start another distributed transaction, it chooses 'no' at the choice 'Stop?' and cycle back to the box labeled

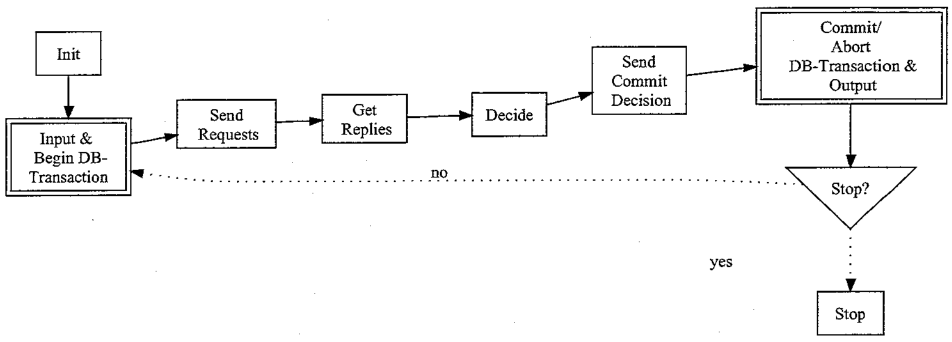

Figure 2.1: Client-program: flow-diagram

'Input & Begin DB-Transaction', otherwise it chooses 'yes' after which the client-program terminates.

Having explained the global structure of figure 2.1, we can explain the effects of the statements associated with the various labels;

Init each time the client program is (re-)started, it must go through a brief initialization phase. Among other things, a connection must be made with the IFSA-bus.

Input & Begin DB-Transaction This is the first label of an actual local transaction. The client-program receives input from its environment which leads to a distributed transaction of which it must ultimately return the result. The environment can be another program, machine or a human using a user-interface. After examining the input, the client program decides which services must be consulted in order for the output to be returned. Also, the client starts a local database transaction.

Send Requests Corresponding to the input received, the client creates a request for each service needed in the distributed transaction and sends it to the corresponding service.

Get Replies For each request sent, it must get a reply (or a timeout). The replies contain all the usual information offered by the corresponding service and also a vote from the service to commit or abort the distributed transaction. Services must vote to abort if this is dictated by their business-application logic, or, if some error occurred (we return to this in section 2.3).

Decide *If* no timeout was received in the **Get Replies**-phase *and* all of the services that take part in the distributed transaction voted to commit, *then* the client program *may* decide to commit. *Else*, the client *must* decide to abort. Note that business-application logic, malformed replies or errors during processing by the client may lead to a decision by the client to abort the distributed transaction.

Send Commit Decision A F&F-message is sent to each service part of the distributed transaction, informing them of the commit decision taken in the previous phase.

Commit/Abort DB-Transaction & Output Corresponding to the commit decision, the client either commits or aborts its local database transaction and returns the result of the distributed transaction to its environment.

Note that all double-boxes in figure 2.1 contain actions that begin, commit or abort a local database transaction: we use this convention throughout this chapter.

Some remarks are in place;

- Note that all requests are sent before all replies (or timeouts) are received and before all commit decisions are sent. This allows a fair amount of parallelism: services that were already sent a request by the client can start processing while the client is still sending requests to other services. Distributed transactions generally take longer if a request is sent to a service and the corresponding reply or timeout is received before sending another request to another service.

- The protocol does not work correctly if some service is used more than once in a single transaction: all requests within a distributed transaction *must* be sent to different services.

- The mutual order in which requests are sent is of no concern and can be chosen freely. The same holds for the mutual order in which replies are received and the mutual order in which commit decisions are sent.

- Throughout this chapter, the (emphasized) key words *'must', 'must not', 'may'*, and *'may not'* are to be interpreted as described in RFC 2119 [Bra97].

- The client program may only shutdown at the choice **Stop?** and nowhere else. In fact, any deviation by the client from the protocol may have undesirable consequences such as deadlock of (other) clients or services or inconsistency of the distributed database. We return to this point in section 2.5.

2.2.2 Pseudo-code description

The program-variables and constants referred to in the pseudo-code of the client program are given in table 2.1. Note that in an actual implementation, more variables may be needed. If a type is prefixed by '+' in the second column, we mean that only unsigned numbers are needed for the corresponding implementation variable or constant. For arrays, the type in the second column is followed by its size between square brackets if known, or '[ ]' if not.

In the third column it is indicated whether or not a variable or constant requires initialization before a distributed transaction can be started. This initialization must be performed during the **Init**-phase of the client-program on page 31. The only constant is `MAX_SZ`: the other identifiers are variables.

Identifier	*Type*	Initialization required?
`MAX_SZ`	`+int`	yes
`sz`	`+int`	no
`tid`	`+int64`	yes
`ssRR`	`IFSAHSRVSLOT[MAX_SZ]`	no
`ssFF`	`IFSAHSRVSLOT[MAX_SZ]`	no
`req`	`IFSAHMSGH[MAX_SZ]`	no
`plRep`	`char[MAX_SZ][ ]`	no

Table 2.1: Variables needed in the client-program

A description of all variables present in table 2.1 is given below;

- `MAX_SZ`: this constant equals the maximum number of services that may be used in any single distributed transaction: the maximum *distributed transaction size* allowed.

- `sz`: this variable is used to store the actual distributed transaction size during a distributed transaction: $1 \leq$ `sz` $\leq$ `MAX_SZ`.

- `tid`: this variable *must* be loaded from stable storage when the client program is (re-) started. If the client program is terminated, the value of `tid` *must* be saved to stable storage. This variable is used for global identification of distributed transactions and the corresponding local transactions, hence the requirement of stable storage. At the end of a distributed transaction, the variable `tid` equals the sum of the size of all distributed transactions *ever* performed by the client. Hence it *must* equal zero if the client program of client c never ever performed a distributed transaction. But if we shutdown the program after some transactions were processed, `tid` *must not* start from zero again! In summary, `tid` is a *persistent variable*.
- `ssRR` and `ssFF` are arrays of service-slots (see section A.2.1). The arrays are used to store the different service-slots of the services used in a distributed transaction. During a transaction, `ssRR`$[s]$ contains the R/R-service-slot of service s and `ssFF`$[s]$ contains its F&F-service-slot. Note that we require 2 service-slots for each service that participates in a distributed transaction! This is clarified in section 2.3.
- `req`: during a transaction, `req`$[s]$ is used to store the request-handler of the request sent to service s.
- `plRep`: during a transaction, `plRep`$[s]$ is be used to store the payload of the corresponding reply from service s.

Table 2.2 summarizes which variables *must* occur in the payloads of messages sent *in addition* to any other client or service-dependent variables. Within a distributed transaction, the request and commit-decision sent to a single service always carry the same transaction-identifier `tid` and no other request or commit decision sent by the same client carries the same transaction-identifier `tid`. Note that we do not need a variable `tid` in the reply: if the

Message	*Identifier*	*Type*	*Explanation*
Request	`tid`	`+int64`	Transaction-identifier
Reply	`sv`	`bool`	The service-vote: this variable has value `true` if the sender (a service) can commit its transaction, it has the value `false` if the sender must abort its transaction.
Commit-Decision	`tid`	`+int64`	Transaction-identifier
	`cd`	`bool`	This variable has value `true` if the sender *must* commit its transaction, it has value `false` if the sender *must* abort its transaction.

Table 2.2: Additional variables needed in messages

client receives a reply, the BAI-method `IFSAgetAnyReply` also returns the corresponding request (see section A.4.3).

Some variables occur both in the client-program and in messages. In the sequel we subscript a variable with c to indicate that a variable is a client-program-variable to avoid confusion. For example, $\texttt{tid}_c$ is the client-program-variable `tid`.

The pseudo-code version of the client program is given on page 31. The program consists of several phases labeled by a heading in bold font (for example: **Decide**). Each phase may consist of one or more steps labeled by numbers.

Init Load $\texttt{tid}_c$ from stable storage and connect to the IFSA-bus.

Input & Begin DB-Transaction During this phase of the protocol, the client program typically accepts some sort of question or instruction from either another program or a user-interface. Because of the large number of possible scenarios, we only describe the state in which the client program must be before proceeding to the label '**Send Requests**' and assume the client program to have started a local database transaction. In addition we require that:

- sz equals the distributed transaction size *and* $1 \leq \texttt{sz} \leq \texttt{MAX\_SZ}$ *and*
- for all s, $0 \leq s < \texttt{sz}$:
  - ssRR[s] contains the R/R-service-slot of service s *and*,
  - ssFF[s] contains the F&F-service-slot of service s

Send Requests In this phase, all requests are sent to the services chosen during the previous phase. This corresponds to performing the following once for each s, $0 \leq s < \texttt{sz}$:

1. create the request req[s] using the service-slot ssRR[s], a payload of the request containing:
   - all fields and values required by service s
   - an additional integer variable tid with the value $\texttt{tid}_c + s$
2. send the request req[s] by means of the BAI-method IFSASend (see section A.3.1.2) using the request, payload and service-slot from the previous step.

Get Replies The replies corresponding to the requests sent during the previous phase are received in this phase. This corresponds to doing the following sz times:

1. call the BAI-method IFSAgetAnyReply (see section A.4.3) to receive any pending reply: the BAI must return a reply or timeout, the corresponding request, and the payload of the reply (which equals NULL if a timeout is returned).
2. Let the request returned in the previous step be req[s'], $0 \leq s' < \texttt{sz}$. Store the corresponding payload returned in the previous step (possibly NULL) in plRep[s'].

Decide We reach a commit decision as follows;

1. *if* no timeout was received during the previous phase *and* in all payloads plRep[s], $0 \leq s < \texttt{sz}$, the boolean variable sv equals true, *then* the decision to commit or abort the local database transaction is free to make, *else* it *must* be decided to abort.
2. *if* the decision to commit or abort the local database transaction is free to make, *then* perform any processing of the replies necessary to reach a commit decision and decide.

Send Commit Decision

1. Begin a unit of work by means of the BAI-method IFSABeginUOW (see section A.5).
2. Perform the following once for each s, $0 \leq s < \texttt{sz}$:
   (a) create a F&F-message using the service-slot ssFF[s] and a payload containing;
   - all fields and values required by service s
   - an additional integer variable tid with the value $\texttt{tid}_c + s$
   - an additional boolean variable cd with the value: *if* it was decided to commit *then* choose cd = true, *else* choose cd = false.

   (b) send the F&F-message by means of the BAI-method IFSASend (see section A.3) using the message, payload and service-slot from the previous step.
3. Commit the unit of work using the BAI-method IFSACommitUOW (see section A.5).

Commit/Abort DB-Transaction & Output

1. *If* the decision was to commit *then* perform any processing necessary and commit, *else* perform any processing necessary and abort.
2. Increase tid_c by sz and output the result of the distributed transaction to the environment.
3. *If* another distributed transaction must be started *then* goto the label '**Input & Begin DB-Transaction**' *else* store $\texttt{tid}_c$ to stable storage and stop.

2.3 The service program

In this section we discuss the service-program of the n-R/R protocol. The structure of this section is similar to section 2.2: a high-level description of the service-program is given in section 2.3.1, a pseudo-code description is given in section 2.3.2.

2.3.1 High-level description

A flow diagram of the service-program is shown in figure 2.2. The boxes in figure 2.2 contain the actual labels used in the pseudo-code of the service-program. The flow-diagram

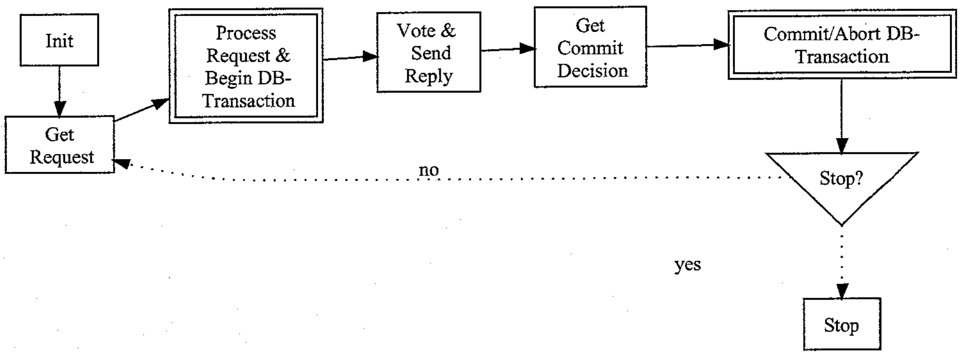
Init
Process Request & Begin DB-Transaction
Vote & Send Reply
Get Commit Decision
Commit/Abort DB-Transaction
Get Request
no
Stop?
yes
Stop

Figure 2.2: Service-program: flow-diagram

contains a directed path from the box labeled 'Get Request' to the inverted triangle 'Stop?' (a choice). The statements corresponding to the labels on that path correspond to the service's processing of a distributed transaction. If the service must process another transaction, it chooses 'no' at the choice 'Stop?' and cycle back to the box labeled 'Get Request', otherwise it chooses 'yes' after which the service-program may terminate at the label 'Stop'. Note that the service-program may only stop here!

Having explained the global structure of figure 2.2, we can explain the effects of the statements associated with the various labels;

Init each time the service program is (re-)started, it must go through a brief initialization phase. Among other things, a connection must be made with the IFSA-bus.

Get Request This label is the first of an actual transaction: the service-program gets a request from the IFSA-bus.

Process Request The service-program analyzes the request, starts a local database transaction and executes the required query. The local database transaction may not yet bet committed!

Vote & Send Reply Send the reply containing the query result and also, a vote to commit or abort the distributed transaction.

Get Commit Decision Receive a F&F-message from the client called the *commit-decision*: this message contains an instruction to commit or abort the local database transaction of the service.

Commit/Abort DB-Transaction Commit or abort the local database transaction in accordance with the client's commit decision.

2.3.2 Pseudo-code description

The variables and constants referred to in the pseudo-code of the service program are given in table 2.3. If a type is prefixed by '+' in the second column, we mean that only unsigned numbers are needed for the corresponding implementation variable or constant.

In the third column it is indicated if a variable or constant requires initialization before a transaction can be processed. This initialization must be performed during the **Init**-phase of the service-program on page 34.

Identifier	*Type*	Initialization required?
`c`	`+int`	no
`conFF`	`IFSAHCONN`	yes
`conRR`	`IFSAHCONN`	yes
`ct`	`+int64`	no
`req`	`IFSAHMSGH`	no
`rep`	`IFSAHMSGH`	no

Table 2.3: Variables needed in the service-program

A description of the variables is given below: we only explain the initialization of a variable if the last entry of the corresponding row equals 'yes';

- `c`: this variable is used to identify the address of the client in a transaction. At the moment of writing of this thesis, the actual address of a client cannot be determined by calling a BAI-method. However, this information *is* accessible to layer below the BAI but has been hidden for application programmers. There are three simple solutions to this problem;
    - The entire protocol can be implemented at a layer under the BAI which *can* access the address of the sender of a message.
    - The adress of a sender can be made available by means of a method in an update of the BAI.
    - Clients can themselves add their address or some identifier unique to the client both to the payloads of the request and commit-decision sent in a protocol run. To avoid confusion it is advisable to use the same name for this payload-variable: `c`.

  Of course, the first two solutions are to be preferred. In reality, the type of `c` may differ from `int`. We simply model it as such.
- `conFF` and `conRR` represent two *different* connections of the service to the IFSA-bus. Both variables must be initialized correspondingly during the phase **Init**. All F&F-messaging must be done using `conFF`, all R/R-messaging must be done using `conRR`. The protocol can *not* work correctly if only one connection is made to the bus as the BAI provides no functionality (yet) that lets the service specifically get a F&F-message instead of a request (or vice versa).
- `ct`: this variable is used to store the transaction-id during a transaction.
- `req`: during a transaction, `req` is used to store the request-handler of the request received.
- `rep`: during a transaction, `rep` is used to store the reply-handler of the reply (to be) sent.

For variables occurring in messages we refer to table 2.2. The pseudo-code version of the service-program is given below.

Init Connect to the IFSA-bus: store the R/R-connection in the variable `conRR`, store the F&F-connection in the variable `conFF`. Register the service in the bus-repository.

Get Request Receive a request using the BAI-method `IFSAGetMessage` using the connection `conRR` and copy it to `req`. Note that this actually involves repeatedly invoking `IFSAGetMessage` until a request was indeed available and received (see section A.4.1). Analyze the payload, and copy the client adres in program variable `c` and the payload-variable `tid` in `ct`.

Process Request Process the local database query corresponding to the request such that any changes to the local database can be undone at the end of the transaction. Typically, this involves the following steps:

1. start a local database transaction
2. process the query.

Vote & Send Reply

1. Create a reply by means of the BAI-method `IFSACreateReply` (see section A.3.4) using the request `req`, the reply `rep`, and a payload containing:
   - the query-result and all fields and values required by the service
   - the variable `sv` with value: *if* the local database transaction can be committed *then* set `sv` to `true`, *else* set `sv` to `false`. Note however that the database transaction *must not* yet be committed or aborted! Also note that malformed requests, errors during query-processing or business-application-logic may all lead to a vote to abort.
2. Send the reply `rep` by means of the BAI-method `IFSASend` (see section A.3) using the connection `conRR` and the reply and payload from the previous step.

Get Commit Decision this amounts to receiving F&F-messages from `conFF` until a message from client `c` is received with payload-variable `tid` equal to `ct`;

1. begin a unit of work using the BAI-method `IFSABeginUOW` (see section A.5) with connection `conFF`.
2. *Repeat:* receive a F&F-message by means of the BAI-method `IFSAGetMessage` (see section A.4.1) using the connection `conFF`, a suitable value for `WAIT_DELAY`, the message handler `mFF` and the payload handler `plFF`.
   Until: the payload-variable `tid` in `plFF` equals `ct` *and* `mFF` was sent by client `c`

Commit/Abort DB-Transaction

1. *If* the payload-variable `cd` equals *true*, *then* commit the local database transaction; *else* abort the local database transaction.
2. commit the unit of work using the BAI-method `IFSACommitUOW` (see section A.5) with connection `conFF`.
3. *If* the service must be shut down *then* stop *else* goto the label **Get Request**.

2.4 Example transactions

In this section we give two examples of scenarios of distributed transactions. The first scenario is discussed in section 2.4.1. It concerns a distributed transaction between one client and two services (size 2). The second scenario is discussed in section 2.4.2. It concerns three distributed transaction, all with size 1. The scenarios are meant to increase the understanding of the workings of the n-R/R protocol.

2.4.1 Scenario: a distributed transaction with size 2

Figure 2.3 gives a graphical representation of a possible scenario of the execution of a single transaction between one client c_0 and two services s_1 and s_2. Some things were left out in order to reduce the size and complexity of the figure: the exact payload of the messages sent, the manipulations of program-variables in the client or services, interaction of the client and services with the outside world, etc.

We give an informal description of the scenario given in figure 2.3. The distributed transaction $\mathcal{T}$, $\mathcal{T} = T0_{c_0} \| T1_{s_1} \| T2_{s_2}$, has size 2: two different services are involved. Because s_1 and s_2 are idle in the beginning, they can both service the request sent by c_0. The replies are sent in time by both s_0 and s_1. Also, c_0 receives both replies in time. Both services voted to commit the distributed transaction. Hence c_0 may decide to commit and does so. After having received the commit decision sent by c_0, s_1 and s_2 commit, as does c_0. Figure

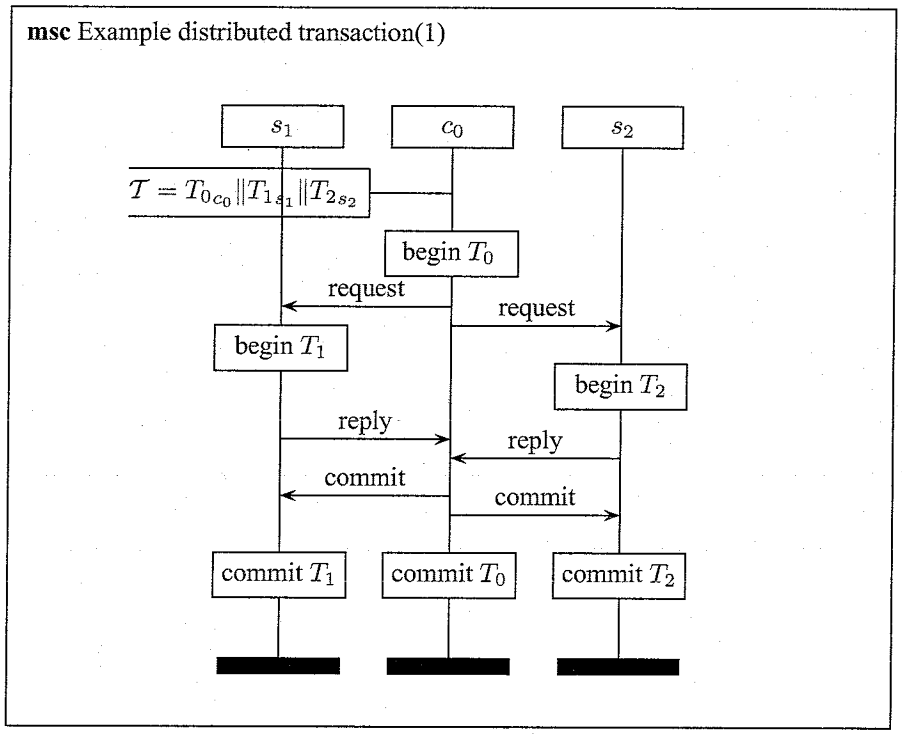

Figure 2.3: n-R/R-protocol: example distributed transaction (1)

2.3 illustrates that the protocol allows a fair amount of parallel processing. Indeed: actions of different components appearing at the same vertical positions are executed in parallel.

2.4.2 Scenario: three distributed transaction with size 1

Figure 2.4 gives a graphical representation of a somewhat more complex scenario of the execution of three distributed transactions, all with size 1. In the scenario we have two clients c_0 and c_1, and a single service s_0.

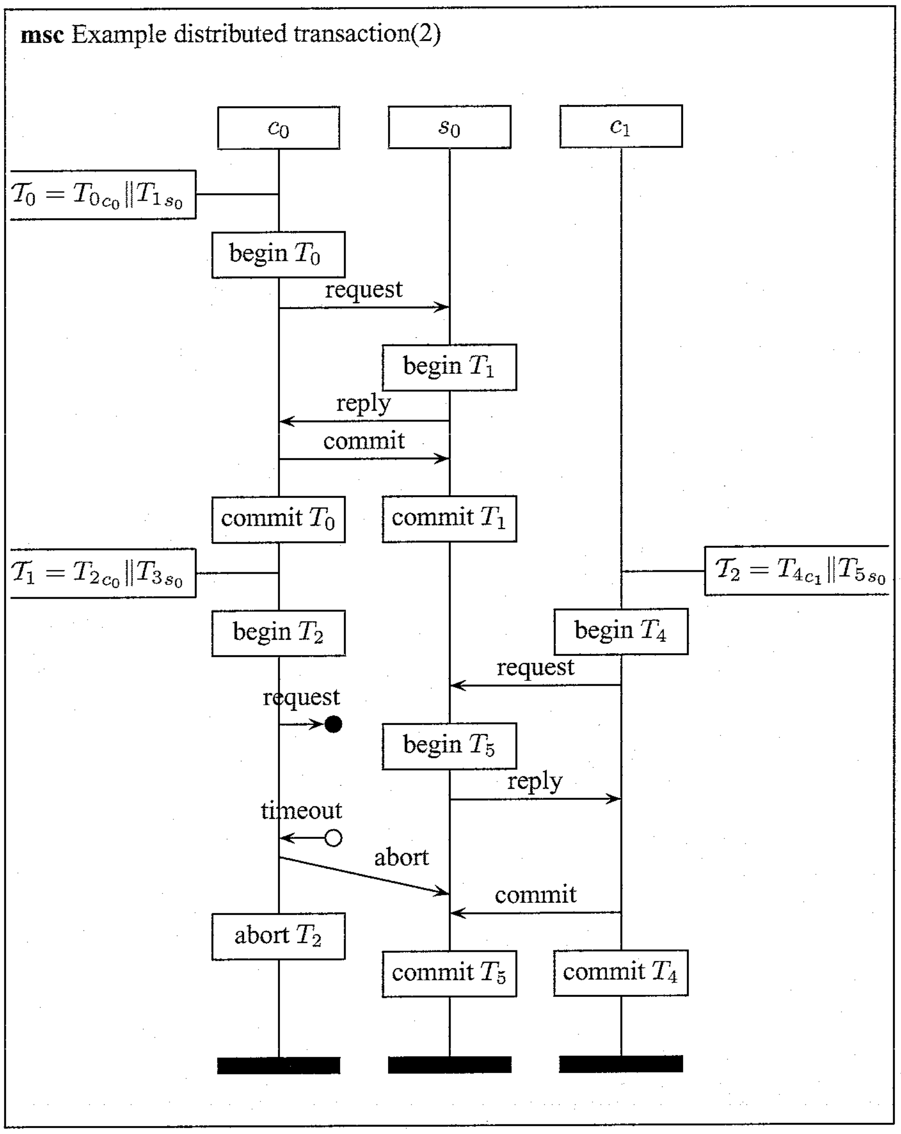

Figure 2.4: n-R/R-protocol: example distributed transaction (1)

Below we give an informal description of figure 2.4;

- $\mathcal{T}_0$, $\mathcal{T}_0 = T_{0c_0} \| T_{1s_0}$: the first distributed transaction, between c_0 and s_0. Because s_0 is idle, it can service the request sent by c_0. The reply is sent in time by s_0 and received in time by c_0 and contains a vote of s_0 to commit the distributed transaction. Hence c_0 may decide to commit and does so. After having received the commit decision sent by c_0, s_0 commits, as does c_0. In the meantime, client c_1 was idle.
- $\mathcal{T}_1$, $\mathcal{T}_1 = T_{2c_0} \| T_{3s_0}$: this distributed transaction is executed in parallel with $\mathcal{T}_2$. Client c_0 sent its request to s_0, but s_0 was also sent another request which is al-

ready being processed. Because s_0 does not respond in time, c_0 receives a timeout. Consequently, it must decide to abort. It sends this decision to s_0: s_0 receives this decision but, being in another transaction, simply ignores it. Finally, client c_0 aborts the database transaction it started.

- $\mathcal{T}_2$, $\mathcal{T}_2 = T_{4c_1} \| T_{5s_0}$: this distributed transaction is executed in parallel with $\mathcal{T}_1$. Client c_1 sent it request to s_0 and service s_0 is able to process it. The only difference with transaction $\mathcal{T}_0$ is that this time, service s_0 receives two commit decisions. One belonging to $\mathcal{T}_1$ (abort), the other belonging to $\mathcal{T}_2$ (commit). The service can however determine which commit decision belongs the transaction it is in by means of the combination of the sender and the transaction-identifier in the payload of the messages.

2.5 Considerations

In this subsection we discuss some important details of the n-R/R protocol that were left out of sections 2.2 and 2.3. We also argue that requirements 3 and 4 of the problem statement given on page 17 are met (requirements 1 and 2 are covered in chapter 4);

- In section 2.5.1 we discuss the implementation of the transaction-identifiers.
- In section 2.5.2 we discuss deadlock and suitable values for `MAX_SZ` and `MAX_TIMEOUT` and argue that requirement 3 of the problem statement given on page 17 is met.
- In section 2.5.3 we discuss the efficiency of the n-R/R protocol and argue that requirement 4 of the problem statement given on page 17 is met.
- In section 2.5.4 we discuss how to handle exceptions raised by the BAI.
- Section 2.5.5 discusses some limitations of the n-R/R protocol.

2.5.1 The transaction-identifier

In the design of the protocol, the client-program variable $\texttt{tid}_\texttt{c}$, the payload-variable `tid` and the service-program variable $\texttt{ct}_\texttt{s}$ are used to uniquely identify transactions.

It is important to understand that if multiple requests (or replies or F&F-messages) carry the same transaction-identifier, the protocol is *not* guaranteed to function properly. Hence the requirement that $\texttt{tid}_\texttt{c}$ is stably stored after termination of the client program and loaded on a restart.

However, the machine integer used to represent $\texttt{tid}_\texttt{c}$, cannot grow forever: at some point it overflows. In other words, there is a danger that partial-transactions cannot be uniquely identified causing the protocol to malfunction. The chance of the protocol malfunctioning becomes smaller if we use more bits to represent $\texttt{tid}_\texttt{c}$. One should use 64-bit integers to represent $\texttt{tid}_\texttt{c}$, the payload-variable `tid` and the service-program variable $\texttt{ct}_\texttt{s}$ which take forever (in computer terms) to overflow. Indeed, if a client would be in operation for 100 years, it would have to process about 5,849,424,174 transactions per second on average for $\texttt{tid}_\texttt{c}$ to overflow!

2.5.2 Deadlock and suitable values for `MAX_SZ` and `MAX_TIMEOUT`

A suitable maximum for `MAX_SZ` in the client-program should be determined using testing. Closely related is the value for `MAX_TIMEOUT` in the client-program: if `MAX_SZ` is chosen large (i.e. 30 seconds), `MAX_TIMEOUT` should also be set rather large (i.e. 30 seconds). There is however a problem associated with setting `MAX_TIMEOUT` to such high values which we discuss now.

Suppose two different clients c_0 and c_1 both want to perform a transaction simultaneously. Client c_0 chooses the services in the set S_0 and client c_1 chooses the services in the set S_1. Let $S_0 \cap S_1 = \{s_0, s_1\}$. Obviously, only one of the requests sent to s_0 by c_0 and c_1 can be received first by s_0. The same holds for the requests sent by c_0 and c_1 to s_1.

Consider the scenario where s_0 chooses the request sent by c_1 first and s_1 chooses the request sent by c_0 first. Because s_1 is performing a transaction with c_0, it sends a reply to c_0 and waits for the commit decision from c_0 to arrive. Likewise, because s_0 is performing a transaction with c_1, it sends a reply to c_0 and waits for the commit decision from c_0 to arrive. After c_0 has sent all requests, it tries to receive all replies. It cannot receive a reply from s_0 however, because s_0 is in a transaction with c_1 and itself waiting. Likewise, after c_1 has sent all requests, it tries to receive all replies. It cannot receive a reply from s_1 however, because s_1 is in a transaction with c_0 and itself waiting.

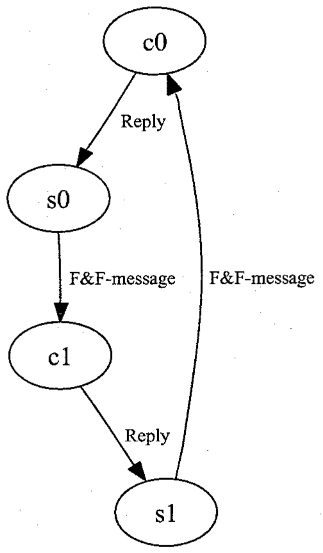

Figure 2.5: Wait-for graph

Figure 2.5 is a wait-for graph [SKS98] of the scenario described above: the vertices in the graph represent the clients and services. The outgoing arc from for example c_0 to s_1 labeled 'Reply' indicates that c_0 is waiting for a reply from s_1. The graph contains a cycle, meaning that there is danger of deadlock for the machines represented by the vertices of the cycle.

However, the IFSA-bus returns a timeout to c_0 and/or c_1 such that eventually all components in the graph resume normal operation (the protocol has been proved to be deadlock-free in section 4.6.3). The point however is to observe that the machines are stuck for up to `MAX_TIMEOUT` time in the given scenario. Hence the following guidelines should be taken into consideration if this is deemed undesirable;

- *Reduce the chances of a client c performing a transaction $\mathcal{T}$ and a different client c' performing a transaction $\mathcal{T}'$ simultaneously, such that $\mathcal{T}$ and $\mathcal{T}$' have more than one service in common*

- *If the former guideline cannot be followed: set* `MAX_TIMEOUT` *rather small and consequently, set* `MAX_SZ` *rather small.*

Again, real-world testing should be performed in order to determine what exactly are 'rather small' values for `MAX_TIMEOUT` and `MAX_SZ`. Note that setting `MAX_TIMEOUT` *too* small would lead all replies to timeout, whence all transactions would abort in which case requirement 3 of the problem statement on page 17 would not be met. A sensible value of `MAX_TIMEOUT` together with the two guidelines given above ensure that the n-R/R protocol does indeed meet requirement 3 in all probability.

In section 1.2 it was mentioned that one could simulate R/R-messaging by means of F&F-messaging with some additional identifiers. The advantage of doing this would be to provide for reliable R/R-messaging. It was noted that simply replacing R/R-messaging by such F&F-messaging could introduce deadlock. For the n-R/R protocol with $1 < n$, this is the case. Indeed: the deadlock scenario depicted in figure 2.5 is not be resolved by the bus if c_0 and c_1 forever wait for 'F&F-replies' from s_1 and s_0 respectively. Of course one could devise a similar timeout mechanism as present in R/R-messaging, but this would undo all advantages of simulating R/R-messaging by means of F&F-messaging!

2.5.3 Efficiency

In this section we discuss the efficiency of the n-R/R protocol. The protocol requires $3n$ messages to be sent during size-n distributed transactions: n F&F-messages, n requests and n replies.

It was already mentioned in section 1.2 that the overhead associated with F&F-messages is far greater than that of requests and replies. However, it was also argued in chapter 1 that distributed transaction integrity cannot be achieved if we only allow R/R-messaging. Hence the n required F&F-messages seem a small price to pay for achieving distributed transaction integrity. Especially in comparison to the formerly considered solution (see section 1.2) of replacing all R/R-messaging by F&F-messaging: this would require at least $2n$ F&F-messages.

The speed by which messages are transported by the bus is about 1 megabyte per second according to IT-architects at ING. The payload-variables required by our protocol take up less than a kilobyte. Hence the additional payload-variables amount for an increase of the processing time of a single transactions in the range of a few milliseconds.

In summary, we consider the n-R/R protocol to provide a good balance between price and performance whence requirement 4 of the problem statement on page 17 is met by the n-R/R protocol.

2.5.4 Handling Exceptions

In this section we dicuss how to deal with the loss of messages due to system-failures that cause a BAI-method to return with an error (see section A.6). Note that the advise given in this section is not backed up by formal proofs of properties of the transaction-system!

Each BAI-method returns a return-code and corresponding reason-code (see appendix A). These return- and reasoncodes must be inspected after each BAI-method invocation to detect the loss of messages.

Failures in the client-program related to the bus

- Failure connecting to the IFSA-bus (**Init**): Human intervention is required, the client-program must shutdown.
- Failure obtaining a service-slot (**Input**): human intervention may be required. Instead of proceeding to the label **Send Requests**, the client-program must go to the label **Commit/Abort DB-Transaction & Output**, abort the local database transaction (if one was started already in the **Input**-phase) and output a result indicating that a failure occurred to its environment.
- Failure sending a request (**Send Request**) or receiving a reply (**Get Replies**): human intervention may be required. The client must however stick to the protocol with only one exception: the decision in phase **Decide** *must* be to abort the transaction. The client program may only shutdown at the end of phase **Commit/Abort DB-Transaction & Output** but *not* before!
- Failure beginning a unit of work(**Send Commit Decision**) or committing a unit of work (**Commit/Abort DB-Transaction & Output**): human intervention is required immediately because services it is transacting with may be waiting for a commit-decision to arrive! This situation should be treated as if each commit decision failed to be sent (treated below).
- Failure sending a commit decision (**Send Commit Decision**): human intervention is required immediately because services it is transacting with may be waiting for a commit-decision to arrive! The easiest solution is to have the operator of the addressed service insert the message that failed to be sent in the service's F&F-queue.

Failures in the service-program related to the bus

- Failure connecting to the IFSA-bus (**Init**): Human intervention is required, the service-program must shutdown.
- Failure getting a request (**Get Request**): human intervention may be required. Instead of proceeding to the label (**Process Request**), the service program must either return to the label **Get Request** and a start a new transaction, or shutdown.
- Failure sending a reply (**Vote & Send Reply**): human intervention may be required. Note that the client with which the service is transacting eventually receives a timeout for the reply that failed to be sent. The client-program eventually decides to abort the transaction. Hence the service is sent a corresponding commit decision. It may as well abort immediately and either return to the label **Get Request**, or shutdown.
- Failure starting a Unit of Work: human intervention is required immediately. Ultimately, the service must finish the transaction according to its protocol. How this can be achieved depends on the error in the bus. See also the next bullet.

- Failure getting the commit decision (**Get Commit Decision**): human intervention is required immediately for the commit decision that failed to be received may well be the one corresponding to the transaction the service was performing. We briefly sketch a solution to this problem. We let the service stick to its protocol: it must continue its repetition in the phase **Get Commit Decision**. The operator of the service must consult the operator of the client in order to determine the contents of the commit decision and insert a corresponding commit decision in the service's F&F-queue.
- Failure committing the Unit of Work: human intervention may be required. The service must finish the transaction according to the protocol: as long as the commit decision was already received, failing to commit the Unit of Work does not affect the protocol.

2.5.5 Limitations

The protocol has been designed for single-instance, single-threaded services. It is however not difficult to extend the protocol such that it can function with multi-instance and/or multi-threaded services. Nevertheless, we refrain from giving such an extension here because of time-constraints of this project. We provide more directions for further research and development concerning transaction integrity in IFSA in section 5.3.

Chapter 3

Verification of the 1-R/R protocol

In this chapter we formally prove that requirements 1 and 2 of the problem statement on page 17 are met by the 1-R/R protocol. The 1-R/R protocol is a special case of the n-R/R protocol introduced in section 2: it only allows for size-1 transactions between any client/service pair. Chapter 4 extends the results from this chapter to the n-R/R protocol for any positive n. The main reason for first considering the 1-R/R protocol is to bridle the complexity of our proof obligations. Section 3.1 gives an overview of this chapter.

3.1 Overview

In this section we give an overview of this chapter. Connected to this is our choice of formalism, which we discuss first. The following reasons motivated our choice for specification and verification of the protocol as a shared-memory multiprogram in the Guarded Command Language (GCL) [FvG99];

- We study a transaction system with a virtually unbounded number of clients and services. Each client may select any service and the number of transactions is also unbounded in our model. The unbounded number of clients, services and transactions is problematic in formalisms that rely on explicit state-space generation.
- One could model-check the state-space of a transaction system with a bounded number of clients, services and transactions. It is questionable whether or not results from an artificially bounded transaction system would apply to the unbounded transaction system, given the seemingly unbounded state-space of the transaction system under investigation. Hence we opted for the safest route: to formally prove that the n-R/R protocol satisfies requirements 1 and 2 of the problem statement in an unbounded transaction system.

A derivational style of the program texts is often associated with the GCL. Algorithms (or protocols) are often derived incrementally from a required system invariant or postcondition. We also followed this approach to derive the 1-R/R protocol but came to the conclusion that also presenting it as such would only increase the complexity of our presentation due to the large number of invariants and assertions needed, the cause/effect relations between them and the large number of program texts needed in an incremental derivation. We present a fully annotated program instead.

We formalize requirement 1 of the problem statement given on page 17 by means of a system-invariant. In order to prove that the system-invariant holds, we need many assertions and other system-invariants. Assertions in turn, must also be shown to be both;

- *locally correct:* if an assertions is at a control point of, say, a client, we must show that the statement before the assertion indeed establishes the assertion.
- *globally correct:* if an assertions appears in the program text of for example a client, we must show that no atomic statement of other clients or services disturb the assertion.

If assertions are both locally correct and globally correct, they are *partially correct.* With the help of the correct assertions, we can prove all system invariants to be maintained by the transaction system. Even if we can show the system-invariants to hold, this does not mean that always eventually something (useful) happens. This is referred to as the issue of *progress* (requirement 2 of the problem statement on page 17).

The program texts, invariants, assertions and associated proof obligations are treated in separate sections. In section 3.2 we introduce some types and variables used in the subsequent sections. In section 3.3 we model the messaging functionality of the IFSA-bus discussed in section 1.1.2 and appendix A. In section 3.4 we introduce the components of the multi-program.

In section 3.5 we give the main correctness criterion in the form of system-invariants. Assertions are introduced and proved locally correct in section 3.6. Global correctness of the assertions is proved in section 3.7. Correctness of all system-invariants is proved in section 3.8. In section 3.9 we prove that progress is guaranteed in the 1-R/R protocol, which covers requirement 2 of the problem statement.

Throughout this chapter we refer to the protocol described in chapter 2 as the *informal specification* of the protocol and the version described in this chapter as the *formal specification.* Differences between these specifications are explained when appropriate.

We advise readers to keep copies of pages 56, 57, 130, 131 and 133 at hand while reading this chapter in order to avoid a lot of page turning in the sequel (see appendix B).

3.2 Identifiers and Types

We represent all clients by the set C, services are represented by the set S, where C and S are disjoint, infinite sets of natural numbers. The members of the set $A = C \cup S$ are called *addresses.* In the sequel it is assumed that variables $c, c', \ldots$ are members of C, $s, s', \ldots$ of S and $a, a', \ldots$ of A.

In our formal specification we let *both* clients *and* services assign a transaction-id to transactions whereas in the informal specification, this was only done by clients. The service's transaction-id is only used for specification purposes and is neither communicated nor accessible to clients or other services. The transaction-id assigned by the service is called the *service-transaction-id,* that of the client the *client-transaction-id* or simply *the* transaction-id if no confusion is possible.

The $tid + 1^{th}$ transaction of a client is assigned client-transaction-id tid. The interpretation of the service-transaction-id is similar. The variables $ct, ct', \ldots$ (all members of $\mathbb{N}$) are used to refer to client-transaction-id's, the variables $st, st', \ldots$ (all integers at least -1) for those of the services. The use of a service-transaction-id equal to -1 will be explained in section 3.5.1.

Distributed transactions are identified by a triple (c, s, ct) meaning, 'the distributed transaction with client c, service s and *client*-transaction-id ct'.

Requests, replies and F&F-messages alike are represented by the following specification record-type;

$$\textbf{type } Message = \begin{cases} fa: & A \\ ta: & A \\ tid: & \mathbb{N} \\ to: & \mathbb{B} \end{cases}$$

The differences in message-format between the formal and informal specifications have been summarized in table 3.1. The column *formal specification* lists message-fields in the formal specification, a corresponding entry in the column *informal specification* gives the corresponding payload-variable in the informal specification or '-' if none exists.

Message	*formal specification*	*informal specification*
Request	fa	-
	ta	-
	tid	`tid`
	to	-
Reply	fa	-
	ta	-
	tid	-
	to	¬`sv`, return- and reason-codes
F&F-message	fa	-
	ta	-
	tid	`tid`
	to	¬`cd`

Table 3.1: Differences in message format

The fields of the type $Message$ are explained below, along with the differences compared to the message-format in the informal specification;

fa : A: (from address), the address of the sender. This field is not needed in the informal specification as the bus provides for addressing.

ta : A: (to address), the address of the addressee. This field is not needed in the informal specification as the bus provides for addressing.

tid : $\mathbb{N}$: (transaction id), the *client* transaction-id of the transaction this message is sent for, similar to the payload-variable `tid` in the informal specification.

to : $\mathbb{B}$: (time out). We model the reception of a timeout by means of the reception of an actual message in the formal specification: timeouts are indicated by return- and reasoncodes in the informal specification. As discussed in the informal version of the protocol, a service must communicate its vote to commit or abort a distributed transaction by means of the payload-variable `sv` in the phase **Vote & Send Reply** (see section 2.3.2). We do not discriminate between the reception by a client of a vote to abort or a timeout. The effect of either message is the same in the n-R/R protocol: the client must decide to abort the distributed transaction. The fields to is used to indicate a time-out or commit decision, depending on the context;

- If the message is a request from the R/R interface then fa is a service, ta a client and tid the client-transaction-id; the field to is superfluous. We use the convention of setting to equal to $false$ in requests.
- If the message is a reply from the R/R interface, then ta is a client, fa a service and the interpretation of to is as follows:

to	$\equiv$	'the message is a timeout or service fa voted to abort the distributed transaction', and
$\neg to$	$\equiv$	'the message is a reply, service fa voted to commit the distributed transaction'.

In the remainder of this chapter we call a reply with field to equal to $true$ a *timeout*. A reply with field to equal to $false$ is called simply a *reply*.

- In case the message is a F&F-message then fa is a client, ta a service and the interpretation of to is as follows:

  $$to \equiv \text{'client } fa \text{ decided to abort transaction } (fa, ta, tid)\text{', and}$$
  $$\neg to \equiv \text{'client } fa \text{ decided to commit transaction } (fa, ta, tid)\text{'.}$$

  F&F-messages as described above are called *commit decisions* throughout this chapter. Compared to the informal specification, we have that the variable to corresponds to the payload-variable $\neg$cd.

We represent a record instance by means of a list of its members between angular brackets. For example, the request $\langle c, s, 0, false\rangle$ is one from client c, to service s. It has a client-transaction-id equal to zero and a field to equal to false. Note that we also use angular brackets in defining atomicity in the sequel.

We use the following global arrays in our specification;

var tid	:	$Array[A]$	:	$[\mathbb{N})$;
var cd	:	$Array[A][-1..\infty)$	:	$\mathbb{B}$;
var cw	:	$Array[C][\mathbb{N}]$	:	S;
var $ct2st$	:	$Array[C][\mathbb{N}]$	:	$[-1..\infty)$;

Below we give intuitive descriptions and properties of the arrays presented above. Those properties are formalized by means of invariants and/or assertions in the sequel;

- tid (transaction id); $tid[a] = K \equiv$'machine a has completed K transactions'. Note that the second index of cd and the range of $ct2st$ both includes -1: we explain the use of this in section 3.5.1.
- cd (commit decision), for $a \in A, at < tid[a]$;
  - $cd[a, at] \equiv$ 'machine a (either a client or service) decided to commit its $at + 1^{th}$ competed transaction';
  - $\neg cd[a, at] \equiv$ 'machine a decided to abort its $at + 1^{th}$ competed transaction'.
- cw (communicate with), for $ct < tid[c]$; $cw[c, ct] = s \equiv$ 'in its $ct + 1^{th}$ completed transaction, client c transacted with service s'.
- $ct2st$ (client-transaction-id to server transaction id), for $ct < tid[c]$; $ct2st[c, ct] = st \equiv$'in its $ct + 1^{th}$ completed transaction, client c transacted with service s, which was the $st + 1^{th}$ transaction for s'.

The differences between the variables in the formal and informal specifications are given in table 3.2. The first column lists globally declared arrays present in the formal specification (for example: cd). The corresponding entry in the third column gives the corresponding variable in the client component (for the array $cd[c]$, this is cd_c). The corresponding entry in the second column gives the corresponding variable in the service component (for the array $cd[s]$ there is none: '-'). An explanation of table 3.2 is given below;

Formal Specification:	*Informal Specification:*	
	client c	*service s*
$cw[c]$, $ct2st$, $tid[s]$	-	-
$cd[c]$	-	-
$cd[s]$	-	-
$tid[c]$	$\mathtt{tid}_{\mathtt{c}}$	-
-	$\mathtt{sz}_{\mathtt{c}}$	-

Table 3.2: Implementation of globally declared variables

- The variables $cw[c]$ (for clients c), $ct2st$, $tid[s]$ (for services s), are all specification variables needed only for proving properties of the protocol. Hence those variables are not needed in an actual implementation.
- The arrays $cd[c]$ and $cd[s]$ correspond to commits or aborts of database transactions in the informal specification.
- The variable $\mathtt{sz}_{\mathtt{c}}$ is not needed in the formal specification of the 1-R/R protocol because we study only distributed transactions with size 1 in this chapter.

3.3 Formal model of the IFSA application-bus

It is assumed that the reader is familiar with the R/R- and F&F-patterns (see sections 1.1.2 or, A.3.1 and A.3.2). In this section we model message passing as provided by the bus by means of adding messages to- or taking messages from sets of messages. The variables needed to this end are introduced in section 3.3.1. In section 3.3.2 we introduce some language constructs needed in the sequel. Sections 3.3.3 and 3.3.4 give a formal model of the R/R- and F&F-patterns respectively.

3.3.1 Variables

As noted earlier, we model message passing as provided by the bus by means of adding messages to- or taking messages from sets of messages. Each set of messages represents a *message queue* in IFSA terminology. Note that although the term 'queue' is used, no ordering on messages is implied! The arrays needed to represent the message queues are given below;

$$\begin{array}{llllll}
\textbf{var}\ rqi & : & Array[S] & : & \textbf{Set of}\ Message; \\
\textbf{var}\ rpi & : & Array[C] & : & \textbf{Set of}\ Message; \\
\textbf{var}\ ffi & : & Array[S] & : & \textbf{Set of}\ Message; \\
\textbf{var}\ rd & : & Array[S] & : & \mathbb{B};
\end{array}$$

An overview of the differences between the informal and formal specifications is given in table 3.3. We explain table 3.3 below;

- The set $rqi[s]$ represents the *R/R queue* of service s. Typically, only clients add requests to $rqi[s]$ - which amounts to sending requests to service s- although it is conceivable that in some applications services may also send requests to other services. Only service s may take messages from $rqi[s]$ (receive a request): services cannot receive messages intended for other services. Services received requests using the

Formal Specification:	*Informal Specification:*	
	client c	*service s*
$rqi[s]$	ssRR[s′]	$\texttt{conRR}_\texttt{s}$
$rpi[c]$	ssRR[s′]	$\texttt{conRR}_\texttt{s}$
ffi$[s]$	ssFF[s′]	$\texttt{conFF}_\texttt{s}$
$rd[s]$	bus	bus

Table 3.3: Implementation of message-queues

connection conRR in the informal specification. Clients sent a request by means of the service-slot ssRR[s′], $0 \leq \texttt{s}' < \texttt{sz}$, where ssRR[s′] represents a service-slot for service s. Note that s′ was *not* an address in the informal specification whereas s is one in the formal specification!

- The set *rpi*$[c]$ contains only replies or timeouts addressed to client c. Hence only client c may take messages from it. A service s may add replies to *rpi*$[c]$, but only after taking a corresponding request from *rqi*$[s]$. It is clarified in the sequel that also client c itself may add timeouts to *rpi*$[c]$. Services sent replies using the connection conRR in the informal specification. Clients received replies or timeouts by means of the service-slot ssRR[s′], $0 \leq \texttt{s}' < \texttt{sz}$, where ssRR[s′] represents a service-slot for service s.

- The set *ffi*$[s]$ represents the F&F queue of service s: any message therein is a F&F-message. Each service s may take messages from *ffi*$[s]$, other machines may add messages to it. In the n-R/R protocol, only clients send messages to *ffi*$[s]$, namely the commit decisions. Services received commit-decisions using the connection conFF in the informal specification. Clients sent a commit-decision by means of the service-slot ssFF$[s']$, $0 \leq s' < \texttt{sz}$, where ssRR$[s']$ represents a service-slot for service s.

- rd (<u>r</u>ea<u>d</u>y); $(rd[s] \wedge rqi[s] \equiv \emptyset)$ ≡‘service s is ready to receive a request’. We use the array rd to mimic the behavior of the R/R-pattern offered by the bus: this is clarified in section 3.3.3.

3.3.2 Language constructs

In this section we introduce some language constructs and notational conventions that are needed in the sequel. In our specification we take as atomic statements;

- every single statement;
- every program fragment enclosed between angular brackets '$\langle$' and '$\rangle$'.
- every guard evaluation not enclosed between angular brackets.

We also use statements of the following form:

var $v_0 : T_0$;

$\vdots$

var $v_k : T_k$;

$v_i : P(v_0, \ldots, v_i, \ldots, v_k)$;

$\{\ P(v_0, \ldots, v_i, \ldots, v_k)\ \}$

Here v_i is some variable, P some predicate over program variables. The interpretation of the statement above is 'assign a value to v_i such that $P(v_0, \ldots, v_i, \ldots, v_k)$ holds'. Note that the predicate $P(v_0, \ldots, v_i, \ldots, v_k)$ is only locally correct so far. If multiple values exists such that $P(v_i)$ holds, one is selected non-deterministically. In some cases we explicitly require this choice to be fair. In most cases the predicate P is rather trivial or there exists a BAI-method that establishes it.

We also use **await**-statements of the following form;

```
{ Control point 1 }
⟨ await(P);
  {P}
  atomic body
⟩
{ Control point 2 }
```

Suppose the above statement occurs in client c. The interpretation is that execution of client c remains at control point 1 until P holds. Execution can enter the atomic brackets only if both P holds *and* if it is c's turn to make a step in the interleaving with other components. If this is the case, the atomic body is executed such that it is not interleaved with statements of other clients or services. It is usually the task of other components to establish P.

Sections 3.3.3 and 3.3.4 give a formal specification of the BAI-methods for R/R and F&F communication respectively, in terms of operations on the message-queues introduced in section 3.3.1.

3.3.3 R/R communication

In section 1.1.2 we discussed R/R-messaging. Figure 1.2 on page 14 provided a reduced overview of the three possible scenarios. Figure 3.1 shows how we model R/R-messaging in the formal specification, the numbering of the scenarios corresponds to that of figure 1.2. Scenario 1 is the same in both figures. In scenario 2 of figure 3.1, we knotted together the lost and found messages present in scenario 2 of figure 1.2. Indeed, we model scenario 2 by letting the client send a timeout to itself. The same has been done in scenario 3 of figure 3.1: instead of losing the reply and finding a timeout, we do as if the service reliably sends a timeout. In our model we take care to preserve the property of the R/R-interface that clients cannot distinguish between scenarios 2 and 3, and the property that services cannot distinguish between scenarios 1 and 3.

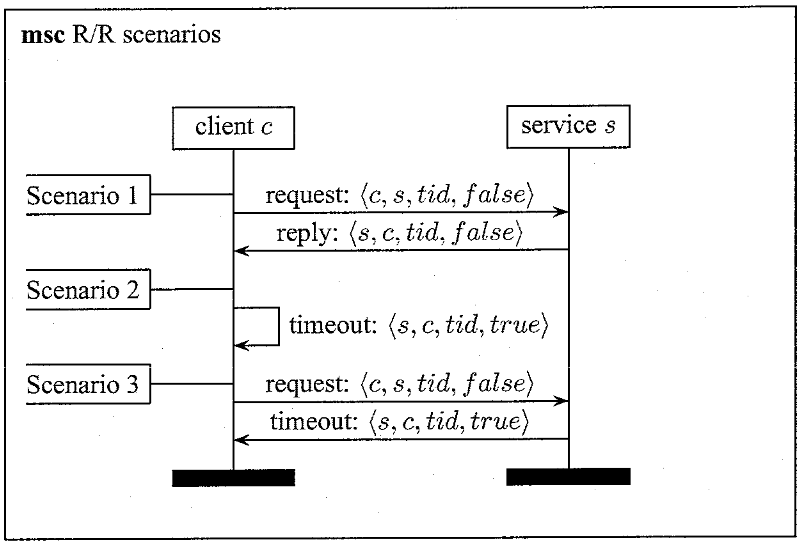
msc R/R scenarios
client c
service s
Scenario 1
request: $\langle c, s, tid, false\rangle$
reply: $\langle s, c, tid, false\rangle$
Scenario 2
timeout: $\langle s, c, tid, true\rangle$
Scenario 3
request: $\langle c, s, tid, false\rangle$
timeout: $\langle s, c, tid, true\rangle$

Figure 3.1: R/R-scenarios in the formal specification

Having explained this, we can proceed by giving GCL-specifications of the operations on the message-queues. We model the sending of requests and replies as follows;

Sending the $tid + 1^{th}$ **request from client** c **to service** s**;**

```
⟨ if   $(rd[s] \wedge rqi[s] = \emptyset) \rightarrow rqi[s] := rqi[s] \cup \{\langle c, s, tid, false\rangle\}$;
   ▯  $\neg(rd[s] \wedge rqi[s] = \emptyset) \rightarrow rpi[c] := rpi[c] \cup \{\langle s, c, tid, true\rangle\}$;
  fi
⟩
```

This corresponds to the BAI-method IFSASend (see section A.3.5). A request gets lost if the service is not 'ready' to receive one. A service is not 'ready' if $\neg rd[s] \vee rqi[s] \neq \emptyset$ holds. In that case the request is not added to the service's request queue but a timeout is added to the clients reply queue instead. The first alternative of the **if**-statement corresponds to scenarios 1 and 3, the second to scenario 2 (all of figure 3.1).

If multiple clients compete in sending a request to the same service, it is not the case that some clients have a higher chance of successfully sending the request (instead of receiving a timeout). We say the choice between multiple clients to be *fair*. This assumption is valid according to bus-architects.

Sending the reply $\langle s, c, tid, false\rangle$ **or timeout** $\langle s, c, tid, true\rangle$ **from** s **to** c**;**

```
⟨ if
   ▯ $true \rightarrow rpi[c] := rpi[c] \cup \{\langle s, c, tid, false\rangle\}$;
   ▯ $true \rightarrow rpi[c] := rpi[c] \cup \{\langle s, c, tid, true\rangle\}$;
  fi
⟩
```

This corresponds to the BAI-method IFSASend (see section A.3.5). A reply may be 'lost' in which case the service *did* receive (and process) the corresponding request and the client receives a timeout instead of a reply. Recall that a vote to abort from a service corresponds to a timeout in the formal specification. The field to of the request is chosen non-deterministically.

The first alternative of the **if**-statement corresponds to scenario 1, the second to scenario 3 (both of figure 3.1).

We model the reception of requests and replies as follows;

Receiving a request by s;

```
$rd[s] \ := \ true;$
$\langle$ await$(rqi[s] \neq \emptyset);$
  $m : m \in rqi[s];$
  $rqi[s],\ rd[s] \ := \ rqi[s]/\{m\},\ false;$
$\rangle$
```

This corresponds to the BAI-method IFSAGetMessage (see section A.5). The first statement indicates that s is ready to receive a request. The second statement corresponds to waiting until a request is available. The third statement chooses a message from $rqi[s]$ and assigns it to the program variable m. The last statement removes the request from the queue and reflects that service s is no longer willing to receive a request.

Receiving the reply m from s in the $ct+1^{th}$ transaction of client c with service s;

```
$\langle$ await$(\langle\exists m' : m' \in rpi[c] \ \wedge \ m'.fa = s \ \wedge \ m'.tid = ct\rangle)$
  $m : m \in rpi[c] \ \wedge \ m.fa = s \ \wedge \ m.tid = ct;$
  $rpi[c] \ := \ rqi[c]/\{m\};$
$\rangle$
```

This models the behavior of the IFSAGetReply method (see section A.4.2) where a specific reply to a request (passed as a parameter) can be received. In our model this reply is identified by its transaction-id and sender.

In order to preserve the property of the R/R-interface that clients cannot distinguish between scenarios 2 and 3, we disallow the inspection of any variables in the client component that would reveal or record this choice outside of the send and receive actions: rd, rqi and rpi. Likewise, we disallow the inspection of rqi and rpi by services outside of the send/receive actions in order to preserve the property that services cannot distinguish between scenarios 1 and 3.

Note that the ta-field in the type $Message$ may seem superfluous given the send and receive operations introduced above. It can however be advantageous in modeling the use of using units of work. This is explained in section 3.3.4.

3.3.4 F&F communication

The F&F channel is assumed to be reliable even if section A.6 mentions cases in which it is not. According to bus architects those cases are restricted to extreme disasters (during which the world itself may perish) or programming errors. They approve of the model of F&F messaging being reliable.

Sending and receiving F&F-messages is modelled as follows;

Sending the F&F-message m to s;

```
$ffi[s] \ := \ ffi[s] \cup \{m\}$
```

Receiving a F&F-message m by s;

```
⟨ await($ffi[s] \neq \emptyset$);
  $m : m \in ffi[s]$;
  $ffi[s] \ := \ ffi[s]/\{m\}$
⟩
```

Note that if multiple choices are available, a fair choice is made: it cannot be the case that some F&F-message remains in *ffi*[*s*] until the end of time if messages are received repeatedly. This assumption is valid according to bus-architects.

Note that F&F messaging supports units of work (see section A.5). The above retrieval is done without a unit of work. On the other hand, one can argue that the use of a unit of work that contains a single send- or receive-operation is implied by the atomicity brackets.

In general, we can model units of work as follows. Let for each component that can send or receive F&F-messages, FO and FI be local sets of F&F-messages. The first is used to temporarily store outbound messages during a unit of work, the second for inbound messages. Then we can model the operations of a unit of work as follows;

beginning a unit of work;

```
$FO, FI \ := \ \emptyset, \emptyset$;
```

receiving a F&F-message m within a unit work;

```
⟨ await($ffi[s] \neq \emptyset$);
  $m \ : \ m \in ffi[s]$;
  $FI, ffi[s] \ := \ FI \cup \{m\}, ffi[s]/\{m\}$;
⟩
```

sending F&F-messages m within a unit work;

```
$FO \ := \ FO \cup \{m\}$;
```

committing the unit of work;

```
⟨ do $FO \neq \emptyset \ \rightarrow$
      |[ var $m : Message$;
        $m \ : \ m \in FO$;
        $ffi[m.ta] \ := \ ffi[m.ta] \cup \{m\}$;
        $FO \ := \ FO/\{m\}$;
      ]|
  od
⟩
```

rolling back a unit of work;

```
$ffi[s] \ := \ ffi[s] \cup FI$;
```

The formalization given above is certainly not unique. Depending on the protocol that is to be developed and the formalism to be used, other formal representations of the unit of work mechanism may be more convenient. It is important to understand that the effects of F&F-messaging within a unit of work remain invisible to all other clients and services until the unit of work is committed.

Hence if in some protocol, units of work must always be committed and *only* send actions take place in it, then another way to model those units of work is the use of atomicity brackets surrounding the normal send operations of F&F-messaging. In fact, we choose and explain this modeling in the sequel.

Beginning a unit of work translates to '$\langle$', committing translates to '$\rangle$';

$\langle$ 'atomic body of the unit of work: perform F&F- send operations here' $\rangle$

When mixing send operations of F&F-messaging in the atomic body of a unit of work with other statements, an argument must be provided that validates this mixture. It is dangerous to add R/R-messaging to the atomic body. Even if one could prove properties of such program texts, one would have proved properties of an invalid model as units of work have no effect on R/R-messaging.

If F&F-messages are *received* within the atomic body of a unit of work, danger of deadlock may have been introduced in the formal model that is not present in reality. We illustrate this by means of an example multi-program given below. It simulates two clients sending a F&F-message to a single service and the service receiving the two messages within a unit of work.

```
$|[$ **var** $ffi[s]$;

  **component** $Client(c : C) = |[ ffi[s] := ffi[s] \cup \{\langle c, 0, 0, false\rangle\};\ ]|$

  **component** $Service(s : S) =$
  $|[$ **var** $m_0, m_1 : Message$;

    $\langle$ **await**$(ffi[s] \neq \emptyset)$;
     $m_0 : m_0 \in ffi[s]$;
     $ffi[s] := ffi[s]/\{m_0\}$;

     **await**$(ffi[s] \neq \emptyset)$;
     $m_1 : m_1 \in ffi[s]$;
     $ffi[s] := ffi[s]/\{m_1\}$;
    $\rangle$
  $]|$

  $ffi[s] := \emptyset$;
  $Service(0) \parallel Client(1) \parallel Client(3)$;
$]|$
```

We give a rather informal treatment of the deadlock scenario. In the multi-program given above, the service deadlocks on its second await statement if it enters its unit of work before before both clients sent their message. In this scenario, one client deadlocks also: the one that did not yet send a message to the service. Indeed, by deadlock of the service within its atomic body, execution never reaches a control point upon which this client can perform its send operation.

In the real world however, the service cannot deadlock because the clients cannot deadlock, hence the model is invalid. Indeed, either client completes its send operation whence the service can always eventually receive the two messages within its unit of work. In the sequel we allow of all send or receive operations *only* F&F-send operations in the atomic body of a unit of work.

The above example also illustrates some of the problems associated with the use of atomicity brackets in general. In the sequel we use them with care: we shall always provide arguments that validate their use.

3.4 Specification

In this section we present all components: clients (section 3.4.2) and services (section 3.4.3). The main program is presented in section 3.4.4. In section 3.4.5 we provide brief arguments that validate the placing of the atomicity brackets in the program texts given in sections 3.4.2, 3.4.3 and 3.4.4. Section 3.4.1 provides and introduction.

3.4.1 Introduction

This introduction serves the following purposes;

- introduction of ghost-variables (needed for proving global correctness) in section 3.4.1.1.
- labeling of assertions and statements present in the components given in sections 3.4.2, 3.4.3 and 3.4.4 in section 3.4.1.2.
- explanation of the commit decision taken by the client in section 3.4.1.3.

3.4.1.1 Ghost-variables

Some ghost variable arrays have been introduced (for the purpose of proving global correctness in the sequel): the arrays RQO, RPO and CD: $C \times \mathbb{N} \to \mathbb{N}$. Their interpretation is as follows;

- $RQO[c, ct]$ ='the number of requests sent by client c in its $ct + 1^{th}$ transaction'
- $RPO[c, ct]$ ='the number of timeouts' + 'the number of replies' delivered to $rpi[c]$ in client c's $ct + 1^{th}$ transaction
- $CD[c, ct]$ ='the number of commit decisions sent by client c in its $ct + 1^{th}$ transaction'

It follows that all these arrays must be initialized by $\overrightarrow{0}$. Also, none of the values of the capital arrays ever exceeds 1 in the 1-R/R protocol. The ghost variables given above are not inspected by any component, they have been introduced for specification purposes only. Hence they are absent in the informal specification.

3.4.1.2 Labeling of assertions, statements and substitutions

We give a placeholder for each assertion occurring in the program texts presented in the sequel. The assertions itself are presented in section 3.6. We use the convention that the statement with pre-assertion labeled i is labeled statement i. Assertions i is denoted by Ai, statement i by Si.

If a statement is the sequential composition of multiple sub-statements, we refer to for example the 2^{nd} sub-statement of statement $S5$ as $S5.2$. Also, we may subscript a statement to denote by which component it is executed. For example, $S5.1_c$ is the first sub-statement of statement 5 of client c. We use a similar subscripts for the assertions of a component. The labeling system is further extended in section 3.6.

3.4.1.3 The client's vote and commit decision

As explained in section 2.2.2, a client is allowed to decide to abort a distributed transaction even if it received a reply from the service stating that it may commit. This is useful in practice for applicative reasons. One could view the whole process of reaching a commit decision as a vote that only ends in a commit if the client and service unanimously vote to do so.

To this end, we let the client register its vote in a local variable cv. After having received the reply or timeout m_2, say, from the service, the commit decision corresponds to $\neg m_2.to \wedge cv$.

3.4.2 Clients

The client component is given below. The local variable m_2 is used for receiving a reply or timeout, the local variable s is used to abbreviate $cw[c, tid[c]]$. For sending the request or commit decision we do not need a variable. We let the client select a service s non-deterministically (in an implementation a more founded choice would be made). Statement 5 selects a service s and registers this choice in $cw[c, tid[c]]$, statement 6 sends a request to s. Statement 7 receives the reply (or timeout) corresponding to the request sent in statement 6. Also, the client's vote whether or not to commit is recorded in cv. Statement 8 makes the commit decision. This commit decision is recorded in cd and sent to service s.

```
component $Client(c : C) =$
|[
  var $m_2$  :  $Message;$
  var $s$    :  $S;$
  var $cv$   :  $\mathbb{B};$

 do $true \rightarrow$
    { Assertion 5 }
    ⟨ $s : true;$
      $cw[c, tid[c]] := s;$
    ⟩

    { Assertion 6 }
    ⟨ if  $(rd[s] \wedge rqi[s] = \emptyset) \rightarrow rqi[s],\ RQO[c, tid[c]] :=$
              $rqi[s] \cup \{\langle c, s, tid[c],\ false\rangle\},\ RQO[c, tid[c]] + 1;$
      ▯  $\neg(rd[s] \wedge rqi[s] = \emptyset) \rightarrow rpi[c],\ RQO[c, tid[c]],\ RPO[c, tid[c]] :=$
              $rpi[c] \cup \{\langle s, c, tid[c], true\rangle\},\ RQO[c, tid[c]] + 1,\ RPO[c, tid[c]] + 1;$
      fi
    ⟩

    { Assertion 7 }
    ⟨ await$(\langle \exists m : m \in rpi[c] : m.fa = s \wedge m.tid = tid[c]\rangle);$
      $m_2 : m_2 \in rpi[c] \wedge m_2.fa = s \wedge m_2.tid = tid[c];$
      $rpi[c] := rpi[c]/\{m_2\};$
      $cv : true;$
    ⟩

    { Assertion 8 }
    ⟨ $CD[c, tid[c]],\ cd[c, tid[c]] := CD[c, tid[c]] + 1,\ \neg m_2.to \wedge cv;$
      $ffi[s] := ffi[s] \cup \{\langle c, s, tid[c], cd[c, tid[c]]\rangle\};$
      $tid[c] := tid[c] + 1;$
    ⟩

 od
]|
```

3.4.3 Services

The service component is given below. The variables m_1 and m_3 are used for receiving the request and commit decision in a transaction respectively, we need no variable for sending the reply. In order to improve readability, the service component has local variables c and ct, used for the client and client-transaction-id respectively. As the service may receive requests from any client, the corresponding condition it waits for is simply that some request is available. The situation is similar when waiting for commit decisions: commit decisions must be received and removed until the commit decision of the active transaction has been received. Statement 13 receives a request and registers which service-transaction-id corresponds to the client-transaction-id of the request, statement 14 sends a reply or timeout, statement 16 receives the commit decision and statement 18 records the commit decision.

```
**component** $Service(s : S) =$
$[\![$ **var** $m_1, m_3 : Message;$
  **var** $c : C$
  **var** $ct : \mathbb{N};$

  **do** $true \rightarrow$

      { Assertion 13 }
      $\langle$ **await**$(rqi[s] \neq \emptyset);$
        $m_1 : m_1 \in rqi[s];$
        $c, ct := m_1.fa, m_1.tid;$
        $rqi[s], ct2st[c, ct], rd[s] := rqi[s]/\{m_1\}, tid[s], false;$
      $\rangle$

      { Assertion 14 }
      $\langle$ **if** $true \rightarrow rpi[c], RPO[c, ct] := rpi[c] \cup \langle s, c, ct, false \rangle, RPO[c, ct] + 1;$
        $[\!]$ $true \rightarrow rpi[c], RPO[c, ct] := rpi[c] \cup \langle s, c, ct, true \rangle, RPO[c, ct] + 1;$
        **fi**
      $\rangle$

      **repeat**
              { Assertion 16 }
              $\langle$ **await**$(ffi[s] \neq \emptyset);$
                $m_3 : m_3 \in ffi[s];$
                $ffi[s] := ffi[s]/\{m_3\};$
              $\rangle$
              { Assertion 17 }
      **until** $(m_3.fa = c \wedge m_3.ta = s \wedge m_3.tid = ct)$

      { Assertion 18 }
      $cd[s, tid[s]], tid[s], rd[s] := \neg m_3.to, tid[s] + 1, true;$

  **od**
$]\!]$
```

3.4.4 Main program

The main program is given below;

```
|[ type $Message$ = record |[ $fa, ta : A,\ tid : \mathbb{N};\ to : \mathbb{B}$ ]|

   const $C$ = $\{2n | n \in \mathbb{N}\}$
   const $S$ = $\{2n+1 | n \in \mathbb{N}\}$
   const $A$ = $C \cup S$;

   var $tid$           : $Array[A]$                : $\mathbb{N}$;
   var $cd$            : $Array[A][-1..\infty)$    : $\mathbb{B}$;
   var $cw$            : $Array[C][\mathbb{N}]$    : $S$;
   var $ct2st$         : $Array[C][\mathbb{N}]$    : $[-1..\infty)$;

   var $rqi$           : $Array[S]$                : Set of $Message$;
   var $rpi$           : $Array[C]$                : Set of $Message$;
   var $ffi$           : $Array[S]$                : Set of $Message$;
   var $rd$            : $Array[S]$                : $\mathbb{B}$;

   var $CD, RQO, RPO$  : $Array[C][\mathbb{N}]$    : $\mathbb{N}$;

  { Assertion 0 }
  ⟨ $tid := \overrightarrow{0}$;
    $cd := \overrightarrow{false}$;
    $rd := \overrightarrow{true}$;
    $ct2st := \overrightarrow{-1}$;
    $rqi, rpi, ffi := \overrightarrow{\emptyset}, \overrightarrow{\emptyset}, \overrightarrow{\emptyset}$;
    $CD, RQO, RPO := \overrightarrow{0}, \overrightarrow{0}, \overrightarrow{0}$;
  ⟩

  { Assertion 1 }
  $(\| c : c \in C : Client(c))\ \ \|\ \ (\| s : s \in S : Service(s))$

]|
```

3.4.5 Atomicity

In this section we provide brief arguments that validate the placing of the atomicity brackets in the components given in sections 3.4.2, 3.4.3 and 3.4.4. To this end we need the following definitions taken from [FvG99];

local variable: a variable is a local variable of a component if it can only be read or modified by the component itself.

private variable: a variable is a private variable of a component if it can only be modified by the component itself.

shared variable: a variable is a shared variable if two or more components may modify it.

All local and private variables along with their owners are given in table 3.4.

Local variable	*owner*
$tid[a]$, $cd[a]$	component a
$cw[c]$, $CD[c]$, $RQO[c]$	client c
m_{2c}, s_c, cv_c	client c
m_{1s}, m_{3s}, c_s, ct_s	service s
Private variable	*owner*
$rd[s]$	service s

Table 3.4: Local- and private variables and their owners

Table 3.5 lists all shared variables and the components that can modify them.

Shared variable	*Shared by*
$rqi[s]$,$ffi[s]$	all clients and service s.
$rpi[c]$, $RPO[c]$	all services and client c.
$ct2st[C]$	all services.

Table 3.5: Shared variables and their modifiers

Having explained this, we can return to the placing of the atomicity brackets. The use of atomicity brackets '$\langle$' and '$\rangle$' has two reasons;

- to reduce the proof burden: each control point needs an assertion and of each assertion we need to proof global correctness. In summary, the number of proofs is $\mathcal{O}(cp^2)$, where cp equals the number of control points. By placing angular brackets around statements we eliminate all control points between the statements within the angular brackets. We may still put assertions at the eliminated control points, but we do not need to prove global correctness of such assertions.
- to mimic complex real-world behavior: in section 3.3 we formalized some BAI-methods. Most of those methods cannot be given by means of single statements in the specification language used.

Below we provide an explanation of all atomicity brackets in the components given in sections 3.4.2, 3.4.3 and 3.4.4 respectively. Because no component inspects any of the ghost-variables RQO, RPO and CD, we silently allow statements that assign values to the ghost-variables between angular brackets below.

Atomicity brackets in the client-component: Statement 5 selects a service s and registers this choice in $cw[c, tid[c]]$, statement 6 sends a request to s, statement 7 receives the reply (or timeout) corresponding to the request sent in statement 6. Also, the client's vote whether or not to commit is recorded in cv. Statement 8 makes the commit decision. This commit decision is recorded in cd and sent to s. Hence $S6$, $S7$ and $S8$ are BAI-methods as explained in section 3.3, with as only additions,

- the client's vote to commit or abort($S7.4$): the variable cv is a local variable. Placing this choice after the angular bracket that closes $S7$ cannot influence the behavior of other components. We only loose some interleavings, aside from that, c does not behave any different either.
- the assignments to $tid[c]$ and $cd[c, tid[c]]$ in $S8$. We can implement this by using a unit of work of a single message (see sections 3.3.4 and A.5) of which the begin and end are marked by the angular brackets of statement $S8$. As $tid[c]$ and $cd[c]$ are local variables of client c, placing these assignment after the angular bracket that closes $S8$ cannot influence the behavior of other components. We only loose some interleavings, aside from that, c does not behave any different either.

A similar argument can be given for the assignments in $S5$: variables occurring in it are local.

In the informal specification, clients committed or aborted their database transactions after having sent their commit decision to the service(s). In the formal specification this order has been reversed, which is justified by the fact that $cd[c]$ is a local variable of client c.

Atomicity brackets in the service-component: Statement 13 receives a request and registers which service-transaction-id corresponds to the client-transaction-id of the request. Statement 14 sends a reply or timeout, statement 16 receives a commit decision and statement 18 records the commit decision. Hence $S13$, $S14$ and $S16$ are BAI-methods as explained in section 3.3, with as only addition, the assignments to c, ct and $ct2st$ in $S13$ and the assignment to $cd[s, tid[s]]$ in $S18$. As the reader may check, variables c and ct are superfluous local variables: by substituting c with $m_1.fa$ and ct with $m_1.tid$, the assignments may be removed from the program text. The variables c and ct were only introduced to increase readability.

As for the assignment to $ct2st[c_s, ct_s]$ in $S13$: there is no statement in any of the components that inspects the array $ct2st$. Hence here also we have that the difference of placing this assignment outside the angular brackets of $S13$ cannot influence the behavior of other components.

As for the assignment to $cd[s, tid[s]]$ in $S18$: $cd[s]$ and $tid[s]$ are both local variables of service s. Hence here also we have that the difference of placing these assignments after the angular brackets of $S18$ cannot influence the behavior of other components.

Atomicity brackets surrounding $S0$: There is no parallelism before statement S_0 has terminated: the atomicity brackets have been placed only to reduce the number of assertions.

3.5 Partial correctness criteria

In this section we formalize requirement 1 of the problem statement on page 17. We also introduce other invariants and auxiliary functions needed to show that requirement 1 is met by the 1-R/R protocol. In section 3.5.1 we formalize requirement 1, section 3.5.2 introduces the other invariants and auxiliary functions needed.

3.5.1 Formalization of requirement 1

In this section we formalize requirement 1 of the problem statement on page 17 by means of the system invariant I_0. To this end we need the boolean function $tra(c, s, ct, st)$ for $c \in C$, $s \in S$, $ct \in \mathbb{N}$ and $st \in [-1 \ldots \infty)$;

$$tra(c,s,ct,st) \equiv (cw[c,ct] = s \ \wedge \ ct2st[c,ct] = st \ \wedge \ 0 \leq ct < tid[c] \ \wedge \ -1 \leq st < tid[s])$$

An informal interpretation of $tra(c, s, ct, st)$ is: $tra(c, s, ct, st) \equiv$ 'in its $ct + 1^{th}$ transaction, client c transacted with service s, which was the $st + 1^{th}$ transaction of s, and distributed transaction (c, s, ct) terminated'.

Note that $tra(c, s, ct, -1)$ can also hold: every transaction of which the request is lost - Scenario 2 of figure 3.1- is modeled to have a service-transaction-id of -1. In such transactions, client c chooses the 2^{nd} alternative upon execution of $S6$. The service s can never decide to commit the transaction with service-transaction-id -1 because $tid[s]$ is initially 0 and cannot decrease over time. Hence those transactions always 'abort'. This is known as a strategy that defaults to aborting transactions [KLS90]. Connected to this are the modeling choices to make cd rectangular (but not square) and extending the range of $ct2st$ to include -1. A consequence of this is that $ct2st$ is not an injective map from client- to service-transaction-id's.

For all transactions for which tra holds we require that the commit decisions of c and s are equal, which is expressed by $I_0(c, s)$ given below. Finally, I_0 expresses that for each client and service pair (c, s), $I_0(c, s)$ holds (requirement 1 of the problem statement), the main partial correctness criterion;

$$\begin{array}{lll} I_0 & : & \langle \forall c, s \ : \ c \in C \ \wedge \ s \in S \ : \ I_0(c,s) \rangle \\ I_0(c,s) & : & \langle \forall ct, st \ : \ tra(c,s,ct,st) \ : \ cd[c,ct] \equiv cd[s,st] \rangle \text{, for } c \in C, \ s \in S. \end{array}$$

In the sequel we show that I_0 is maintained by the main program given in section 3.4.4. In doing so we need quite a lot of invariants and assertions. The invariants are introduced in section 3.5.2. The assertions needed are given in section 3.6 along with compact proofs of their local correctness. Global correctness of the assertions is proved in section 3.7. The validity of the invariants is discussed in section 3.8.

3.5.2 Other invariants and auxiliary functions

In this section we define some auxiliary functions and and system- and repetition-invariants needed to prove that I_0 holds in the sequel. The auxiliary functions are introduced in section 3.5.2.1, the system-invariants are introduced in section 3.5.2.2 and the repetition-invariants are introduced in section 3.5.2.3

3.5.2.1 Auxiliary functions

We define the following auxiliary functions for $c \in C$, $s \in S$ and $ct \in \mathbb{N}$;

$$\begin{array}{ll} srqi(c,s,ct) & = \{m \mid m \in rqi[s] \ \wedge \ m.fa = c \ \wedge \ m.tid = ct\} \\ srpi(c,s,ct) & = \{m \mid m \in rpi[c] \ \wedge \ m.fa = s \ \wedge \ m.tid = ct\} \\ sffi(c,s,ct) & = \{m \mid m \in ffi[s] \ \wedge \ m.fa = c \ \wedge \ m.tid = ct\} \\ gv(c,ct) & = (\ RQO[c,ct], \ RPO[c,ct], \ CD[c,ct]\ ) \end{array}$$

An intuitive explanation is given below;

- $srqi(c, s, ct)$: the smallest subset of $rqi[s]$ that contains all requests of the $ct + 1^{th}$ transaction of client c.
- $srpi(c, s, ct)$: the smallest subset of $rpi[c]$ that contains all replies of the $ct + 1^{th}$ transaction of client c.
- $sffi(c, s, ct)$: the smallest subset of $ffi[s]$ that contains all commit decisions of the $ct + 1^{th}$ transaction of client c.
- $gv(c, ct)$: the ghost variable array containing the ghost variables introduced in section 3.4 of the $ct + 1^{th}$ transaction of client c.

3.5.2.2 System invariants

In the invariants below, we introduce some superfluous dummy variables in order to increase readability, a drawback of this approach is of course a more complex domain description in the quantifications. In particular, the dummy variable ct for client-transaction-id is superfluous in predicates $prqi_1$,$prpi_1$, and $pffi_1$. The dummy variable c for client is superfluous in predicates $prqi_1$ and $pffi_1$ and s for service is superfluous in predicate $prpi_1$.

$$\begin{array}{ll}
prqi_0 : & \langle \forall m, s \;:\; s \in S \wedge m \in rqi[s] \;:\; m.fa \in C \wedge m.ta = s \rangle \\
prqi_1 : & \langle \forall m, s, c, ct \;:\; s \in S \wedge m \in rqi[s] \wedge c = m.fa \wedge ct = m.tid \;: \\
 & \quad gv(c, ct) = (1, 0, 0) \wedge cw[c, ct] = s \\
 & \rangle \\
prqi_2 : & \langle \forall s \;:\; s \in S \;:\; |rqi[s]| \leq 1 \rangle
\end{array}$$

$$\begin{array}{ll}
prpi_0 : & \langle \forall m, c \;:\; c \in C \wedge m \in rpi[c] \;:\; m.fa \in S \wedge m.ta = c \rangle \\
prpi_1 : & \langle \forall m, c, s, ct \;:\; c \in C \wedge m \in rpi[c] \wedge s = m.fa \wedge ct = m.tid \;: \\
 & \quad gv(c, ct) = (1, 1, 0) \wedge (tid[s] = ct2st[c, ct] \vee m.to) \wedge srqi(c, s, ct) = \emptyset \\
 & \rangle
\end{array}$$

$$\begin{array}{ll}
pffi_0 : & \langle \forall m, s \;:\; s \in S \wedge m \in ffi[s] \;:\; m.fa \in C \wedge m.ta = s \rangle \\
pffi_1 : & \langle \forall m, s, c, ct \;:\; s \in S \wedge m \in ffi[s] \wedge c = m.fa \wedge ct = m.tid \;: \\
 & \quad gv(c, ct) = (1, 1, 1) \wedge cd[c, ct] = \neg m.to \wedge srpi(c, s, ct) = \emptyset \\
 & \rangle
\end{array}$$

$$\begin{array}{ll}
pcw_0 : & \langle \forall c, s, ct \;:\; c \in C \wedge s \in S \wedge ct \in \mathbb{N} \;:\; |srqi(c, s, ct) \cup srpi(c, s, ct)| \leq 1 \rangle
\end{array}$$

$$\begin{array}{ll}
pct2st_0 : & \langle \forall c, s, ct \;:\; c \in C \wedge s \in S \wedge RQO[c, ct] = 0 \;:\; ct2st[c, ct] < tid[s] \rangle
\end{array}$$

$$\begin{array}{ll}
pRQO_1 : & \langle \forall c, s, ct \;:\; c \in C \wedge s \in S \wedge ct \in \mathbb{N} \wedge RQO[c, ct] = 0 \;: \\
 & \quad srqi(c, s, ct) \cup srpi(c, s, ct) = \emptyset \\
 & \rangle \\
pcd_0 : & \langle \forall s, st \;:\; s \in S \wedge tid[s] \leq st \;:\; \neg cd[s, st] \rangle
\end{array}$$

$$\begin{array}{ll}
pCD_1 & \langle \forall c, s \;:\; c \in C \wedge s \in S \wedge CD[c, tid[c]] = 0 \;: \\
 & \quad \neg cd[c, tid[c]] \wedge \neg cd[s, ct2st[c, tid[c]]] \\
 & \rangle
\end{array}$$

$$\begin{array}{ll}
pgv_0 : & \langle \forall c, ct \;:\; RQO[c, ct] = 1 \wedge CD[c, ct] = 0 : \; tid[c] = ct \rangle
\end{array}$$

An intuitive description of the invariants is given below;

- $prqi_0$: expresses that every request in $rqi[s]$ was sent to s by some client.

- $prqi_1$: expresses that for a request $\langle c, s, tid, to\rangle \in rqi[s]$, the request belongs to the $tid + 1^{th}$ transaction of client c in which it chose to transact with s: no reply or commit decision have yet been sent during this transaction.
- $prqi_2$: expresses that the request queue of a service cannot contain more than one request.
- $prpi_0$: expresses that every reply or timeout in $rpi[c]$ was sent to c by some service.
- $prpi_1$: expresses that for a reply or timeout $\langle s, c, tid, to\rangle \in rpi[c]$, that it was sent during the $tid + 1^{th}$ transaction of client c. If the message is not a timeout, the service has not completed its part of the distributed transaction (and is waiting for a commit decision). The request-queue of the service contains no requests of this distributed-transaction.
- pcw_0: expresses that for any distributed transaction (c, s, ct), it cannot be the case that both the request queue of s and the reply queue of c contain a message sent during this transaction. Also, those queues can contain at most one message of transaction (c, s, ct) at any time.
- $pct2st_0$: expresses that if a request has not been sent for a transaction, the corresponding service-transaction-id must be smaller than $tid[s]$.
- $pffi_0$: expresses that every commit decision in $ffi[c]$ was sent to s by some client.
- $pffi_1$: expresses that a commit decision $\langle c, s, tid, to\rangle \in ffi[s]$was sent in the $tid + 1^{th}$ transaction of client c. In addition, one request and one reply (or timeout) were sent during this transaction and the reply-queue of the client does not contain that reply (or timeout).
- $pRQO_1$: expresses that if a client c has sent no request during its $ct + 1^{th}$ transaction, then no such request can exist in a request-queue of any s.
- pcd_0: expresses that a service cannot have decided to commit a local-transaction if the local-transaction did not yet terminate.
- pCD_1: expresses that if a client c has sent no commit decision during its $ct + 1^{th}$ distributed transaction with service s, say, then neither s nor c have decided to commit their corresponding local-transactions.
- pgv_0: expresses that if a request was sent in the $ct + 1^{th}$ transaction of client c but the commit decision was not yet sent, then the transaction has not yet terminated.

3.5.2.3 Repetition invariants

For each client c we introduce the following repetition invariants:

$$\begin{array}{lll}
pRQO_0(c) & : & \langle \forall ct \,:\, ct \in \mathbb{N} \,:\, RQO[c, ct] \leq 1 \,\wedge\, (ct < tid[c] \equiv RQO[c, ct] = 1)\rangle \\
pRPO_0(c) & : & \langle \forall ct \,:\, ct \in \mathbb{N} \,:\, RPO[c, ct] \leq 1 \,\wedge\, (ct < tid[c] \equiv RPO[c, ct] = 1)\rangle \\
pCD_0(c) & : & \langle \forall ct \,:\, ct \in \mathbb{N} \,:\, CD[c, ct] \leq 1 \,\wedge\, (ct < tid[c] \equiv CD[c, ct] = 1)\rangle
\end{array}$$

The invariants express that one request, one reply (or timeout) and one commit decision are sent during each distributed-transaction. For each service s we define the following repetition invariants:

$$\begin{array}{lll}
pct2st_1(s) & : & \langle \forall c, ct \,:\, c \in C \,\wedge\, ct \in \mathbb{N} \,\wedge\, cw[c, ct] = s \,:\, ct2st[c, ct] < tid[s]\rangle \\
prpi_2(s) & : & \langle \forall m, c \,:\, c \in C \,\wedge\, m \in rpi[c] \,\wedge\, m.fa = s \,:\, m.to\rangle
\end{array}$$

$pct2st_1(s)$ expresses that there exist no client transaction with service-transaction-id $tid[s]$. $prpi_2(s)$ expresses that any message in a reply-queue of a client that was sent by s, must be a timeout.

3.6 Assertions: local correctness

In this subsection we give all assertions along with the hints needed to prove their *local* correctness. We treat global correctness of the assertions in section 3.7. The hints are detailed in the sense that the exact conjuncts from assertions and/or invariants needed are provided. First year-style proofs that contain a step by step derivation have been omitted as we trust that any reader interested in this chapter can work out such details with the given hints. Also, the hints could be used in mechanically verifying the proofs. We remind readers to keep copies of pages 56, 57, 130, 131 and 133 at hand in order to avoid a lot of page turning (see appendix B).

As discussed in section 3.4.2, statements and assertions are numbered. In order to keep things readable, we may use multiple assertions at a single control point. For example, assertion $A5$ equals $A5.1 \wedge A5.2$. By $A5.1^{i}$ we mean the i^{th} conjunct of $A5.1$. By $A5.1^{i,j,\cdots}$ we mean the $i^{th}, j^{th}, \ldots$ conjuncts of $A5.1$, by $A6.1, 2, 3$ we mean assertions $A6.1$, $A6.2$ and $A6.3$.

For statements that consist of multiple sub-statements, we number the substitutions occurring in the entire statements by the scheme $\sigma_{SN.1}, \sigma_{SN.2} \ldots$, where SN is the statement number. Note that these substitutions may be composed. For example, by $\sigma_{13.2}$ we mean the *substitution* $c, ct \ := \ m_1.fa,\ m_1.tid$. If statement SN is the sequential composition of K statements, we we use σ_{SN} to denote $(\sigma_{SN.K})(\sigma_{SN.K-1}) \ldots (\sigma_{SN.1})$.

The two guards in the if-statement of $S6$ are labeled B_6^1 and B_6^2. The guard of the repeat-until statement in the service-component is labeled B_{17}.

Some assertions like for example $A5.2$ are a corollary of co-assertions and/or invariants. Although we use the same abbreviation system, it is noted of what co-assertions and/or invariants the assertion is a corollary, if it is one.

3.6.1 Assertions in the main program

3.6.1.1 Assertion 0

Trivial: we use $true$ as predicate over the initial state.

3.6.1.2 Assertion 1

We must show that $A0 \Rightarrow A1(\sigma_0)$ holds.

- **assertion 1.1:** $\langle \forall c : c \in C : pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c) \rangle$
  assertion 1.2: $\langle \forall s : s \in S : prpi_2(s) \ \wedge \ pct2st_1(s) \ \wedge \ rd[s] \rangle$
  local correctness: apply substitutions.

3.6.2 Assertions in the client component

3.6.2.1 Assertion 5

We must show that $\langle \forall c' \ : \ c' \in C \ : \ A1 \ \Rightarrow \ A5_{c'} \rangle$ and $A5(\sigma_8) \ \Leftarrow \ A8$ hold for local correctness of $A5$. Assertion $A5.1$ expresses the repetition invariants.

- **assertion 5.1:** $pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c)$
  local correctness: Initial validity follows from assertion 1.1. Validity after $S8$: use $A8.3^1$ and $A8.1$ and substitution.
- **corollary 5.2:** $gv(c, tid[c]) = (0, 0, 0)$, corollary of assertion 5.1.

3.6.2.2 Assertion 6

We obtain local correctness of $A6$ if we can show;

$$\langle \forall s' : s' \in S : A5 \Rightarrow (\ A6(cw[c, tid[c]] \ := \ s)(s \ := \ s')\ )\rangle$$

- **assertion 6.1:** $pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c)$
  local correctness: use assertion 5.1, and orthogonality of $S5$.
- **assertion 6.2:** $cw[c, tid[c]] = s$
  local correctness: consequence of substitution.
- **corollary 6.3:** $gv(c, tid[c]) = (0, 0, 0)$
  local correctness: corollary of $A6.1$.
- **corollary 6.4:** $srqi(c, s, tid[c]) \cup srpi(c, s, tid[c]) = \emptyset$.
  local correctness: corollary of $A6.1$[1] and $pRQO_1$.

3.6.2.3 Assertion 7

For local correctness we must establish $A6 \Rightarrow [(\ A7(\sigma_{6.1})\ ) \ \wedge \ (\ A7(\sigma_{6.2})\ )]$, where $\sigma_{6.1}$, $\sigma_{6.2}$ correspond to the multiple assignments of the 1st and 2nd alternative in $S6$ respectively.

- **assertion 7.1:** $(\ pRQO_0(c) \ \wedge \ pRPO_0(c)\ )\ (\ \mathbb{N} \ := \ \mathbb{N}/\{tid[c]\}\ ) \ \wedge \ pCD_0(c)$
  local correctness: use assertion 6.1 and orthogonality of $S6$.
- **assertion 7.2:** $cw[c, tid[c]] = s$
  local correctness: use assertion 6.2 and orthogonality of $S6$.
- **assertion 7.3:** $gv(c, tid[c]) = (1, 0, 0) \vee gv(c, tid[c]) = (1, 1, 0)$
  local correctness: follows from

$$(A6.3 \Rightarrow (\ gv(c, tid[c]) = (1, 0, 0)\sigma_{6.1}\ )\ ) \wedge (A6.3 \Rightarrow (gv(c, tid[c]) = (1, 1, 0)\sigma_{6.2})\ ).$$

- **corollary 7.4:** $|srqi(c, s, tid[c]) \cup srpi(c, s, tid[c])| \leq 1$.
  local correctness: corollary of pcw_0.

3.6.2.4 Assertion 8

Note that the variable cv does not occur in any co-assertion of $A8$. Hence we obtain local correctness of $A8$ if we can prove;

$$(\langle \exists m : m \in rpi[c] : m.fa = s \ \wedge \ m.tid = tid[c]\rangle \ \wedge \ A7\ )$$
$$\Rightarrow$$
$$\langle\ \forall m : m \in rpi[c] \ \wedge \ m.fa = s \ \wedge \ m.tid = tid[c] : A8\ (rpi[c] \ := \ rpi[c]/\{m_2\})\ (m_2 \ := \ m)\ \rangle.$$

- **assertion 8.1:**

$$(\ pRQO_0(c) \ \wedge \ pRPO_0(c)\ )\ (\ \mathbb{N} \ := \ \mathbb{N}/\{tid[c]\}\ ) \ \wedge \ pCD_0(c)$$

  local correctness: use assertion 7.1 and orthogonality of $S7$.
- **assertion 8.2:** $cw[c, tid[c]] = s$

local correctness: use assertion 7.2 and orthogonality of $S7$.

- **assertion 8.3:** $gv(c, tid[c]) = (1,1,0) \ \wedge \ (\ tid[s] = ct2st[c, tid[c]] \ \vee \ m_2.to \ )$
  local correctness: use $prpi_1$ (m_2 was taken from $rpi[c]$).
- **assertion 8.4:** $srqi(c, s, tid[c]) = \emptyset \ \wedge \ srpi(c, s, tid[c]) = \emptyset.$
  local correctness: using $A8.2$, $prpi_0$ and $prqi_0$ and $C \cap S = \emptyset$ we have

  $$srqi(c, s, tid[c]) \cap srpi(c, s, tid[c]) = \emptyset.$$

  Using $A7.4$ and the removal of m_2 from $rpi[c]$, 8.4 follows.

3.6.3 Assertions in the service component

3.6.3.1 Assertion 13

Assertions $A13^{1,2}$ expresses the outermost repetition invariants of service s. We must show that $A1 \Rightarrow A13$ and $A18 \Rightarrow (A13(\sigma_{18}))$ hold.

- **assertion 13.1:** $prpi_2(s)$
  local correctness: initially: use $A1.2^1$. After $S18$: use $A18.4$ and orthogonality of $S18$.
- **assertion 13.2:** $pct2st_1(s)$
  local correctness: initially: use $A1.2^2$. After $S18$: apply substitutions, use $A18.1^3$ and $A18.5$.
- **assertion 13.3:** $rd[s]$
  local correctness: initially: use $A1.2^3$. After $S18$: apply substitutions.

3.6.3.2 Assertion 14

We must show;

$$(A13 \ \wedge \ rqi[s] \neq \emptyset) \Rightarrow \langle \forall m : m \in rqi[s] : A14(\sigma_{13.3})(\sigma_{13.2})(m_1 \ := \ m)\rangle$$

- **assertion 14.1:** $gv(c, ct) = (1,0,0) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
  local correctness: apply substitutions; use $prqi_0$, and $prqi_1$.
- **assertion 14.2:** $srqi(c, s, ct) = \emptyset \ \wedge \ srpi(c, s, ct) = \emptyset$
  local correctness: using $prpi_0$, $prqi_0$ and $C \cap S = \emptyset$ we obtain

  $$srqi(c, s, tid[c]) \ \cap \ srpi(c, s, tid[c]) = \emptyset.$$

  Using pcw_0, and the removal of m_1 from $rqi[s]$, $A14.2$ follows.
- **assertion 14.3:** $prpi_2(s)$
  local correctness: use $A13.1$ and orthogonality of $S13$.
- **assertion 14.4:**

$$\langle \forall c', ct' \ : \ c' \in C \wedge ct' \in \mathbb{N} \wedge cw[c', ct'] = s \wedge \neg(c' = c \wedge ct' = ct) \ : \ ct2st[c', ct'] < tid[s]\rangle$$

  local correctness: use $A13.2$ and orthogonality of $S13$.
- **assertion 14.5:** $\neg rd[s]$
  local correctness: apply substitution.

3.6.3.3 Assertion 16

First, we must show that $A14 \Rightarrow (A16(\sigma_{14.1}))$ and $A14 \Rightarrow (A16(\sigma_{14.2}))$ hold, where $\sigma_{14.1}, \sigma_{14.2}$ correspond to the multiple assignments of the 1^{st} and 2^{nd} alternatives in $S14$ respectively. Second, we must show that ($A17 \wedge \neg(m_3.fa = c \wedge m_3.ta = s \wedge m_3.tid = ct)$) $\Rightarrow A16$ holds. Note that for $i: \ 1 \leq i \leq 2$, the assertions $16.i$ and $17.i$ are equal. For $j: \ 3 \leq j \leq 6$, we have that $A16.j$ and $A17.j + 2$ are equal. Hence we only provide hints for the first proof obligation;

- **assertion 16.1:** $(\ gv(c, ct) = (1,1,0) \vee gv(c, ct) = (1,1,1)\ ) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$

  local correctness: apply substitutions; use $A14.1$ and $\vee$-weakening.

- **assertion 16.2:** $srqi(c, s, ct) = \emptyset$

  local correctness: use $A14.2$ and orthogonality of $S14$.

- **assertion 16.3:** $prpi_2(s)(C := C/\{c\})$

  local correctness: use $A14.3$ and orthogonality of $S14$.

- **assertion 16.4:** $\langle \forall m \ : \ m \in rpi[c] \ \wedge \ \neg m.to \ \wedge \ m.fa = s \ : \ m.tid = ct \rangle$

  local correctness: apply substitutions; use $A14.3$.

- **assertion 16.5:**

$$\langle \forall c', ct' \ : \ c' \in C \wedge ct' \in \mathbb{N} \wedge cw[c', ct'] = s \wedge \neg(c' = c \wedge ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$$

  local correctness: use $A14.4$ and orthogonality of $S14$.

- **assertion 16.6:** $\neg rd[s]$

  local correctness: use $A14.5$ and orthogonality of $S14$.

3.6.3.4 Assertion 17

We must show:

$$[A16 \ \wedge \ ffi[s] \neq \emptyset] \ \Rightarrow \ \langle \forall m : m \in ffi[s] : A17(\sigma_{16.2})(m_3 \ := \ m) \rangle.$$

- **assertion 17.1:** $[gv(c, ct) = (1,1,0) \vee gv(c, ct) = (1,1,1)] \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$

  local correctness: use $A16.1$ and orthogonality of $S16$.

- **assertion 17.2:** $srqi(c, s, ct) = \emptyset$

  local correctness: use $A16.2$ and orthogonality of $S16$.

- **assertion 17.3:** $gv(m_3.fa, m_3.tid) = (1,1,1) \ \wedge \ cd[m_3.fa, m_3.tid] = \neg m_3.to \ \wedge \ srpi(m_3.fa, s, m_3.tid) = \emptyset$

  local correctness: use $pffi_0$ to obtain $m_3.ta = s$, use $pffi_1$.

- **assertion 17.4:** $m_3.fa \in C$

  local correctness: use $pffi_0$: m_3 was taken from $ffi[s]$

- **assertion 17.5:** $prpi_2(s)(C := C/\{c\})$

  local correctness: use $A16.3$ and orthogonality of $S16$.

- **assertion 17.6:** $\langle \forall m \ : \ m \in rpi[c] \ \wedge \ \neg m.to \ \wedge \ m.fa = s \ : \ m.tid = ct \rangle$

local correctness: use $A16.4$ and orthogonality of $S16$.

- **assertion 17.7:**

$$\langle \forall c', ct' \ : \ c' \in C \wedge ct' \in \mathbb{N} \wedge cw[c', ct'] = s \wedge \neg(c' = c \wedge ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$$

  local correctness: use $A16.5$ and orthogonality of $S16$.

- **assertion 17.8:** $\neg rd[s]$

  local correctness: initially: use $A16.6$ and orthogonality of $S16$.

3.6.3.5 Assertion 18

We must show that $(\ A17\ \wedge\ B_{17})\ \Rightarrow\ A18$ holds.

- **assertion 18.1:** $gv(c, ct) = (1, 1, 1)\ \wedge\ cw[c, ct] = s\ \wedge\ ct2st[c, ct] = tid[s]$

  local correctness: we obtain 18.1^1 by combining B_{17} and $A17.3^1$. The other conjuncts equal those of 17.1.

- **assertion 18.2:** $srqi(c, s, ct) = \emptyset\ \wedge\ srpi(c, s, ct) = \emptyset$

  local correctness: combine $A17.2$, B_{17} and $A17.3^3$

- **assertion 18.3:** $cd[c, ct] = \neg m_3.to$

  local correctness: combine $A17.3^2$ and B_{17}

- **assertion 18.4:** $prpi_2(s)$

  local correctness: combine B_{17}, $A17.3^3$ and $A17.5$.

- **assertion 18.5:**

$$\langle \forall c', ct' \ : \ c' \in C \wedge ct' \in \mathbb{N} \wedge cw[c', ct'] = s \wedge \neg(c' = c \wedge ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$$

  local correctness: use $A17.7$.

- **assertion 18.6:** $\neg rd[s]$

  local correctness: use $A17.8$.

3.7 Assertions: global correctness

In this section we prove global correctness of the assertions presented in section 3.6.

3.7.1 Introduction

We have the following proof obligations;

No client c' disturbs assertions of a different client c: This proof obligation is discussed in section 3.7.2.

No service s disturbs assertions of client c or vice versa: This proof obligation is discussed in section 3.7.3.

No service s' disturbs assertions of a different service s: This proof obligation is discussed in section 3.7.4.

Note that assertions $A0$ and $A1$ are globally correct as there is no parallelism yet at those control points whence we disregard them in this section. Before delving into the three proof obligations of global correctness, we discuss two types of assertions of which global correctness is trivial to prove;

- Assertions in a component that contain only its own local- and/or private variables: for such assertions we can use the Rule of Private Variables [FvG99] whence their global correctness is guaranteed.
- Assertions that are a corollary of other assertions and/or system-invariants: those are globally correct if we can show the other assertions and/or system-invariants to be so.

Combining the above remarks and tables 3.4 and 3.5, table 3.6 lists all assertions different from $A0$ and $A1$, that are not a corollary of other assertions and/or system invariants and contain at least one variable not owned by the component in which the assertion occurs.

assertion	*co-assertions*
$A5$	$A5.1^2$
$A6$	$A6.1^2$
$A7$	$A7.1^2, A7.3$
$A8$	$A8.1^2, A8.3, A8.4$
$A13$	all except $A13.3$
$A14$	all except $A14.5$
$A16$	all except $A16.6$
$A17$	all except $A17.4, 8$
$A18$	all except $A18.6$

Table 3.6: Remaining assertions (1)

Assertions omitted because they are a corollary have a local correctness hint given in section 3.6 mentioning of what assertions and/or invariants it is a corollary. Assertions omitted because they contain only variables owned by the component in which the assertion occurs, only contain variables listed in table 3.4. In the sequel we only discuss global correctness of the assertions given in table 3.6.

3.7.2 Absence of disturbance between clients

In this subsection we show that a client c' cannot disturb assertions of a different client c: one of the proof obligations for global correctness of the assertions of clients.

Clients have neither assertions containing, nor statements modifying private variables. Neither has client c assertions in which local variables of client c' occur. Hence we must only deal with assertions of client c that contain shared variables in this subsection.

From table 3.5 we obtain that the only shared variables written by c' are $rqi[s_{c'}]$, $rpi[c']$, $RPO[c']$ and $ffi[s_{c'}]$. Note that c has no assertions in which ffi, $RPO[c']$ or $rpi[c']$ occur. This leaves only $rqi[s_{c'}]$ which occurs only in the form $srqi(c, s_c, tid[c])$ in $A8.4_c$. Client c' contains only statements that are orthogonal to this set. Indeed $c' \neq c \Rightarrow srqi(c, s, ct) \cap srqi(c', s', ct') = \emptyset$, for all services s, s' and naturals ct, ct'.

3.7.3 Absence of disturbance between clients and services

In this section we show absence of disturbance between clients and services. We treat similar assertions in a single argument;

- Assertions $A13.1$, $A14.3$, $A16.3, 4$, $A17.5, 6$ and $A18.4$ of service s are all similar in the sense that $rpi[c_s]$ is the only occurring variable that is not owned by service s. Hence for those assertions, the only non-orthogonal statements of a client c are $S6_c$ and $S7_c$. Statement $S6_c$ (2^{nd} alternative) adds only timeouts to $rpi[c]$, $S7_c$ removes only messages from $rpi[c]$: the reader may check that those statements cannot disturb $A13.1$, $A14.3$, $A16.3, 4$, $A17.5, 6$ or $A18.4$.
- Assertions $A13.2$, $A14.4$, $A16.5$, $A17.7$ and $A18.5$ of a service s are all similar. The free variables occurring in these assertions not owned by s are cw in the antecedent and $ct2st$ in the consequent: the only potentially disturbing statement in the client component is $S5$. By $A5.1^1$ and system-invariant $pct2st_0$ the pre-condition to $S5.2$ implies $ct2st[c, tid[c]] < tid[s]$.
- As for $A17.3$, note that a client c can only disturb it if (3.1) defined below holds.

  $$c = m_3.fa \ \wedge \ tid[c] = m_3.tid \tag{3.1}$$

  Assuming (3.1), we have that the ghost-variables provide disjointness;

  $$\left(A17.3^1 \ \wedge \ (A5.2 \ \vee \ A6.3 \ \vee \ A7.3 \vee \ A8.3^1\ )\right) \equiv \ false$$

  Hence we have either disjointness if (3.1) holds or orthogonality otherwise.

Using the above arguments and table 3.6, we need only prove absence of disturbance between clients and services of the assertions given in the first two columns of table 3.7. For the first 4 rows in table 3.7, the fourth column lists which statements in the service components are non-orthogonal to the client-assertion in the first column due to the shared variables listed in the third column: those variables also occur in some of the statements of clients.

Likewise, for the last 4 rows in table 3.7, the fourth column lists which statements in the client components are non-orthogonal (n.o.) to the service-assertion in first column due to the variables listed in the third column.

Next we show that client c and service s can only disturb the assertions listed in table 3.7 if the following *non-orthogonality condition* holds;

$$c_s = c \ \wedge \ ct_s = tid[c] \tag{3.2}$$

assertion	*co-assertions*	*variables*	*n.o. statements*
$A5$	$A5.1^2$	RPO	$S14$
$A6$	$A6.1^2$	RPO	$S14$
$A7$	$A7.1^2, A7.3$	RPO	$S14$
$A8$	$A8.1^2, A8.3, A8.4$	$RPO, tid, ct2st, rqi, rpi$	$S13.3, S14, S18$
$A13$	none	none	none
$A14$	$A14.1, 2$	$RQO, RPO, CD, cw, rqi, rpi$	$S5, S6, S7, S8$
$A16$	$A16.1, 2$	RQO, RPO, CD, cw, rqi	$S5, S6, S8$
$A17$	$A17.1, 2$	RQO, RPO, CD, cw, rqi	$S5, S6, S8$
$A18$	$A18.1, 2, 3$	$RQO, RPO, CD, cd, cw, rqi, rpi$	$S5, S6, S7, S8$

Table 3.7: Remaining assertions (2)

- For the remaining assertions of client c occurring in table 3.7, the only statements of s that can disturb assertions of c are $S13$, $S14$ and $S18$;
  - Of all variables shown in the first four rows of table 3.7, $S14$ contains only assignments to RPO and $rpi[c_s]$. As remarked before, rpi occurs only in the form $srpi(c, s, tid[c])$ in assertions of client c whence $S14$ is only non-orthogonal to this set if (3.2) holds. As for RPO, this array occurs only with first index c in assertions of client c. Hence for $S14_s$ to be non-orthogonal to assertions of client c, $c_s = c$ is a necessary condition. Due to $A14.1^1$ and pgv_0 we have that $c_s = c \Rightarrow ct_s = tid[c]$. Hence *if* $S14_s$ is non-orthogonal to the assertions listed in table 3.7, *then* (3.2) holds at $A14$.
  - We can use the same line of reasoning for $S13.3$ as for $S14$. Of all variables shown in the fourth rows of table 3.7, $S13.3$ contains only assignments to rqi and $ct2st$. These arrays occur only with first index c in assertions of client c. Hence for $S13.3_s$ to be non-orthogonal to assertions of client c, $c_s = c$ is a necessary condition. Note that using $prqi_1$ we obtain $gv(c_s, ct_s) = (1, 0, 0)$ as precondition to $S13.3_s$. Then using pgv_0 (which is not disturbed by $S13.1, 2$) we have that $c_s = c \Rightarrow ct_s = tid[c]$ holds as precondition to $S13.3$. Hence *if* $S13.3_s$ is non-orthogonal to the assertions listed in table 3.7, *then* (3.2) holds as precondition to $S13.3$.
  - $S18_s$ is only non-orthogonal to $A8.3^2_c$, but only if $s_c = s$: assume so. Furthermore, $S18_s$ can only *disturb* $A8.3^2_c$ if in addition, the second disjunct of $A8.3^2_c$ equals $false$ in which case the first disjunct holds: assume so. Combining $A8.2_c$ and the first disjunct of $A8.3^2_c$ we obtain $cw[c, tid[c]] = s \wedge ct2st[c, tid[c]] = tid[s]$. From $A18.5_s$ it follows that for no pair other than c_s, ct_s we have $cw[c_s, ct_s] = s \wedge ct2st[c_s, ct_s] = tid[s]$ whence (3.2) holds.
- The arrays RQO, RPO, CD, cd and cw occur only at index c_s, ct_s in the remaining assertions of service s listed in table 3.7. Also, any client c only manipulates those arrays at index $c, tid[c]$. As for rqi and rpi, note that those occur only in the form of $srqi(c_s, s, ct_s)$ and $srpi(c_s, s, ct_s)$ in assertions of s. These sets can only be affected by statements $S6_{c_s}$ and $S7_{c_s}$, but only if (3.2) holds.

In the remainder of this section we (silently) assume (3.2) to hold. It happens to be so that this non-orthogonality condition greatly reduces the number of global correctness proofs. In the service's assertions the ghost variable vector $gv(c, ct)$ occurs, in those of the clients, $gv(c, tid[c])$ occurs. In table 3.8 the field denoted by row i, column j gives the value of $gv(c, tid[c])$ when client c and service s are at control points i,j, respectively, under the assumption that $ct_s = tid[c] \wedge c_s = c$. For example, row $A7.3$, column $A16.1^1$, has the value $(1, 1, 0)$ that can be inferred from $A7.3 \wedge A16.1 \wedge ct = tid[c]$.

Note that we also added the column $S13.3$: this corresponds to the service s being at 'control-point' $S13.3$ while client c is at either of its control points, assuming (3.2). Although $A13$ lacks a co-assertion expressing the value of $gv(c, ct)$, we can locally derive $gv(c, ct) = (1, 0, 0)$ as precondition to $S13.3$ using $prqi_1$ and orthogonality of statements $S13.1, 2$ to $prqi_1$.

	$S13.3$	$A14.1^1$	$A16.1^1$	$A17.1^1$	$A18.1^1$
$A5.2$	-	-	-	-	-
$A6.3$	-	-	-	-	-
$A7.3$	(1,0,0)	(1,0,0)	(1,1,0)	(1,1,0)	-
$A8.3$	-	-	(1,1,0)	(1,1,0)	-

Table 3.8: Values of $gv(c, tid[c])$ at joint control points, assuming (3.2).

At for example row $A5.2$, column $A18.1^1$, '-' indicates that the conjunction of the co-assertions describing the ghost variables and the assumption $ct = tid[c]$ yields $false$. It follows that neither neither $S18$ can disturb $A5$, nor can $S5$ disturb $A18$: the control points are disjoint under (3.2).

Combining tables 3.7 and 3.8, table 3.9 shows for which assertions we must still prove non-disturbance. How exactly one arrives at table 3.9 is described informally by the following algorithm which should be applied to each row of table 3.7:

1. remove all statements from the fourth column for which table 3.8 shows disjointness.
2. repeat the following until nothing is removed anymore:
   (a) remove any variable from column 3 which does not occur both in any assertion of column 2 and any statement in column 4.
   (b) for each row: remove all assertions from column 2 and statements from column 4 that contain no variables listed in column 3.

Note that any statement or variable removed by this procedure is either disjoint or orthogonal and table 3.7 shrinks during each repetition of the second statement whence the procedure terminates.

assertion	*co-assertions*	*variables*	*non-disjoint and non-orthogonal*
$A5$	none	none	none
$A6$	none	none	none
$A7$	$A7.1^2, A7.3$	RPO	$S14$
$A8$	none	none	none
$A13$	none	none	none
$A14$	$A14.2^2$	rpi	$S7$
$A16$	$A16.1^1$	CD	$S8$
$A17$	$A17.1^1$	CD	$S8$
$A18$	none	none	none

Table 3.9: Remaining assertions (3)

In the remainder of this section we treat non-disturbance of the assertions listed in the second column of table 3.9 by the statements listed in the fourth column, assuming (3.2).

3.7.3.1 Assertion 7

From table 3.9 it follows that we must show that $S14$ cannot disturb $A7.1^2$ or $A7.3$, assuming (3.2).

- $A7.1^2$: the antecedent of $A7.1^2$ becomes $false$ if we substitute the dummy variable ct by $tid[c]$. Hence we have orthogonality.
- $A7.3$: From table 3.8 we obtain;

  $$(\ A7.3 \ \wedge\ A14.1^1 \ \wedge\ ct = tid[c]\ ) \ \Rightarrow\ gv(c, tid[c]) = (1,0,0).$$

  Global correctness of $A7.3$ follows from the following (valid) Hoare-triple

  $$\begin{array}{c}\{\ gv(c, tid[c]) = (1,0,0)\ \} \\ S14 \\ \{\ gv(c, tid[c]) = (1,1,0)\ \}\end{array}$$

  Which implies (by weakening the postcondition):

  $$\begin{array}{c}\{\ A7.3 \ \wedge\ A14.1^1 \ \wedge\ ct = tid[c]\ \} \\ S14 \\ \{\ A7.3\ \}\end{array}$$

3.7.3.2 Assertion 14

From table 3.9 it follows that we must show that $S7$ cannot disturb $A14.2^2$. Assuming (3.2), we have widening of $A14.2^2$. In fact, with somewhat work one can show disjointness under (3.2).

3.7.3.3 Assertions 16 and 17

From table 3.9 it follows we must show that $S8$ cannot disturb $A16.1^1$ and $A17.1^1$. For $A17.1^1$ we have the same proof obligation. As $A17.1^1$ equals $A16.1^1$, we give the proof once for $A16.1^2$.

From table 3.8 we obtain;

$$\left((A16.1^1 \ \vee\ A17.1^1) \ \wedge\ A8.3^1 \ \wedge\ ct = tid[c]\right) \ \Rightarrow\ gv(c, tid[c]) = (1,1,0).$$

Non-disturbance of $A16.1^1$ by $A8$ follows from the following (valid) Hoare-triple:

$$\begin{array}{c}\{\ gv(c, tid[c]) = (1,1,0)\ \} \\ S8 \\ \{\ gv(c, tid[c]) = (1,1,1)\ \}\end{array}$$

Which implies (by weakening the postcondition):

$$\begin{array}{c}\{\ A16.1^1 \ \wedge\ A8.3^1 \ \wedge\ ct = tid[c]\ \} \\ S8 \\ \{\ A16.1^1\ \}\end{array}$$

3.7.4 Absence of disturbance between services

In this section we prove that no service s' disturbs assertions of a different service s: one of the proof obligations for global correctness of the assertions of the services.

We must prove global-correctness of the assertions listed in the second column of table 3.7. For the local- or private and shared variables of component s we refer to tables 3.4 and 3.5 respectively. From table 3.5 we find that $rd[s]$ and $rqi[s]$ do not occur in statements of service s'. Neither do $rd[s']$ and $rqi[s']$ occur in assertions of service s. Hence we may ignore these variables in showing absence of disturbance between services. Also note that assertions of service s contain neither local nor private variables of service s' (see table 3.4).

As the reader may check, $S18_{s'}$ from any service $s' \neq s$ is orthogonal to assertions of service s. Indeed: $cd[s']$ - the only non-local variable changed by $S18_{s'}$ - does not occur in assertions of s for the following reasons;

- It does not occur in $A17.3_s^2$ by $A17.4_s$ and $C \cap S = \emptyset$

$$(s' \neq s \wedge s' \in S \wedge A17.4_s \wedge C \cap S = \emptyset \wedge s' = m_3.fa) \equiv false$$

- From table 3.4 we infer that $cd[c]$ is a local variable of client c, hence $A18.3_s$ cannot be disturbed by $S18_{s'}$. The other assertions of s do not contain cd at all.

The only non-local variable changed by $S16_{s'}$ is $ffi[s']$, which does not occur in any assertions. This leaves only the possibility of $S13_{s'}$ or $S14_{s'}$ disturbing assertions of s. Any disturbance by $S13_{s'}$ would be due only to the assignment to $ct2st$ in $S13.3_{s'}$. Indeed, $rqi[s']$ and $rd[s']$ do not occur in assertions of service s.

Table 3.10 provides an overview of which assertions of service s are potentially non-orthogonal to statements $S13.3_{s'}$ and $S14_{s'}$. In the second column, the variables are shown that occur both in those statements and the assertions in columns $A13.3_s$, $A14_s$ etc. Assertions in these columns are non-orthogonal due to the variable in the first column. The table is constructed such that similar co-assertions are on the same row in the table. In particular, for every pair of co-assertions in the same row we have that the co-assertion that is most to the left in table 3.10 implies the other co-assertion. In the sequel we treat co-assertions row-wise as they appear in table 3.10. We identify the co-assertions in a particular row by the left-most co-assertion occurring in it. For example, the co-assertions in the last row in table 3.10 are called 'row $A14.1_s^1$'.

Statement	*variables*	$A13_s$	$A14_s$	$A16_s$	$A17_s$	$A18_s$
$S13.3_{s'}$	$ct2st[c_{s'}, ct_{s'}]$		$A14.1_s^3$	$A16.1_s^3$	$A17.1_s^3$	$A18.1_s^3$
	$ct2st[c_{s'}, ct_{s'}]$	$A13.2_s$	$A14.4_s$	$A16.5_s$	$A17.7_s$	$A18.5_s$
$S14_{s'}$	$rpi[c_{s'}]$		$A14.2_s^2$			$A18.2_s^2$
	$rpi[c_{s'}]$	$A13.1_s$	$A14.3_s$	$A16.3_s$	$A17.5_s$	$A18.4_s$
	$rpi[c_{s'}]$			$A16.4_s$	$A17.6_s$	
	$rpi[c_{s'}]$				$A17.3_s^3$	
	$RPO[c_{s'}, ct_{s'}]$		$A14.1_s^1$	$A16.1_s^1$	$A17.1_s^1$	$A18.1_s^1$

Table 3.10: Overview of disturbance between services

We start with rows $A17.3_s^3$, $A13.1_s$, $A16.4_s$, and $A13.2_s$;

- row $A17.3_s^3$: $S14_{s'}$ can only disturb it if

$$m_{3_s}.fa = c_{s'} \wedge m_{3_s}.tid = ct_{s'},$$

in which case we have disjointness by

$$(A17.3_s^1 \ \wedge \ A14.1_{s'}^1) \ \equiv false.$$

- rows $A13.1_s$ and $A16.4_s$: $S14_{s'}$ adds a message m to $rpi[c_{s'}]$ for which $m.fa = s'$ holds: the statement is orthogonal to all predicates appearing in rows $A13.1_s$ and $A16.4_s$ by $s' \neq s$. Indeed, those predicates express only properties of messages $m' \in rqi[c_s]$ for which $m'.fa = s$ holds.
- row $A13.2_s$: note the conjunct $cw[c, ct] = s$ appearing in the antecedent of all predicates (where c and ct are dummies but s is not!). By $prqi_1$ we have that the precondition of $S13.3_{s'}$ implies $cw[c_{s'}, ct_{s'}] = s'$. Hence $S13_{s'}$ is orthogonal to the predicates in row $A13.2_s$.

It remains to show service s' cannot disturb the co-assertions in rows $A14.1_s^3$, $A14.2_s^2$ and $A14.1_s^1$. For assertions in those rows, we use a single argument of the following structure:

- First we show that s' can only disturb assertions in those rows if (3.3) defined below is precondition to the disturbing statements.

$$c_{s'} = c_s \wedge ct_s = ct_{s'} \tag{3.3}$$

- Second we show this condition cannot hold, in which case we have shown disjointness under non-orthogonality.

We start with the first bullet. For the assertions in rows $A14.1_s^1$ and $A14.1_s^3$ it is rather obvious that those can only be disturbed by $S13_{s'}$ or $S14_{s'}$ if (3.3) holds. Indeed, the arrays $ct2st$ and RPO occur only with index c_s, ct_s in those assertions. For row $A14.2_s^2$, the clue is to observe that $S14_{s'}$ adds a message m to $rpi[c_{s'}]$ for which we have $m.ta = c_{s'} \ \wedge m.tid = ct_{s'}$. This statement can only affect $srpi(c_s, s, ct_s)$ if (3.3) holds.

As for the second bullet, we must show that it cannot be the case that (3.3) holds. Observe that 14.1_s^2 expresses $cw[c_s, ct_s] = s$. By $prqi_1$ and orthogonality of $S13.1, 2_s$, the precondition of $S13.3_{s'}$ implies $cw[c_{s'}, ct_{s'}] = s'$ (which equals $A14.1_{s'}^2$). Disjointness follows from:

$$(s' \neq s \ \wedge \ cw[c_s, ct_s] = s \ \wedge \ cw[c_{s'}, ct_{s'}] = s' \ \wedge \ c_s = c_{s'} \ \wedge ct_s = ct_{s'}) \equiv false.$$

3.8 Invariants: correctness

In this section we prove the validity of the *system*-invariants introduced in sections 3.5.1 and 3.5.2. Validity of all system-invariants at control point $A1$ follows directly after applying the substitutions present in $S0$. Hence we only show that non-orthogonal statements of clients or services cannot disturb the system-invariants.

3.8.0.1 Invariance of I_0

Compared to other system-invariants, we give a somewhat more detailed analysis of I_0's invariance as it captures requirement 1 of the problem statement given on page 17. The structure of I_0 is such that *if* $tra(c, s, ct, st)$ holds for $c \in C, s \in S, ct \in \mathbb{N}$ and $-1 \leq st$, *then* $cd[c, ct]$ must equal $cd[s, st]$. Hence from the client component, only statement $S8.3$ potentially disturbs I_0. Indeed, $\neg tra(c, s_c, tid[c], ct2st[c, tid[c]])$ holds as precondition to each (sub-)statement of client c by $\neg\,(tid[c] < tid[c])$ and cd is only written by client c at index $c, tid[c]$. Aside from the invariants I_0 and pCD_1 we need assertions $A8.2$ and $A8.3$;

$\|[\quad I_0 \ \wedge\ s \in S \ \wedge\ c \in C \ \wedge\ cw[c, tid[c]] = s \ \wedge\ gv(c, tid[c]]) = (1, 1, 0) \ \wedge$
$\quad pCD_1 \ \wedge\ (m_2.to \vee ct2st[c, tid[c]] = tid[s])$

$\rhd \quad I_0(\sigma_{8.3})(\sigma_{8.2})(\sigma_{8.1})$

$\equiv \quad$ {definition of $I_0, \sigma_{8.2}$ is orthogonal to $I_0(\sigma_{8.3})$}
$\quad \langle \forall c', s' : c' \in C \ \wedge\ s' \in S : I_0(c', s')\rangle\ (\sigma_{8.3})\ (\sigma_{8.1})$

$\equiv \quad$ {split off: $c' = c$, use $c \in C$}
$\quad (\,\langle \forall c', s' \ :\ c' \in C/\{c\} \ \wedge\ s' \in S \ :\ I_0(c', s')\,\rangle \ \wedge\ \langle\, \forall s' \ :\ s' \in S \ :\ I_0(c, s')\rangle\,)\ (\sigma_{8.3})\ (\sigma_{8.1})$

$$\equiv \quad \left\{ \begin{array}{ll} 1^{st}\text{ conjunct:} & \text{orthogonal to substitutions: use } I_0. \\ 2^{nd}\text{ conjunct:} & \text{definition of } I_0(c, s'), tra, \text{ skolemization.} \end{array} \right\}$$

$\quad \langle \forall s', ct', st' \ :$
$\qquad s' \in S \ \wedge\ cw[c, ct'] = s' \ \wedge\ ct2st[c, ct'] = st' \ \wedge\ 0 \leq ct' < tid[c] \ \wedge$
$\qquad -1 \leq st' < tid[s'] \ :\ cd[c, ct'] \equiv cd[s', st']$
$\quad \rangle\ (\sigma_{8.3})(\sigma_{8.1})$

$\equiv \quad$ {substitution, once}
$\quad \langle \forall s', ct', st' \ :$
$\qquad s' \in S \ \wedge\ cw[c, ct'] = s' \ \wedge\ ct2st[c, ct'] = st' \ \wedge\ 0 \leq ct' < tid[c] + 1 \ \wedge$
$\qquad -1 \leq st' < tid[s'] \ :\ cd[c, ct'] \equiv cd[s', st']$
$\quad \rangle\ (\sigma_{8.1})$

$\equiv \quad$ {split off: $ct' = tid[c]$, use $c \in C$ implies $tid[c] \in \mathbb{N}$ by declaration of c and tid}
$\quad (\quad \langle \forall s', ct', st' \ :\ s' \in S \ \wedge\ cw[c, ct'] = s' \ \wedge\ ct2st[c, ct'] = st' \ \wedge$
$\qquad\quad 0 \leq ct' < tid[c'] \ \wedge\ -1 \leq st' < tid[s'] \ :\ cd[c, ct'] \equiv cd[s', st']$
$\qquad \rangle \wedge$
$\qquad \langle \forall s', st' \ :\ s' \in S \ \wedge\ cw[c, tid[c]] = s' \ \wedge\ ct2st[c, tid[c]] = st' \ \wedge\ -1 \leq st < tid[s'] \ :$
$\qquad\quad cd[c, tid[c]] \equiv cd[s', st']$
$\qquad \rangle$
$\quad)\ (\sigma_{8.1})$

$$\equiv \quad \left\{ \begin{array}{ll} 1^{st}\text{ conjunct:} & \text{orthogonal to } \sigma_{8.1}\text{, use } I_0 \\ 2^{nd}\text{ conjunct:} & \text{apply substitution; use } cw[c, tid[c]] = s \wedge s \in S \end{array} \right\}$$

$\quad \langle \forall st' \ :\ ct2st[c, tid[c]] = st' \ \wedge\ -1 \leq st' < tid[s] \ :\ (\neg m_2.to \wedge cv) \equiv cd[s, st']\,\rangle$

$\equiv \quad$ {1-point rule}
$\quad (-1 \leq ct2st[c, tid[c]] < tid[s]) \Rightarrow ((\neg m_2.to \wedge cv) \equiv cd[s, ct2st[c, tid[c]]])$

$\equiv \quad$ {use $gv(c, tid[c]]) = (1, 1, 0)$, in particular $CD[c, tid[c]] = 0$. Use $pCD_1 \ :\ \neg cd[s, ct2st[c, tid[c]]]$}
$\quad -1 \leq ct2st[c, tid[c]] < tid[s] \ \Rightarrow\ (m_2.to \vee \neg cv)$

$\equiv \quad$ {use $m_2.to \vee tid[s] = ct2st[c, tid[c]]$}
$\quad true$

$]\|$

The next thing to show is that no service disturbs I_0. There are two potentially disturbing statements in the service component. $S13.3$, in particular the assignment to $ct2st[c_s, ct_s]$ and $S18$. As for $S13.3$, observe that $\neg tra(c_s, s, ct_s, tid[s])$ holds as a postcondition of $S13.3$ as $\neg(tid[s] < tid[s])$, hence I_0 is not disturbed (widening). As for $S18$, using $A18.1^{2,3}$ and $A18.3$ it is immediate that $S18$ leaves I_0 invariant. Indeed, by $A18.1^{2,3}$ it follows $S18$ only affects $I_0(c_s, s)$. The reader may verify that the following holds:

$$(I_0(c_s, s) \ \wedge \ A18.1^{2,3} \ \wedge \ A18.3) \Rightarrow I_0(c_s, s)\ (\sigma_{18.1}).$$

3.8.0.2 Invariance of $prqi_0$

The only free variable occurring in $prqi_0$ is $rqi[S]$ in the antecedent. Hence $S5, S7, S8, S14, S16$ and $S18$ are orthogonal.

- $S6$, 1st alternative only: use $c \in C$ and $s \in S$: follows from declaration of c, s respectively.
- $S13.3$: widening.

3.8.0.3 Invariance of $prqi_1$

Free variables occurring in $prqi_1$ are $rqi[S]$ in the antecedent and $cw[C], RQO[C], RPO[C]$ and $CD[C]$ in the consequent. Hence $S7$, $S16$ and $S18$ are orthogonal.

- $S5$: from $A5.1^1$ and $pRQO_1$ we obtain $srqi(c, s, tid[c]) = \emptyset$.
- $S6$:
  - 1st alternative: use $A6.2$, $A6.3$ and substitution.
  - 2nd alternative: use $A6.4$.
- $S8.1$: use $A8.4^1$.
- $S13.4$, widening.
- $S14$, either alternative: use $A14.2^1$.

3.8.0.4 Invariance of $prqi_2$

The only free variable occurring in $prqi_2$ is $rqi[S]$ in the consequent. Hence only $S6$ (1st alternative) and $S13$ are non-orthogonal.

- $S6$ (1st alternative only): use B_{6^1}.
- $S13.4$, widening.

3.8.0.5 Invariance of $prpi_0$

The only variable occurring in $prpi_0$ is $rpi[C]$ in the antecedent. Hence $S5, S8, S13, S16$ and $S18$ are orthogonal.

- $S6$ (2nd alternative only): use $c \in C$ and $s \in S$: follows from declaration of c, s respectively.
- $S7.2$: widening.
- $S14$, either alternative: use $c \in C$ and $s \in S$: follows from declaration of c, s respectively.

3.8.0.6 Invariance of $prpi_1$

Variables occurring in $prpi_1$ are $rpi[C]$ in the antecedent and $tid[S]$, $ct2st[C]$, $rqi[S]$, $RQO[C]$, $RPO[C]$, and $CD[C]$ in the consequent. Hence only $S5$ and $S16$ are orthogonal.

- $S6$:
  - 1st alternative: use $A6.4$.
  - 2nd alternative: use $A6.3, A6.4^1$ and the fact that the message added to $rpi[c]$ is a timeout.
- $S7.3$: widening.
- $S8$: for $S8.1$: use $A8.4^2$.
- $S13.4$: using pcw_0 and the fact that m_1 is taken from $rqi[s]$, it follows that the precondition of $S13.4$ implies $srpi(c, s, ct) = \emptyset$.
- $S14$: use $A14.1^{1,3}$ and $A14.2^1$;
- $S18$: use $A18.4$.

3.8.0.7 Invariance of $pffi_0$

The only variable occurring in $pffi_0$ is $ffi[S]$ in the antecedent. Hence $S5$, $S6$, $S7$, $S13$, $S14$ and $S18$ are orthogonal.

- $S8$: use $c \in C$ and $s \in S$: follow from declaration of c, s respectively.
- $S16$: widening.

3.8.0.8 Invariance of $pffi_1$

Variables occurring in $pffi_1$ are $ffi[S]$ in the antecedent and $RQO[C]$, $RPO[C]$, $CD[C]$, $cd[C]$ and $rpi[C]$ in the consequent. By $gv(c, ct) = (1, 1, 1)$ in the consequent and the fact that every assertion in the client component expresses that $gv(c, tid[c])$ has a value different from $(1, 1, 1)$, we have by $pffi_1$ at $A5$, $A6$, $A7$ and $A8$:

$$\neg \langle \exists m, s \ : \ m \in ffi[s] \ : \ m.fa = c \ \wedge \ m.tid = tid[c] \rangle \,.$$

Consequently, $S8.2$ is the only potentially disturbing statement of the client component. For the same reason, we may exclude disturbance of $pffi_1$ by $S13$ or $S14$, $S18$ is orthogonal by $s \notin C$. This leaves only $S8.2$ and $S16$;

- $S8$: apply substitutions: use $A8.3^1$ and $A8.4^2$.
- $S16$: widening.

3.8.0.9 Invariance of pcw_0

Variables occurring in pcw_0 are $rqi[S]$ and $rpi[C]$ in the consequent. Hence statements $S5$, $S8$, $S16$ and $S18$ are orthogonal.

- $S6$: either alternative: use $A6.4$;
- $S7$: widening.
- $S13$: widening.
- $S14$: either alternative: $A14.2$.

3.8.0.10 Invariance of $pct2st_0$

Variables occurring in $pct2st_0$ are RQO in the antecedent, and $ct2st[C]$ and $tid[S]$ in the consequent. The only assignment to RQO occurs in $S6$, which only widens $pct2st_0$ by $A5.1^1$ (or by the type of RQO). The only assignment to $ct2st$ occurs in $S13$. As m_1 is taken from $rqi[s]$ in $S13$, we have by $prqi_1$: $gv(c, ct) = (1, 0, 0)$, i.e. $RQO[c, ct] = 1$ whence $S13$ is orthogonal to $pct2st_0$. The only assignment to $tid[S]$ occurs in $S18$, here we have widening.

3.8.0.11 Invariance of $pRQO_1$

Variables occurring in $pRQO_1$ are $RQO[C]$ in the antecedent and $rqi[S]$ and $rpi[C]$ in the consequent. By the conjunct $RQO[c, ct] = 0$ in the antecedent we may disregard statements $S7$, $S8$, $S14$, $S16$ and $S18$ by $A7.3$, $A8.3^1$, $A14.1^1$, $A16.1^1$ and $A18.1^1$ respectively. $S5$ is orthogonal, this leaves only $S6$ and $S13$;

- $S6$: either alternative $A6.4$ and definition of the statement: widening.
- $S13$: widening.

3.8.0.12 Invariance of pcd_0

The only free variable occurring in pcd_0 is $cd[S]$ in the consequent. Hence only $S18$ is non-orthogonal, it is trivial to verify this statement does not disturb pcd_0.

3.8.0.13 Invariance of pCD_1

Variables occurring in pCD_1 are $CD[C]$ in the antecedent and cd and $ct2st$ in the consequent. Hence only $S8$, $S13$ and $A18$ are potentially non-orthogonal.

- $S8$: use $A8.3^1$ and definition of the statement: widening.
- $S13$: using pcd_0 we have that the precondition to $S13.3$ implies $\neg cd[s, tid[s]]$.
- $S18$: use $A18^1 \Rightarrow CD[c, ct] = 1$.

3.8.0.14 Invariance of pgv_0

Variables occurring in pgv_0 are $RQO[C]$ and $CD[C]$ in the antecedent, and $tid[C]$ in the consequent. In the client component, $S6$ and $S8$ are non-orthogonal. $S6$ changes $RQO[c, tid[c]]$ and $S8$ widens pgv_0. All statements in the service component are orthogonal.

3.9 Progress (requirement 2)

In this section we prove that progress is guaranteed in the 1-R/R protocol, which covers requirement 2 of the problem statement. Precise definitions of our progress requirements are given in section 3.9.3. We prove lemmas 4, 5 and 6 in section 3.9.3. Lemma 4 states that clients cannot deadlock. Lemma 5 states that a service can neither live- nor deadlock in its innermost repetition. Lemma 6 states that a service s can only deadlock on $S13.1$ if no client c ever chooses to perform a transaction with s (in which case it is 'idle'). We conclude that these lemmas indeed cover requirement 2 in section 3.9.4.

We need a few extra assertions in order to prove the lemmas 4, 5 and 6. Those are provided in sections 3.9.1 and 3.9.2.

3.9.1 Additional assertions in the client component

We need the following additional assertions in the client components to prove that progress is guaranteed in the 1-R/R protocol;

3.9.1.1 Assertion 7

- **D7.1:** $|srpi(c, s, tid[c])| = 1 \vee gv(c, tid[c]) = (1, 0, 0)$

  local correctness: apply substitutions: use $A6.3$ and $A6.4$.

  global correctness: Let c and c' be two different clients. Statements of client c' are orthogonal to $srpi(c, s, tid[c])$ and $gv(c, tid[c]) = (1, 0, 0)$ if $c' \neq c$.
  The only non-orthogonal statement in the service component is $S14$, but only if (3.2) holds. Although in such cases, the second disjunct of $D7.1$ is made $false$, the first is made $true$ by using $A14.2^1$.

- **D7.2:**

$$\begin{aligned}
&|srpi(c, s, tid[c])| = 1 \vee \\
&(rd[s] \wedge rqi[s] = \{\langle c, s, tid[c], false\rangle\}) \vee \\
&ct2st[c, tid[c]] = tid[s]
\end{aligned}$$

  local correctness: for the first two disjuncts: apply substitutions, use $A6.3$ and $A6.4$. For the third disjunct: use $\vee$-weakening.

  global correctness The only non-orthogonal statement of client c', $c' \neq c$, is $S6^1_{c'}$: it is only non-orthogonal to the second conjunct of the second disjunct of $D7.2$. By $B_{6^1_{c'}}$ we obtain disjointness.
  Non-orthogonal statements in the service component are $S13$, $S14$ and $S18$. $S13_s$ is non-orthogonal only to the second disjunct. If this disjunct holds, $S13_s$ falsifies it while making the third disjunct $true$.
  $S14_s$ is non-orthogonal only to the first disjunct, but only if (3.2) holds. Using $A14.2^2$ we obtain disjointness.
  $S18_s$ cannot disturb the second disjunct of $D7.2$ by $A18.6$ (disjointness). It remains to show that $S18_s$ cannot disturb the third disjunct of $D7.2$. Let s be at $A18$ while c is at $A7$ such that $ct2st[c, tid[c]] = tid[s]$ holds. From $A18.5_s$, $A18.1^{2,3}_s$, $A7.2$ and $ct2st[c, tid[c]] = tid[s]$ we obtain (3.2). Using table 3.8 we obtain disjointness.

3.9.2 Additional assertions in the service component

We need the following additional assertions in the service components to prove that progress is guaranteed in the 1-R/R protocol;

3.9.2.1 Assertion 16

- **D16.1:** $gv(c, ct) = (1, 1, 0) \ \vee \ \mathit{sffi}(c, s, ct) \neq \emptyset$

  local correctness:

  - initially: apply substitutions, use $A14.1$ and $\vee-$ weakening.
  - repeatedly: apply substitutions: use $D17.1 \ \wedge \neg B_{17}$

  global correctness: as $D16.1$ equals $D17.1$, we discuss global correctness of both assertions here (no co-assertions of $A16$ or $A17$ are needed). Of client c, only statements $S6_c$ and $S8_c$ are non-orthogonal to $B16.1$ and $B17.1$, but only if (3.2) holds. From table 3.8 we obtain that control points $A6$ and $A16$ or $A17$ are disjoint under this condition. Although $S8_c$ makes the first disjunct of $D16.1$ $false$ if (3.2) holds, it makes the second $true$.

  For services s', $s' \neq s$, the only statement non-orthogonal to $D16.1$ or $D17.1$ is $S14_{s'}$, but only with precondition $c_{s'} = c_s \ \wedge \ ct_{s'} = ct_s$. But then we obtain disjointness from $S14.1^2_{s'}$ and $A16.1^2_s$ or $A17.1^2_s$.

3.9.2.2 Assertion 17

- **D17.1:** $B_{17} \ \vee \ gv(c, ct) = (1, 1, 0) \ \vee \ \mathit{sffi}(c, s, ct) \neq \emptyset$

  local correctness: use $D16.1$ and orthogonality of $S16$ in case $\neg B_{17}$ holds, or B_{17} otherwise.

  global correctness: see $D16.1$.

3.9.3 Absence of Live- or deadlock

We say a component a to be *individually deadlocked* if it is forever stuck on one of its **await** statements. Instead of saying that a is individually deadlocked, we simply say it to be deadlocked if there is no danger of confusion. If a deadlocks on a statement of the form **await**(B), the predicate $\neg B$ is either a globally correct assertion, or, it happens to be so that each time a inspects B, it equals $false$ because of some other component, a' say, disturbing B just before a inspects it. In such scenarios we say a to be *infinitely overtaken*.

The conditions waited for in either the client or service components are if $true$, always stably $true$. For the condition in $S7.1_c$, we have that no component but c itself removes anything from $rpi[c]$. Similar arguments can made for the conditions in $S13.1_s$ and $S16.1_s$. Hence there is no danger of infinite overtaking in the 1-R/R protocol and *if* a component were to deadlock on **await**(B), *then* B is a globally correct assertion.

We say service s to *livelock*, if it is forever executing its innermost repetition such that it never deadlocks. If service s livelocks, then $\neg B_{17}$ must be a globally correct assertion at $A16$ and $A17$ as it contains only local variables. Using the fairness assumption from section 3.3.4 concerning the receiving of F&F-messages, it must be the case that $\mathit{sffi}(c_s, s, ct_s) = \emptyset$ is a globally correct assertion as well at both $A16$ and $A17$. Indeed, if it was not, then at some point we would have $\mathit{sffi}(c_s, s, ct_s) \neq \emptyset$ at $A16$. But by the fairness assumption, service s would eventually select $\langle c_s, s, ct_s \rangle$ for m_3 at $S16.1_s$ because no component but s removes anything from $\mathit{ffi}[s]$. Hence B_{17} would hold at $A17$ and the innermost repetition would terminate, contradicting that s livelocks.

Given the characterizations of live- and deadlock, table 3.11 summarizes the possible statements that can live- or deadlock, and the corresponding globally correct co-assertions.

Deadlock at:	*Co-assertion:*
$S7.1_c$	$srpi(c, s_c, tid[c]) = \emptyset$
$S13.1_s$	$rqi[s] = \emptyset$
$S16.1_s$	$ffi[s] = \emptyset$
Livelock at:	*Co-assertion:*
$S16_s$ and $s17_s$	$sffi(c_s, s, ct_s) = \emptyset \wedge \neg(B_{16} \vee B_{17})$

Table 3.11: Possibly live- or deadlocking statements and corresponding assertions

Lemma 4. *Clients cannot deadlock.*

Proof. Given the earlier definition of deadlock, we further examine a client c that deadlocked. From table 3.11 we obtain this is only possible at $S7.1_c$, and that at that this control point, the following holds:

$$srpi(c, s, tid[c]) = \emptyset \tag{3.4}$$

Combined with $D7.1$, we obtain;

$$gv(c, tid[c]) = (1, 0, 0) \tag{3.5}$$

Combined with $D7.2$, we obtain;

$$(rd[s] \wedge rqi[s] = \{\langle c, s, tid[c], false\rangle\}) \vee ct2st[c, tid[c]] = tid[s] \tag{3.6}$$

Note that (3.4), (3.5), and (3.6) are all globally correct as those predicates are corollaries of assertions that were proved to be globally correct and an assumption of deadlock which is assumed to be globally correct.

Let c deadlock on $S7.1$. First we show that the second disjunct of (3.6) cannot hold, second we show that the first disjunct of (3.6) cannot stably hold, in which case we have established a contraction.

We show that the second disjunct of (3.6) cannot hold by showing that in such cases, service s cannot be at either of its control points;

- Due to $A13.2_s$ and $A7.2$, service s cannot be at control point $S13_s$.
- Note that $ct2st[c, tid[c]] = tid[s] \wedge A7.2_c$ and

$$\left((A14.1_s^{2,3} \wedge A14.4_s) \vee (A16.1_s^{2,3} \wedge A16.5_s) \vee (A14.1_s^{2,3} \wedge A17.7_s) \vee (A18.1_s^{2,3} \wedge A18.5_s)\right),$$

  imply (3.2). Hence if execution of s were at control points $A14$, $A16$, $A17$ or $A18$ while c is at $A7$ such that $ct2st[c, tid[c]] = tid[s]$ holds we have that (3.2) holds. But then we have that;

  - for control point $A14$, by execution of $S14$, s would disturb 3.4, contradicting that c deadlocks.
  - control points $A16$, $A17$ and $A18$ are disjoint due to

$$\left(ct = tid[c] \wedge (A16.1^1 \vee A17.1^1 \vee A18.1^1) \wedge (3.5)\right) \equiv false.$$

As the second disjunct of (3.6) cannot hold, it must be that the first disjunct stably holds, i.e.;

$$rd[s] \wedge rqi[s] = \{\langle c, s, tid[c], false\rangle\}.$$

From $A13.3_s$, $A14.5_s$, $A16.6_s$, $A17.8_s$ and $A18.6_s$ it follows that service s can only be at control point $A13$. By $rqi[s] = \{\langle c, s, tid[c], false\rangle\}$, service s cannot deadlock on $S13.1_s$. Indeed, it must choose m_1 such that $m_1 = \langle c, s, tid[c], false\rangle$ holds at $A14$. By $A14.5_s$ we find that the second disjunct of (3.6) is eventually disturbed. □

Lemma 5. *A service cannot live- or deadlock in its innermost repetition.*

Proof. Let s be a service that live- or deadlocks in its innermost repetition (i.e. it either deadlocks at $A16$ or it livelocks). Note that by table 3.11 we have that $\mathit{sffi}(c, s, ct_s) = \emptyset$ must then be a system-invariant. Indeed;

- if the service *livelocks*, the system-invariant *equals* the corresponding assertions in 3.11.
- if the service deadlocks at $A16_s$, the system-invariant *follows* from the corresponding assertion in table 3.11.
- no component but s can remove anything from $\mathit{ffi}[s]$. Hence if $\mathit{sffi}(c, s, ct_s) \neq \emptyset$ holds at some point, it would contradict the assumption that s live- or deadlocks.

It follows that it suffices to show that $\mathit{sffi}(c_s, s, ct_s) = \emptyset$ cannot be a system-invariant.

By $\mathit{sffi}(c_s, s, ct_s) = \emptyset$ we have that $\neg B_{17}$ is a globally correct assertion at control point $A17_s$ if s *livelocks* in its innermost repetition. Combined with $D16.1$ and $D17.1$, we obtain that $gv(c_s, ct_s) = (1, 1, 0)$ is also a system invariant if s live- *or* deadlocks in its innermost repetition. Combining $gv(c_s, ct_s) = (1, 1, 0)$ and pgv_0 we obtain that $tid[c_s] = ct_s$ is also a system invariant if s live- or deadlocks in its innermost repetition. Hence $gv(c_s, tid[c_s]) = (1, 1, 0)$ is also a system invariant if s live- or deadlocks in its innermost repetition.

From the ghost-variables vectors $A5.1^1_{c_s}$, $A6.3^1_{c_s}$, $A7.3^1_{c_s}$ and $A8.3^1_{c_s}$ it follows that client c_s must either be at control point $A7$ or at $A8$. By $A7.2_{c_s}$, $A8.2_{c_s}$, $A16.1^2_{c_s}$ and $A17.1^2_{c_s}$ we obtain that $s_c = s$. As by lemma 4, c_s cannot deadlock on $S7.1$, it eventually executes $S8$ with precondition $c = c_s \wedge tid[c] = ct_s$, disturbing $\mathit{sffi}(c_s, s, ct_s) = \emptyset$. □

It would be tempting to try to prove that s cannot deadlock on $S13.1$. Instead we prove that *if* a service s is deadlocked on $S13.1$ *then* it is 'idle', i.e. no client is interested in transacting with s.

Lemma 6. *A service s can only deadlock on $S13.1$ if no client c ever chooses s_c in $S5.1_c$, such that $s_c = s$ at $A6$.*

Proof. By contradiction: let s deadlock on $S13.1$ while some client c chooses s_c in $S5.1_c$, such that $s_c = s$ at $A6$. Then by $A13.3$ and table 3.11 we have at $A13$: $rqi[s] = \emptyset \wedge rd[s]$. At $A6$, we have $cw[c, tid[c]] = s$. But execution of $S6.2$ with precondition $cw[c, tid[c]] = s \wedge rqi[s] = \emptyset \wedge rd[s]$ disturbs $rqi[s] = \emptyset$. □

3.9.4 Conclusion

In this section we briefly show that lemmas 4, 5 and 6 indeed cover requirement 2 of the problem statement given on page 17. Clients cannot deadlock by lemma 4. The service chosen by a client for a distributed transaction can neither dead- nor livelock by lemmas 5 and 6 respectively. Hence both parties involved in a size-1 transaction always make progress. At some point, both have finished processing their part of the distributed transaction, at which point the distributed transaction has terminated.

Chapter 4

Verification of the n-R/R protocol

In chapter 3 we proved requirements 1 and 2 of the problem statement on page 17 to be met by the 1-R/R protocol. In this chapter we generalize the results from chapter 3 to transactions of an arbitrary, positive size n: the n-R/R protocol. In doing so we reuse as much as possible from chapter 3;

- All but one invariant, all auxiliary functions, and all concepts and definitions introduced in chapter 3 are reused without modification. Only one invariant (pgv_0) is slightly modified, we add some fresh invariants in the sequel.
- The service component from the 1-R/R protocol (see section 3.4.3) is reused without modification, along with all its assertions.
- The main program from the 1-R/R protocol (see section 3.4.4) is only slightly modified.
- The formal-model of the bus given in section 3.3 is reused without modification.

The structure of this chapter is also similar to that of chapter 3. In section 4.1 we introduce the new client component. Requirement 1 of the problem statement on page 17 is formalized in section 4.2 by means of two system-invariants. Assertions of the client components are introduced and proved to be locally correct in section 4.3. Global correctness of all assertions is proved in section 4.4, correctness of all system-invariants is proved in section 4.5. Requirement 2 of the problem statement on page 17 is covered in section 4.6.

We advise readers to keep copies of pages 57, 88, 130, 132 and 133 at hand while reading this chapter in order to avoid a lot of page turning in the sequel (see appendix B).

4.1 Specification

We need some additional auxiliary functions and global variables in comparison to the 1-R/R protocol: those are introduced in sections 4.1.1 and 4.1.2 respectively. We explain the overal structure of the client component in relation to that of the 1-R/R protocol in section 4.1.3. The adapted client component itself is given in section 4.1.4. In section 4.1.5 we discuss the placing of the atomicity brackets in the client component.

4.1.1 Additional auxiliary functions

We need the additional auxiliary functions $sct : \mathbb{N} \times \mathbb{N} \to 2^{\mathbb{N}}$, $scw : C \times \mathbb{N} \times \mathbb{N} \to 2^{S}$ and $allRep : C \times 2^{\mathbb{N}} \times 2^{Message} \to \mathbb{B}$. Definitions of the auxiliary functions are given below;

$$\begin{array}{lcl} sct(ct_0, K) & = & \{ct \mid ct_0 \leq ct < ct_0 + K\} \\ scw(c, ct_0, K) & = & \{cw[c, ct] \mid ct \in sct(ct_0, K)\} \\ allRep(c, CT, M) & = & |CT| = |M| \ \wedge \ CT = \{m.tid|m \in M\} \wedge \\ & & \langle \forall m \ : \ m \in M \ : \ m.fa = cw[m.ta, m.tid] \ \wedge m.ta = c\rangle \end{array}$$

The functions $sct(tid[c], sz)$ and $sc(s, tid[c], sz)$ are used to generalize the variables $tid[c]$ and s_c respectively, to distributed transactions with any positive size sz. The predicate $allRep(c, CT, M)$ expresses that the set M is a smallest set containing a reply or timeout of each transaction with a transaction-id ct, $ct \in CT$.

4.1.2 Additional global variables

In this section we discuss some additional global variables needed in our specification. First we explain the need for those. In section 3.2 it was discussed that both clients and services assign a transaction-id to distributed transactions in order to uniquely identify them. The interpretation of the client-transaction-id is somewhat different in view of distributed transactions with a size greater than 1. At the end of a distributed transaction, the variable $tid[c]$ equals all of the following;

- the sum of the size of all distributed transactions *ever* performed by client *c*, and,
- the number of requests *ever* sent by *c*, and,
- the number of commit-decisions *ever* sent by *c*, and,
- the number of requests received + the number of timeouts *ever* received by *c*.

It follows that multiple client-transaction-id's may be assigned to a single distributed transaction because the transaction size may be greater than one. Hence we need the following additional global variables in order to be able to identify such transactions;

$$\begin{array}{lll} \textbf{var}\ dtid & : Array[C] & : \mathbb{N}; \\ \textbf{var}\ dt2ct & : Array[C][\mathbb{N}] & : \mathbb{N}; \\ \textbf{var}\ sz & : Array[C] & : \mathbb{N}; \end{array}$$

An intuitive description of the arrays is given below;

- *dtid* (distributed-transaction-id):
  $dtid[c] = K \equiv$'client *c* has completed K distributed transactions'.
- *dt2ct* (distributed-transaction-id to client transaction-id), for $0 \leq dt < dtid[c]$. $dt2ct[c, dt] = ct \equiv$ 'ct is the smallest client-transaction-id of the $dt + 1^{th}$ distributed transaction of client *c*.
- sz (distributed transaction size):
  $sz[c] = K \equiv$ 'client *c* is performing a distributed transaction with size K'.

All of the variables above except $sz[c]$ are specification variables: only $sz[c]$ is present in the informal specification as $\mathtt{sz}_c$ (see section 2.2.2). The corresponding changes needed to the main-program given in section 3.4.4 are;

- The declarations of *dtid*, *dt2ct* and sz.
- The catenation of statement $S0$ with *dtid*, *dt2ct* $:= \vec{0}, \vec{0}$. The array sz needs no initialization.

4.1.3 Structure of the client component

In this subsection we discuss the relation between the program texts of the client components of the 1-R/R and n-R/R protocols. We advise the reader to compare the program texts given on pages 56 and 88 while reading this subsection. The number of, and the labeling of the statements is the same in both protocols. Note that we neither consider variable declarations nor guard-evaluations to be sub-statements when labeling sub-statements. The main differences between the client components are given below;

- Recall from section 3.9.1 that statement i_c, $6 \leq i \leq 8$, in the 1-R/R protocol had a co-assertion $cw[c, tid[c]] = s_c$: s_c was used only to abbreviate $cw[c, tid[c]]$. We do not use such abbreviations in the n-R/R protocol.
- The additional local variables CT_{j_c}, $6 \leq j \leq 9$ present in each client c, all sets of $\mathbb{N}$. The sets are used to store client-transaction-ids during a distributed transaction. At $A6$, $A7$, and $A8$ we have that the following holds;

  $$\left(\bigcap j : 6 \leq j \leq 9 : CT_j\right) = \emptyset \wedge \left(\bigcup 6 \leq j \leq 9 : CT_j\right) = sct(c, tid[c], sz[c])$$

- Statement i of client c in the n-R/R protocol, $6 \leq i \leq 8$, is a repetition of the following structure;
  - initially $CT_{i_c} = sct(tid[c], sz[c])$, and, during each round:
    - $*$ select some transaction-id $ct \in CT_{i_c}$;
    - $*$ perform the same send/receive/commit operations for (partial-)transaction $(c, cw[c, ct], ct)$ as is done during execution of statement i in the 1-R/R protocol for transaction $(c, s, tid[c])$.
    - $*$ move the element ct from CT_{i_c} to CT_{i+1_c};
  - end if CT_i is empty.

  Note that the choice of ct in statement $S7.2$ of the n-R/R protocol may seem superfluous. Indeed, we have $ct = m_2.tid$ as precondition to $S7.3, 4$. However, we use it (as above) to allow for a more uniform treatment of the statements of the protocol.
- The variable m_{2_c} was used to receive the reply in statement $S7_c$ of the 1-R/R protocol. In the n-R/R protocol we store all replies that must be received during a transaction in the additional set M_{2_c} of messages.
- In statement $S5$ of the 1-R/R protocol we selected a single service and recorded this choice in cw. In statement $S5$ of the n-R/R protocol we do the same for a set of services S' and also record the transaction size in $sz[c]$. The set S' is only used for notational convenience. The sets CT_j, $6 \leq j \leq 9$, and M_2 are also initialized in statement $S5$.

Note that the 1-R/R and n-R/R protocols are the same (up to cosmetic differences) if clients always choose to perform size-1 transactions in the n-R/R protocol. We return to the relation between the client components of the 1-R/R and n-R/R protocols when discussing the assertions in section 4.3.1.

4.1.4 Clients

An explanation of the client component is given on page 87.

```
component $Client(c\ :\ C) =$
$|[$ var $M_2$ : Set of $Message$;
  var $CT_6, CT_7, CT_8, CT_9$ : Set of $\mathbb{N}$;
  var $ct$ : $\mathbb{N}$;
  var $cv$ : $\mathbb{B}$;

  do $true \rightarrow$
      { Assertion 5 }
      $\langle$ var $S'$ : Set of $S$;
        $sz[c],\ S' : S' \subseteq S \wedge sz[c] = |S'|$;
        $cw[c][\, tid[c]..tid[c] + sz[c] - 1\,] := \overrightarrow{S'}$;
        $CT_6,\ CT_7,\ CT_8,\ CT_9,\ M_2 := sct(tid[c], sz[c]),\ \emptyset,\ \emptyset,\ \emptyset,\ \emptyset$;
      $\rangle$

      { Assertion 6 }
      do $CT_6 \neq \emptyset \rightarrow$
          $\langle$ $ct : ct \in CT_6$;
            if  $rqi[cw[c, ct]] = \emptyset \wedge rd[cw[c, ct]]) \rightarrow rqi[cw[c, ct]],\ RQO[c, ct] :=$
                    $rqi[cw[c, ct]] \cup \{\langle c, cw[c, ct], ct,\ false\rangle\},\ RQO[c, ct] + 1$;
            $[\!]$ $\neg(rqi[cw[c, ct]] = \emptyset \wedge rd[cw[c, ct]]) \rightarrow rpi[c],\ RQO[c, ct],\ RPO[c, ct] :=$
                    $rpi[c] \cup \{\langle cw[c, ct], c, ct,\ true\rangle\},\ RQO[c, ct] + 1,\ RPO[c, ct] + 1$;
            fi
            $CT_6,\ CT_7 := CT_6/\{ct\},\ CT_7 \cup \{ct\}$;
          $\rangle$
      od

      { Assertion 7 }
      do $CT_7 \neq \emptyset \rightarrow$
          $\langle$ var $m_2$ : $Message$;
            await( $\exists m :: m \in rpi[c] \wedge m.tid \in CT_7 \wedge m.fa = cw[c, m.tid]$ );
            $ct, m_2 : ct = m_2.tid \wedge m_2 \in rpi[c] \wedge m_2.tid \in CT_7 \wedge m_2.fa = cw[c, m_2.tid]$;
            $rpi[c],\ M_2,\ CT_7,\ CT_8 := rpi[c]/\{m_2\},\ M_2 \cup \{m_2\},\ CT_7/\{m_2.tid\},\ CT_8 \cup \{m_2.tid\}$;
            $cv : true$;
          $\rangle$
      od

      { Assertion 8 }
      $\langle$ do $CT_8 \neq \emptyset \rightarrow$
            $ct : ct \in CT_8$;
            $CD[c, ct],\ cd[c, ct] := CD[c, ct] + 1,\ \neg(\bigvee m : m \in M_2 : m.to) \wedge cv$;
            $ffi[cw[c, ct]] := ffi[cw[c, ct]] \cup \{\langle c,\ cw[c, ct],\ ct, \neg cd[c, ct]\rangle\}$;
            $CT_8,\ CT_9 := CT_8/\{ct\},\ CT_9 \cup \{ct\}$;
        od
        $tid[c],\ dtid[c],\ dt2ct[c, dtid[c] + 1] := tid[c] + sz[c],\ dtid[c] + 1,\ tid[c] + sz[c]$ ;
      $\rangle$
  od
$]|$
```

4.1.5 Atomicity

In this section we validate the placing of the atomicity brackets in the client component given in section 4.1.4. We refer to section 3.4.5 for the placing of the atomicity brackets in the service components. For the brackets surrounding S_0 we also refer to section 3.4.5.

For statements $S5$, $S6$ and $S7$ we can also reuse the arguments provided in section 3.4.5. Indeed, in comparison to the client program of the 1-R/R protocol given in section 3.4.2, we have that;

- We refer to services in the program text of the n-R/R protocol by means of $cw[c, ct']$ for $ct' \in sct(tid[c], sz[c])$ instead of the variable s_c in the 1-R/R protocol. Note that both $cw[c]$ and s_c are local variables. Hence this difference is merely cosmetic.
- Only assignments to local variables have been added. Because those variables can not be read by any other component than the owner, we may as well place them outside the angular brackets. This would only produce somewhat different interleavings, the behavior of clients or services would however be no different.

The brackets surrounding $S8$ in the n-R/R protocol reflect the use of a unit of work that is always committed (see section 3.3.4). All variables occurring in $S8_c$ except *ffi* are local variables of client c. Again, because those variables can not be read by any other component, we may as well place them outside the angular brackets.

4.2 Partial Correctness criteria

In this section we formalize requirement 1 of the problem statement on page 17 and also introduce some fresh system-invariants and auxiliary functions needed in the sequel. Requirement 1 is formalized in section 4.2.1, section 4.2.2 introduces the additional system-invariants and auxiliary functions.

4.2.1 Formalization of requirement 1

The main partial correctness criterion is to ensure that for each distributed transaction, eventually the client and all services it transacted with agree on the outcome: to commit or abort (requirement 1 of the problem statement given on page 17). To this end we reuse I_0 as introduced in section 3.5.1. This system-invariant is however not strong enough in view of distributed transactions with size greater than 1. We explain why this is so and what is needed in addition to I_0.

Let ct be a client-transaction-id, $0 < ct < tid[c]$. The interpretation of $cd[c, ct]$ in the n-R/R protocol is as follows;

- $cd[c, ct]$ $\equiv$'The distributed transaction in which client c sent its $ct + 1^{th}$ request was committed'
- $\neg cd[c, ct]$ $\equiv$'The distributed transaction in which client c sent its $ct + 1^{th}$ request was aborted'

In particular, if a distributed transaction with size 2, say, was assigned client-transaction-ids $ct - 1$ and ct, we need in addition that $cd[c, ct - 1] \equiv cd[c, ct]$ holds. This is expressed by the fresh system-invariant $pdtid_0$;

$$\begin{array}{lcl} pdtid_0 & : & \langle \forall c \,:\, c \in C \,:\, pdtid_0(c) \,\rangle \\ pdtid_0(c) & : & \langle \forall dt \,:\, 0 \leq dt < dtid[c] \,: \\ & & \quad (\, \forall ct, ct' \,:\, dt2ct[c, dt] \leq ct \leq ct' < dt2ct[c, dt + 1] \,:\, cd[c, ct] \equiv cd[c, ct'] \,) \\ & & \rangle \end{array}$$

$pdtid_0(c)$ holds initially for all c if we initialize $dtid$ by $\vec{0}$. As the reader may check, an increase of $dtid[c]$ by one, at the end of a distributed transaction, must have the following precondition:

$$(\ \forall ct, ct' \ :\ dt2ct[c, dtid[c]] \leq ct \leq ct' < dt2ct[c, dtid[c]+1] \ :\ cd[c, ct] \equiv cd[c, ct']\ ).$$

A nice property of $pdtid_0(c)$ is that it contains only local variables of client c: this limits the number of proofs needed to show its invariance.

4.2.2 Additional system-invariants and auxiliary functions

We reuse all but one system-invariant introduced in section 3.5.2.2 without modification. All repetition-invariants introduced in section 3.5.2.2 are reused without modification. The only system-invariant that changed is pgv_0: it is changed to pgv_0^* given below (we will also use $pgv_0^*(c)$ for clients c in the sequel);

$$\begin{array}{lll} pgv_0^* & : & \langle \forall c, ct' : gv(c, ct') = (1,0,0) \vee gv(c, ct') = (1,1,0) : ct' \in sct(tid[c], sz[c]) \rangle \\ pgv_0^*(c) & : & \langle \forall ct' : gv(c, ct') = (1,0,0) \vee gv(c, ct') = (1,1,0) : ct' \in sct(tid[c], sz[c]) \rangle \end{array}$$

We also need the following *additional* system-invariants;

$$\begin{array}{lll} pdt2ct_0 & : & \langle \forall c :\ c \in C\ : tid[c] = dt2ct[c, dtid[c]]\ \rangle \\ pdt2ct_1 & : & \langle \forall c, dt :\ c \in C\ \wedge\ dtid[c] \leq dt\ :\ dt2ct[c, dt] \leq tid[c]\ \rangle \end{array}$$

From the combination of the two invariants it follows we have for each client c;

$$[dt2ct[c, dtid[c]]..dt2ct[c, dtid[c]+1]) = \emptyset$$

All differences between invariants in the 1-R/R and n-R/R protocols have been summarized in table 4.1.

New:	System-invariants:	Introduced in section
	$pdtid_0$	4.2.1
	$pdt2ct_0, pdt2ct_1$	4.2.2
Reused:		
	I_0	3.5.1
	$prqi_0, prqi_1, prpi_0$	3.5.2.2
	$prpi_1, pffi_0, pffi_1$	3.5.2.2
	$pcw_0, pct2st_0$	3.5.2.2
	$pRQO_1(c), pcd_0(c), pCD_1(c)$	3.5.2.2
Changed:		
	pgv_0^* (from pgv_0)	4.2.2
Reused:	Repetition-invariants:	
Service s:	$pct2st_1(s), prpi_2(s)$	3.5.2.3
Client c:	$pRQO_0(c), pRPO_0(c), pCDO_0(c)$	3.5.2.3

Table 4.1: Differences between invariants in the 1-R/R- and n-R/R protocols

4.3 Assertions: local correctness

In section 4.3.1 we present the assertions in the client components, along with a proof of their local correctness.

Note that in comparison to the system-invariants in n-R/R we *changed* only pgv_0 (to pgv_0^*), pgv_0 was not used in proving *local* correctness of any assertion of the service component, and all other system-invariants have been reused. Also, note that we reused the service component *and* its annotations without modifications. Hence we refer to section 3.6.3 for the assertions in the service components and a proof of their local correctness.

We did not add or change loop-invariants of clients or services in the n-R/R protocol. Hence we need no strengthening of assertion A_0 defined in section 3.6.1. The initializations of $dtid$, $dt2ct$ and sz that were catenated to S_0 - see section 4.1.2- are orthogonal to A_0. Hence we may also reuse the corresponding local-correctness argument given in section 3.6.1.

4.3.1 Assertions in the client component

Before actually discussing the assertions of the client component, we discuss their relation to the assertions of the client component of the 1-R/R protocol. This relation is reflected in the numbering of the co-assertions in the n-R/R protocol. Table 4.2 shows which assertions have been added.

Assertion	*Additional co-assertions*					
$A5$	none					
$A6$	$A6.3^2$	$A6.4^2$	$A6.5$	$A6.6$		
$A7$	$A7.3^2$	$A7.4^2$	$A7.5$	$A7.6$	$A7.7$	
$A8$		$A8.3^2$	$A8.5$	$A8.6$	$A8.7$	$A8.8$

Table 4.2: Additional assertions

Assertions in the same column have been added for a similar reason which we discuss here. The changes in the assertions are due to the changes in the client component. Correspondingly, assertions $A6$, $A7$ and $A8$ are now repetition invariants of the repetitions within statements $S6$, $S7$ and $S8$ respectively. For this reason some co-assertions were added or changed. All additional assertions are listed in table 4.2, all assertions that were changed fundamentally are given in table 4.3. An explanation of the assertions in tables 4.2 and 4.3 is given below;

- Let $i \in \{6, 7\}$, $j \in \{3, 4\}$. The assertions in the first two columns of table 4.2 have the following structure:
  - Assertion $i.j^1$ describes properties of transactions $ct' \in CT_i$
  - Assertion $i.j^2$ and $(i+1).j^1$ describe properties of transactions $ct' \in CT_{i+1}$
  - Assertion $i.j^1$ is the generalization of assertion $i.j^1$ from the 1-R/R protocol (we explain what is meant by 'generalization' in the sequel).
  - Assertion $i.j^2$ equals assertion $(i+1).j^1$, the generalization of assertion $(i+1).j^1$ from the 1-R/R protocol.

  By 'generalization' in the last two bullets above we mean that predicates of the form $P(c, s, tid[c])$ in assertion i of client c in the 1-R/R protocol have been replaced by a predicate of the form:

  $$\langle \forall ct' \;:\; ct' \in CT_i : P(c, cw[c, ct'], ct') \rangle .$$

- In order to maximize the correspondence between the client components of the single- and n-R/R protocol on the one hand, and to minimize the number of proof obligations on the other hand, the choice was made to introduce the variables $CT_{6,7,8,9}$

and M_2 and perform all necessary initializations to those in statement $S5.3$. The price of this is of course that we need additional assertions that describe properties of $CT_{6,7,8,9}$ and the effect of $S5.3$. Fortunately, the additional assertions due to $CT_{6,7,8,9}$ contain only local variables. The assertions in columns three and four of table 4.2 are due to the introduction of $CT_{6,7,8,9}$ and M_2.

- $A8.8$ is needed for showing invariance of I_0 in the sequel.

- Due to the introduction of M_2, the choice was made to move assertion $A8.3^2$ from the 1-R/R protocol to assertions $A7.7$ and $A8.7$. This covers the last column in table 4.2 and also explains why those assertions appear in table 4.3 which lists the assertions that were fundamentally changed, or moved to a new placeholder.

- In the 1-R/R protocol we used the variable s to abbreviate $cw[c, tid[c]]$ in the client component. Hence we needed co-assertions $A6.2$, $A7.2$ and $A8.2$ (the first column in table 4.3). We do not use this abbreviation anymore in the n-R/R protocol. Hence those assertions have become superfluous in the n-R/R protocol. The placeholders were reused for assertions describing that during each distributed transaction, a service s can be selected at most once.

- Assertion $A6.1$ has been weakened: the transactions-ids from $sct(tid[c], sz[c])$ for which messages have been sent have been removed from the domain description. Due to the presence of $A6.3^2$ however, we do not loose any information.

Assertion	*Fundamentally changed co-assertions*	
$A5$	none	
$A6$	$A6.1, 2$ (changed)	
$A7$	$A7.2$ (changed)	
$A8$	$A8.2$ (changed)	$A8.3^2$ (moved to $A7.7$ and $A8.7$)

Table 4.3: Fundamentally changed co-assertions

In the remainder of this section we treat all assertions of client c and their local correctness in a similar way as done in section 3.6.2 for client c in the 1-R/R protocol. Due to all assertions being repetition invariants, we provide for each assertion two proofs:

- A proof that shows the invariant to be initially correct: this proof is always preceded by the keyword 'Initially'. The second column in table 4.4 shows of what statement the assertion in the first column must be shown to be a postcondition.

- A proof that shows that the repetition-body does not disturb the invariant: this proof is always preceded by the keyword 'Repeatedly'. Note that by the use of the atomicity brackets, the repetitions of statements $S6$ and $S7$ contain only a single atomic statement. The repetition of $S8$ is completely enclosed between atomicity brackets. The third column in table 4.4 shows of what statement the assertion in the third column must be shown to be a postcondition. In this proof we always provide the assertions needed *in addition* to the assertion of which we are proving local correctness (i.e. we do not give the assertion itself as well). Because the proofs are straightforward we do not explain them in detail. For almost all quantifications one must apply all substitutions and split of the case $ct' = ct$ (or $m = m_2$). The assertions and/or invariants needed for this split-off part are the ones given in the hints.

We also provide variant functions for the repetitions in statements $S6$, $S7$, and $S8$. These variant functions prove termination of the corresponding repetitions.

Postcondition	*to statements:*	
Assertion	*Initially*	*Repeatedly*
$A5$	$S0$	outermost repetition-body
$A6$	$S5$	repetition-body of $S6$
$A7$	$S6$	repetition-body of $S7$
$A8$	$S7$	repetition-body of $S8$

Table 4.4: Proof obligations Repetition invariants of client c

4.3.1.1 Assertion 5

$A5.1$ expresses the repetition invariants of the outermost repetition in the client component. Because we do not want this repetition to terminate, we do not provide a variant function for it.

- **assertion 5.1:** $pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c)$

  local correctness:

  - Initially: follows from assertion 1.1.
  - Repeatedly: $A5.1 \Leftarrow \big(\big(A8.1 \ \wedge \ A8.3^2 \ \wedge \ A8.5^1 \ \wedge \ \neg B_8 \big) \left(\sigma_{8.5}\right) \big)$.

- **corollary 5.2:** $\langle \forall ct' : tid[c] \leq ct' : gv(c, ct') = (0, 0, 0) \rangle$

  local correctness: corollary of assertion 5.1.

4.3.1.2 Assertion 6

As a variant function for the repetition, one can use $|CT_6|$. By $A6.5$ we have that this function is limited by the number of services chosen. Also, $|CT_6|$ decreases in every round.

- **assertion 6.1:** $(\ pRQO_0(c) \ \wedge \ pRPO_0(c)\ )\ (\ \mathbb{N} \ := \ \mathbb{N}/CT_7\ ) \ \wedge \ pCD_0(c)$

  local correctness:

  - Initially: use assertion 5.1 and orthogonality of $S5$.
  - Repeatedly: assignments to RQO and/or RPO are at index c, ct. As ct is added to CT_7 we have widening.

- **assertion 6.2:** $|scw(c, tid[c], sz[c])| = |sct(tid[c], sz[c])|$

  local correctness:

  - Initially: apply substitutions.
  - Repeatedly: orthogonality of the repetition body.

- **corollary** $A6.3^1$ **and assertion** $A6.3^2$**:**

$$\begin{array}{l} \langle \forall ct' : ct' \in CT_6 \ : \ gv(c, ct') = (0,0,0)\ \rangle \ \wedge \\ \langle \forall ct' : ct' \in CT_7 \ : \ gv(c, ct') = (1,0,0) \ \vee \ gv(c, ct') = (1,1,0)\ \rangle \end{array}$$

  local correctness: $A6.3^1$ is a corollary of $A6.1$ and $A6.5$. This leaves only $A6.3^2$;

- Initially: apply substitutions.
- Repeatedly: apply substitutions, use $A6.3$.

- **corollary 6.4:**

$$\langle \forall ct' : ct' \in CT_6 \ : \ srqi(c, cw[c,ct'], ct') \ \cup \ srpi(c, cw[c,ct'], ct') = \emptyset \ \rangle \land$$
$$\langle \forall ct' : ct' \in CT_7 \ : |srqi(c, cw[c,ct'], ct') \cup srpi(c, cw[c,ct'], ct')| \leq 1 \ \rangle$$

$A6.4^1$ is a corollary of $A6.1$, $A6.5^2$ and $pRQO_1$. $A6.4^2$ is a corollary of pcw_0.

- **assertion 6.5:**

$$CT_6 \cup CT_7 = sct(tid[c], sz[c]) \ \land \ \langle \forall i,j \ : \ 6 \leq i < j < 10 \ : \ CT_i \cap CT_j = \emptyset \rangle$$

local correctness: both for initial and repeated validity: apply substitutions.

- **assertion 6.6:** $CT_8 = \emptyset \ \land CT_9 = \emptyset \ \land \ M_2 = \emptyset$

local correctness:

- Initially: apply substitutions.
- Repeatedly: orthogonality of the repetition body.

4.3.1.3 Assertion 7

As a variant function, one can use $|CT_7|$.

- **assertion 7.1:**

$$(\, pRQO_0(c) \ \land \ pRPO_0(c) \,)\, (\, \mathbb{N} \ := \ \mathbb{N}/sct(tid[c], sz[c]) \,) \ \land \ pCD_0(c)$$

local correctness:

- Initially: use $A6.1$ and $A6.5^1$.
- Repeatedly: orthogonality of the repetition body.

- **assertion 7.2:** $|scw(c, tid[c], sz[c])| = |sct(tid[c], sz)|$

local correctness:

- Initially: use $A6.2$.
- Repeatedly: orthogonality of the repetition body.

- **assertion 7.3:**

$$\langle \forall ct' : \ ct' \in CT_7 \ : \ gv(c, ct') = (1,0,0) \ \lor \ gv(c, ct') = (1,1,0) \ \rangle \land$$
$$\langle \forall ct' : \ ct' \in CT_8 \ : \ gv(c, ct') = (1,1,0) \ \rangle$$

local correctness:

- Initially: use $A6.3^2$ and $A6.6^1$.
- Repeatedly: for $A7.3^1$: observe that CT_7 shrinks whence we have widening. For $A7.3^2$: use $prpi_1$.

- **corollary** $A7.4^1$ **and assertion** $A7.4^2$**:**

$$\langle \forall ct' : \ ct' \in CT_7 \ : \ |srqi(c, cw[c,ct'], ct') \cup srpi(c, cw[c,ct'], ct')| \leq 1 \ \rangle \land$$
$$\langle \forall ct' : \ ct' \in CT_8 \ : \ srqi(c, cw[c,ct'], ct') = \emptyset \land \ srpi(c, cw[c,ct'], ct') = \emptyset \ \rangle$$

local correctness: $A7.4^1$ is a corollary of pcw_0. This leaves only $A7.4^2$;

- Initially: use $A6.6^1$.
- Repeatedly: using $prqi_0$, $prpi_0$, and $C \cap S = \emptyset$ we obtain $rqi[m_2.fa] \cap rpi[m_2.ta] = \emptyset$ as precondition to $S7.3$. Combined with $m_2.fa = cw[c, m_2.tid]$ and the removal of m_2 from $rpi[c]$, the necessary precondition to $S7.3$ follows.

- **assertion 7.5:**

  $$CT_7 \cup CT_8 = sct(tid[c], sz[c]) \ \wedge\ \langle \forall i, j \ :\ 6 \leq i < j < 10 \ :\ CT_i \cap CT_j = \emptyset \rangle$$

  local correctness:

  - Initially: use $A6.5$, $A6.6^1$ and $\neg B_6$.
  - repeated validity : apply substitutions.

- **assertion 7.6:** $CT_9 = \emptyset \ \wedge\ allRep(c, CT_8, M_2)$

  local correctness:

  - Initially: use $A6.6$.
  - Repeatedly: apply substitutions; from $A7.5^2$, $m_2.tid \in CT_7$ and $A7.6^2$ we can infer $m_2 \notin M_2$. The predicate $m_2.ta = c \ \wedge\ m_2.tid \in CT_7 \ \wedge\ m_2.fa = cw[c, m_2.tid]$ is established by $S7.2$ and $prpi_1$. $A7.6^1$ is orthogonal to $S7$.

- **assertion 7.7:** $\langle \forall m \ :\ m \in M_2 \ :\ ct2st[c, m.tid] = tid[m.fa] \ \vee\ m.to \,\rangle$

  local correctness:

  - Initially: use $A6.6^3$.
  - Repeatedly: use $prpi_1$.

4.3.1.4 Assertion 8

As a variant function for the repetition in $S8$, one can use $|CT_8|$.

- **assertion 8.1:**

  $$(\,pRQO_0(c) \wedge pRPO_0(c)\,)\,(\,\mathbb{N} := \mathbb{N}/sct(tid[c], sz[c])\,) \wedge pCD_0(c)(\,\mathbb{N} := \mathbb{N}/CT_9\,)$$

  local correctness:

  - Initially: use $A7.1$.
  - Repeatedly: for the first two conjuncts we have orthogonality. For the third conjunct: the assignment to CD is at index c, ct. As ct is added to CT_9 we have widening.

- **assertion 8.2:** $|scw(c, tid[c], sz[c])| = |sct(tid[c], sz[c])|$

  local correctness:

  - Initially: use $A7.2$.
  - Repeatedly: orthogonality of the repetition body.

- **assertion 8.3:**

$$\langle \forall ct' : ct' \in CT_8 : gv(c, ct') = (1,1,0) \rangle \wedge$$
$$\langle \forall ct' : ct' \in CT_9 : gv(c, ct') = (1,1,1) \rangle$$

  local correctness:

  - Initially: use $A7.3^2$ and $A7.6^1$.
  - Repeatedly: As CT_8 shrinks we have widening of $A8.3^1$. For $A8.3^2$: use $A8.3$ and substitution.

- **assertion 8.4:**

$$\langle \forall ct' : ct' \in sct(tid[c], sz[c]) : srqi(c, cw[c, ct'], ct') \cup srpi(c, cw[c, ct'], ct') = \emptyset \rangle$$

  local correctness:

  - Initially: use $A7.4^2$, $A7.5^1$ and $\neg B_7$.
  - Repeatedly: orthogonality of the repetition body.

- **assertion 8.5:**

$$CT_8 \cup CT_9 = sct(tid[c], sz[c]) \wedge \langle \forall i, j : 6 \le i < j < 10 : CT_i \cap CT_j = \emptyset \rangle$$

  local correctness:

  - Initially: use $A7.5$, $A7.6^1$ and $\neg B_7$.
  - Repeatedly: apply substitutions.

- **assertion 8.6:** $allRep(\ c,\ sct(tid[c], sz[c]),\ M_2\ )$

  local correctness:

  - Initially: use $A7.5^1$, $A7.6^2$ and $\neg B_7$.
  - Repeatedly: orthogonality of the repetition body.

- **assertion 8.7:** $\langle \forall m : m \in M_2 : ct2st[c, m.tid] = tid[m.fa] \vee m.to \rangle$

  local correctness:

  - Initially: use $A7.7$.
  - Repeatedly: orthogonality of the repetition body.

- **assertion 8.8:**

$$\langle \forall ct' : ct' \in CT_9 : cd[c, ct'] \equiv (\neg (\bigvee m : m \in M_2 : m.to) \wedge cv) \rangle \wedge$$
$$\langle \forall ct' : ct' \in CT_9 : \neg cd\, [cw\, [c, ct'] , ct2st\, [c, ct']] \rangle$$

  local correctness:

  - Initially: use $A7.6^1$.
  - Repeatedly: for $A8.8^1$: apply substitutions. As for $A8.8^2$: it suffices to show that $\neg cd\, [cw\, [c, ct'] , ct2st\, [c, ct']]$ is precondition to $S8.4$. Using $A8.3^1$ and pCD_1 we have as precondition to $S8.2$: $\neg cd\, [cw\, [c, ct'] , ct2st\, [c, ct']]$. $S8.1$ and $S8.2$ are orthogonal to this predicate as $cw\, [c, ct'] \in S$ by declaration of cw. Hence it it is also precondition to $S8.4$.

4.4 Assertions: global correctness

4.4.1 Introduction

In this section we prove global correctness of the assertions presented in sections 4.3.1 and 3.6.3. We reuse as much as possible from the results obtained in section 3.7. Exactly what is reused is discussed in the subsections that follow.

We have the following proof obligations;

No client c' disturbs assertions of a different client c: This proof obligation is discussed in section 4.4.2.

No service s disturbs assertions of client c: This proof obligation is discussed in section 4.4.3.

No client c disturbs assertions of service s: This proof obligation is discussed in section 4.4.4.

No service s' disturbs assertions of a different service s: Recall that the service component and its assertions are the same as in the 1-R/R protocol, that we did not use pgv_0 to prove that services do not disturb each others assertions in 1-R/R protocol, and that all invariants but pgv_0 have been reused without modification in the n-R/R protocol. We refer the reader to the proof given in section 3.7.4.

As opposed to section 3.7.3, we discuss absence of disturbance of clients by services and vice versa in two separate sections, because the n-R/R protocol and its assertions are somewhat more complex than the 1-R/R protocol.

All local and private variables and their owners are shown in table 4.5. We added no shared variables to the specification of the n-R/R protocol (in comparison to the 1-R/R protocol). Hence table 3.5 shows all shared variables.

Local variable	*owner*
$tid[a], cd[a]$	component a
$cw[c], CD[c], dtid[c], dt2ct[c], RQO[c], sz[c]$	client c
M_{2c},CT_{ic} (for $6 \le i \le 9$), ct_c, cv_c, S'_c, m_{2c}	client c
m_{1s},m_{3s},ct_s	service s
Private variable	*owner*
$rd[s]$	service s

Table 4.5: Local- and private variables and their owners

Note that we have in fact hidden control points in the client components immediately after the evaluation of the guards B_i, and the statement that follows, for $6 \le i \le 8$. However, those guards contain only local variables of client c whence their global correctness is trivial: we did not hide proof obligations!

4.4.2 Absence of disturbance between clients

In this section we prove that two different clients cannot disturb each others assertions. The argument given in section 3.7.2 for the 1-R/R protocol can almost by copied verbatim for the clients in the n-R/R protocol. The only difference is that co-assertions of client c in which $rqi[cw[c, ct']]$ occurs for some $ct' \in \mathbb{N}$ are of the form $srqi(c, cw[c, ct'], ct')$ instead of $srqi(c, s_c, tid[c])$ in the 1-R/R protocol. Again, the reader may check that this set cannot be affected by statements of any client c' such that $c' \neq c$.

4.4.3 Absence of disturbance by clients of assertions of services

In this section we reuse as much as possible of the arguments given in section 3.7.3 in proving that a client c cannot disturb assertions of service s.

Note that we added only local-variables to the n-R/R protocol (in comparison to the 1-R/R protocol). The additional local-variables are all owned by clients and occur only in the owner's assertions, if they occur at all. Hence in showing absence of disturbance by clients of assertions of services, we must deal with the assertions of services listed in the last five rows of table 3.6. Below we treat the same assertions of service s as in the beginning of section 3.7.3;

- For assertions $A13.1$, $A14.3$, $A16.3, 4$, $A17.5, 6$ and $A18.4$ the same argument can be used as in section 3.7.3 because the nature of the modifications made by the clients to $rpi[s]$ for services s is the same as in the 1-R/R protocol.
- Assertions $A13.2$, $A14.4$, $A16.5$, $A17.7$ and $A18.5$ are all similar. Variables occurring in these assertions are cw in the antecedent and $ct2st$ in the consequent: the only potentially disturbing statement in the client component is $S5$. By $A5.1^1$ and $pct2st_0$ the pre-condition to $S5.2$ implies:

  $$\langle \forall ct' \; : \; ct' \in sct(tid[c], sz[c]) \; : \; ct2st[c, ct'] < tid[cw[c, ct']] \rangle.$$

  Hence $S5.2$ is orthogonal to those assertions.
- As for $A17.3$, note that c can only disturb it if (4.1) defined below holds.

  $$c = m_3.fa \;\wedge\; m_3.tid \in sct(tid[c], sz[c]) \tag{4.1}$$

  Assuming (4.1), we have that the ghost-variables provide disjointness;

  $$\begin{array}{ll} (& c = m_3.fa \;\wedge\; m_3.tid \in sct(tid[c], sz[c]) \;\wedge\; A17.3^1 \;\wedge \\ & (\, A5.2 \;\vee\; (A6.3^1 \wedge A6.5^1) \;\vee\; (A7.3^1 \wedge A7.5^1) \;\vee\; (A8.3^1 \wedge A8.5^1)\,) \\ ) & \equiv \; false. \end{array}$$

  Hence we have either disjointness or orthogonality.

The remaining assertions of service s to be dealt with are those in the last four rows of table 3.7. Observe that those assertions can only be disturbed as follows;

- by $S5.2$ if the following is precondition to $S5.2$:

  $$c_s = c \;\wedge\; ct_s \in sct(tid[c], sz[c]) \tag{4.2}$$

- by sub-statements of statement i, $i \in \{6, 7, 8\}$ if the following is precondition to statement $i.2$:

  $$c_s = c \;\wedge\; ct_s = ct_c \;\wedge\; cw[c, ct_s] = s \;\wedge ct_c \;\in CT_i \tag{4.3}$$

  Indeed, the necessity of the first two and fourth conjuncts is trivial. The third conjunct equals assertions $j.1^2_s$, for $j \in \{14, 16, 17, 18\}$.

Observe that for $i \in \{6, 7, 8\}$ we have that execution of any but the first sub-statement of statement i of client c has precondition $ct_c \in CT_{ic}$. By $Ai.3^1$ it follows that after execution of statement $i.1$ we have that the value of $gv(c, ct_c)$ precondition to $Si.2$ equals the value of $gv(c, tid[c])$ at control point i in the client component of the 1-R/R protocol. Assertion $A5$ equals that of the 1-R/R protocol, hence for any ct', $tid[c] \leq ct'$ we have

$gv(c, ct') = (0, 0, 0)$ at $A5$ which again equals the value of $gv(c, tid[c])$ at $A5$ in the 1-R/R protocol. It follows that table 3.8 also gives the possible values of $gv(c_s, ct_s)$ in the n-R/R protocol that must hold if a client c is to execute the first sub-statement of a statement that is non-orthogonal to assertions of service s in the last four rows of table 3.7.

Combining table 3.8 and the last four rows of table 3.7, the last four rows of table 3.9 shows the remaining assertions in the service component to be dealt with. In the remainder of this section we treat those assertions: the proofs are almost identical to those given in section 3.7.3.

4.4.3.1 Assertion 14

From table 3.9 it follows we must show that $S7$ cannot disturb $A14.2^2$. Assuming non-orthogonality, i.e. (4.3), we have widening of $A14.2^2$. In fact, with somewhat work one can show disjointness under the non-orthogonality assumption.

4.4.3.2 Assertions 16 and 17

From table 3.9 it follows we must show that $S8$ cannot disturb $A16.1^1$. For $A17.1^1$ we have the same proof obligation. As $[A17.1^1 \equiv A16.1^1]$, we give the proof once for $A16.1^1$.

In the predicates below we identify ct_s and ct_c with ct as $ct_s = ct_c$ under the non-orthogonality assumption. From table 3.8 we obtain $gv(c, ct) = (1, 1, 0)$. Non-disturbance of $A7.3$ by $S8$ follows from the following (valid) Hoare-triple:

$$\begin{array}{c} \{\, gv(c, ct) = (1,1,0) \,\} \\ S8 \\ \{\, gv(c, ct) = (1,1,1) \,\} \end{array}$$

Which implies (by weakening the postcondition):

$$\begin{array}{c} \{\, A16.1^1 \wedge A8.3^1 \,\} \\ S8 \\ \{\, A16.1^1 \,\} \end{array}$$

4.4.4 Absence of disturbance by services of assertions of clients

In this section we prove that no service s can disturb assertions of client c.

The only non-corollary co-assertion of $A5_c$ is $A5.1_c$. The only potentially non-orthogonal statement of services is $S14_s$ because of the assignment to $RQO[c_s, ct_s]$. $S14_s$ is only non-orthogonal to $A5.1_c$ if $c_s = c$. But then we have disjointness by the ghost-variables: $(A14.1^1 \wedge A5.1^2) \equiv false$.

Recall from section 4.3.1 the distinction between initial and repeated (local-) correctness of the repetition invariants that comprise assertions 6,7, and 8. Because of the use of a unit of work in statement $S8$ (see sections 4.1.5 and A.5), we must only prove initial *global* correctness of the invariants that constitute $A8$. Indeed, because of the atomicity brackets surrounding $S8$, other components cannot disturb $A8_c$ once client c enters (or exits) the atomicity brackets surrounding $S8$. In particular, if client c is at control point $A8$ before it entered the atomicity brackets, we have $CT_9 = \emptyset$ (by $A7.6^1$ and orthogonality of $S7$) whence initial global correctness of $A8.3^2$ and $A8.8^2$ are trivial. Also note that we need not prove global correctness of assertions that are corollaries of other assertions and/or system-invariants.

Combining the above remarks and table 3.5, table 4.6 lists all assertions in the client component of which non-disturbance by any service s remains to be shown.

assertion	*co-assertions*
$A6$	$A6.1^2, A6.3^2$
$A7$	$A7.1^2, A7.3, A7.4^2, A7.7$
$A8$	$A8.1^2, A8.3, A8.4, A8.7$

Table 4.6: Remaining assertions in the client component (1)

Assertions $Ai.1^2$, $6 \leq i \leq 8$, are all similar. The only non-orthogonal statement of service s being $S14$ if $c_s = c$. Using pgv_0^* and $A14.1$ we obtain that at control point $A14$ we have:

$$ct_s \in sct(tid[c], sz[c]).$$

But then we have by $Ai.5^1$ that $S14$ is orthogonal to $Ai.1^2$.

The remaining assertions to be dealt with are given in table 4.7. The table is constructed such that:

- It has three rows: the first two having a *Condition* entry.
- It has four columns.
- Similar assertions appear in the same column. In particular, the last row shows the global variables that occur both in the assertions above it *and* in some statement of the service component: RQO and CD are not present in the last entry of column 2 because they do not appear in statements of service s.
- The *Condition* entry in a row lists the condition necessary for a service s to disturb the assertions in the same row. Note that the conditions are mutually exclusive due to $A6.5^2$, $A7.5^2$ and $A8.5^2$. We only explain why for service s to disturb $A7.7$ or $A8.7$ by executing $S18$, the non-orthogonality condition $c_s = c \ \wedge \ ct_s \in CT_8$ is necessary. For the other assertions/statements this is trivial.

  $S18_s$ can disturb $A7.7$ or $A8.7$ only if for some $m \in M_2$ we have:

  $$\neg m.to \ \wedge \ ct2st[m.fa, m.tid] = tid[m.fa] \ \wedge \ m.fa = s$$

  Then from $A18.1_s^3$ and $A18.5_s$ it follows that for no pair other than c_s, ct_s we have $cw[c_s, ct_s] = s \ \wedge \ ct2st[c_s, ct_s] = tid[s]$. Combined with $A7.6^2$ (or $A8.6$), the non-orthogonality condition follows.
- If one assertion is above another in the same row and column, then the assertions are equal (i.e. $A6.3^2$ and $A7.3^1$).

Assertions			*Condition*
$A6.3^2$ $A7.3^1$			$c_s = c \wedge ct_s \in CT_{7_c}$
$A7.3^2$ $A8.3^1$	$A7.4^2$ $A8.4^1$	$A7.7$ $A8.7$	$c_s = c \wedge ct_s \in CT_{8_c}$
RPO	rqi, rpi	$ct2st, tid$	

Table 4.7: Remaining assertions in the client component (2)

The only statements of services that contain assignments to RPO, rqi, rpi, $ct2st$ or tid are $S13.3$, $S14$ and $S18$. Using $prqi_1$ we obtain that the precondition to $S13.3$ implies $gv(c_s, ct_s) = (1, 0, 0)$. Using this observation, $A14.1^1$, $A18.1^1$ and the values of the ghost-variables given by the assertions in the first column of table 4.7, table 4.8 shows the values of $gv(c_s, ct_s)$ at joint control points given the non-orthogonality condition in the first column. The entries followed by an arrow '$\rightarrow$' give an interpretation of all the entries to the right in the same row. The entries followed by an arrow '$\downarrow$' give an interpretation of all the entries further down in the same column. The abbreviation *cp* stands for 'control point'. The fourth quadrant of the table lists the values of $gv(c, ct)$ at joint control points under the corresponding condition-entry. Again '-' is used to denote disjointness.

		Statement$\rightarrow$	$S13.3_s$	$S14_s$	$S18_s$
		$gv(c_s, ct_s) \rightarrow$	$(1,0,0)$	$(1,0,0)$	$(1,1,1)$
cp$\downarrow$	*Condition*$\downarrow$	$gv(c\ , ct_c) \downarrow$			
$A6_c$	$c_s = c \wedge ct_s \in CT_{6_c}$	$(0,0,0)$	-	-	-
$A7_c$	$c_s = c \wedge ct_s \in CT_{7_c}$	$(1,0,0)$ or $(1,1,0)$	(1,0,0)	(1,0,0)	-
$A8_c$	$c_s = c \wedge ct_s \in CT_{8_c}$	$(1,1,0)$	-	-	-

Table 4.8: Values of $gv(c, ct)$ at joint control points.

Tables 4.7 and 4.8 reveal that it remains to show that neither $A6.3^2$ nor $A7.3^1$ can be disturbed by $S13.3$ or $S14.3$, provided that $c_s = c \wedge ct_s \in CT_{7_c}$ holds.

4.4.4.1 Statement $S13$

$S13$ is orthogonal to $A6.3^2$ and $A7.3^1$.

4.4.4.2 Statement $S14$

First we prove non-disturbance of $A7.3^1$. From tables 4.7 and 4.8 we obtain;

$$(\ A7.3_c^1 \wedge A14.1_s^1 \wedge c_s = c \wedge ct_s \in CT_{7_c}) \Rightarrow gv(c_s, ct_s) = (1,0,0).$$

Global correctness of $A7.3^1$ follows from the following (valid) Hoare-triple

$$\begin{array}{c} \{\ gv(c_s, ct_s) = (1,0,0)\ \} \\ S14_s \\ \{\ gv(c_s, ct_s) = (1,1,0)\ \} \end{array}$$

Which implies (by weakening the postcondition);

$$\begin{array}{c} \{\ gv(c_s, ct_s) = (1,0,0)\ \} \\ S14_s \\ \{\ gv(c_s, ct_s) = (1,1,0) \vee gv(c_s, ct_s) = (1,0,0)\} \end{array}$$

As $A6.3^2 \equiv A7.3^1$, we can use a similar argument for $A6.3^2$ (note that we needed no co-assertions of $A7.3^1$).

4.5 Invariants: correctness

4.5.1 Introduction

In this section we prove correctness of the *system*-invariants shown in table 4.1 marked 'new' or 'reused'. The repetition-invariants shown in table 4.1 are reflected in the assertions: we need not treat those here. The service-components are the same as in chapter 3, as are their annotations. All invariants from the 1-R/R protocol but pgv_0 were reused in the n-R/R protocol. Note that we did not use pgv_0 in proving the correctness invariants other than pgv_0 itself in the 1-R/R protocol. Hence for system-invariants that have been marked as 'reused' in table 4.1, we do not need to prove again that the service components do not disturb them: we refer the reader to section 3.8 instead. Validity of all invariants at control point $A1$ follows directly after applying the substitutions present in $S0$.

4.5.2 Proofs

4.5.2.1 Invariance of I_0 (reused)

As noted in section 4.1.3, the variable s is no longer present in our specification because it is referred to as $cw[c, ct']$ for some client-transaction-id ct'. The dummy variables s and st present in I_0 are superfluous: we may substitute s by $cw[c, ct]$ and st by $ct2st[c, ct']$. Hence it is more convenient to rewrite I_0;

$$\begin{array}{ll}
 & I_0 \\
\equiv & \{ \text{ definition of } I_0,\ I_0(c,s),\ tra. \text{ Dummy transformation and dummy elimination (twice).} \} \\
 & \langle \forall c, ct' \,:\, c \in C \,\wedge\, 0 \le ct' < tid[c] \,\wedge -1 \le ct2st[c,ct'] < tid[cw[c,ct']] \,: \\
 & \quad cd[c,ct'] \equiv cd[cw[c,ct'], ct2st[c,ct']] \\
 & \rangle \\
\equiv & \{ \text{ domain trading } \} \\
 & \langle \forall c, ct' \,:\, c \in C \,\wedge\, 0 \le ct' < tid[c] \,: \\
 & \quad -1 \le ct2st[c,ct'] < tid[cw[c,ct']] \,\Rightarrow\, cd[c,ct'] \equiv cd[cw[c,ct'], ct2st[c,ct']] \\
 & \rangle \\
\equiv & \{ \text{ implication rule, type of } ct2st \} \\
 & \langle \forall c, ct' \,:\, c \in C \,\wedge\, 0 \le ct' < tid[c] \,: \\
 & \quad tid[cw[c,ct']] \le ct2st[c,ct'] \,\vee\, (cd[c,ct'] \equiv cd[cw[c,ct'], ct2st[c,ct']]) \\
 & \rangle
\end{array}$$

We may ignore all statements in the client component preceding $S8.5$ as those only change $cd[c, ct]$ or $cw[c, ct]$ for $tid[c] \le ct$. Indeed, for $S5$ this is trivial, for statement $S6$, $S7$ and $S8$ this follows from $A6.5^1$, $A7.5^1$ and $A8.5^1$ respectively. Because the variables present in I_0 that are modified by client c are local variables of client c, it is advantageous to rewrite I_0 as follows:

$$\begin{array}{ll}
I_0 : & \langle \forall c' \,:\, c' \in C \,:\, I_0(c') \rangle \\
I_0(c) : & \langle \forall ct' \,:\, 0 \le ct' < tid[c] \,: \\
 & \quad tid[cw[c,ct']] \le ct2st[c,ct'] \,\vee\, (cd[c,ct'] \equiv cd[cw[c,ct'], ct2st[c,ct']]) \\
 & \rangle, \quad \text{for } c \in C.
\end{array}$$

As remarked, it suffices to investigate $I_0(c)(\sigma_{8.5_c})$;

$|[\quad I_0(c) \wedge A8.5^1 \wedge A8.6 \wedge A8.7 \wedge A8.8 \wedge \neg B_8$

$\rhd$

$\quad \langle \forall ct' : 0 \le ct' < tid[c] :$
$\quad\quad tid[cw[c,ct']] \le ct2st[c,ct'] \vee (cd[c,ct'] \equiv cd[cw[c,ct'], ct2st[c,ct']])$
$\quad \rangle (\sigma_{8.5_c})$

$\equiv \quad \{$ apply substitution, split off: $ct' \in sct(tid[c], sz[c])$, definition of $I_0(c)\}$
$\quad I_0(c) \wedge$
$\quad \langle \forall ct' : ct' \in sct(tid[c], sz[c]) :$
$\quad\quad tid[cw[c,ct']] \le ct2st[c,ct'] \vee (cd[c,ct'] \equiv cd[cw[c,ct'], ct2st[c,ct']])$
$\quad \rangle$

$\equiv \quad \{$ use $I_0(c)$, $(A8.5^1 \wedge \neg B_8) \Rightarrow (CT_9 = sct(tid[c], sz[c]))$, and $A8.8^1\}$
$\quad \langle \forall ct' : ct' \in sct(tid[c], sz[c]) :$
$\quad\quad tid[cw[c,ct']] \le ct2st[c,ct'] \vee$
$\quad\quad ((\neg (\bigvee m : m \in M_2 : m.to) \wedge cv) \equiv cd[cw[c,ct'], ct2st[c,ct']])$
$\quad \rangle$

$\equiv \quad \{$ use $(A8.8^2 \wedge A8.5^1 \wedge \neg B_8) \Rightarrow (CT_9 = sct(tid[c], sz[c]))\}$
$\quad \langle \forall ct' : ct' \in sct(tid[c], sz[c]) :$
$\quad\quad tid[cw[c,ct']] \le ct2st[c,ct'] \vee \neg(\neg (\bigvee m : m \in M_2 : m.to) \wedge cv)$
$\quad \rangle$

$\equiv \quad \{$ use $A8.6$ (and dummy transformation), de Morgan's law.$\}$
$\quad \langle \forall m' : m' \in M_2 :$
$\quad\quad tid[cw[c, m'.tid]] \le ct2st[c, m'.tid] \vee (\bigvee m : m \in M_2 : m.to) \vee \neg cv$
$\quad \rangle$

$\equiv \quad \{ \vee$- distribution over $\forall$, dummy transformation$\}$
$\quad \langle \forall m : m \in M_2 : tid[cw[c, m.tid]] \le ct2st[c, m.tid] \rangle \vee$
$\quad (\bigvee m : m \in M_2 : m.to) \vee \neg cv$

$\equiv \quad \{ (A8.7 \wedge A8.6) \Rightarrow \langle \forall m : m \in M_2 : ct2st[c, m.tid] = tid[cw[c, m.tid]] \vee m.to \rangle \}$
$\quad true$

$]|$

4.5.2.2 Invariance of $pdt2ct_0$ (new)

Although this invariant is new, it concerns only local variables of clients. Variables occurring in $pdt2ct_0$ are $dtid[C]$, $dt2ct[C]$ and $tid[C]$ in the consequent. Hence the only non-orthogonal statement is $S8$: after applying the substitutions we need only $pdt2ct_0$ itself.

4.5.2.3 Invariance of $pdt2ct_1$ (new)

Although this invariant is new, it concerns only local variables of clients. Variables occurring in $pdt2ct_1$ are $dtid[C]$ in the antecedent and $dt2ct[C]$ and $tid[C]$ in the consequent. Hence the only non-orthogonal statement is $S8$: after applying the substitutions we need only $pdt2ct_1$ itself.

4.5.2.4 Invariance of $pdtid_0$ (new)

As remarked in section 4.2.1, only statements of client c may be non-orthogonal to $pdtid_0(c)$: all statements of any service or a client c', $c' \neq c$, are orthogonal.

From $pdt2ct_{0,1}$ and orthogonality of all assignments but $S8.5_c$ of client c to $pdt2ct_{0,1}$, we obtain as precondition to *each* assignment present in client c (see also, section 4.2.2):

$$dt2ct[c, dtid[c]+1] \le dt2ct[c, dtid[c]] \wedge dt2ct[c, dtid[c]] = tid[c].$$

It follows that the interval $[dt2ct[c, dtid[c]]..dt2ct[c, dtid[c]+1])$ is empty whence all statements preceding $S8.5_c$ are orthogonal to $pdtid_0$ as well. Hence we need only investigate the effect of $S8.5_c$ on $pdtid_0(c)$;

$$
\begin{array}{ll}
[\![& pdtid_0(c) \ \wedge\ A8.8^1 \ \wedge\ A8.5^1 \ \wedge\ \neg B_8 \ \wedge\ pdt2ct_0 \\
\rhd & \\
 & pdtid_0(c)(\sigma_{8.5}) \\
\equiv & \{\text{apply substitution: split off the case } dt = dtid[c], \text{ definition of } pdtid_0(c)\} \\
 & pdtid_0(c) \wedge \\
 & (\forall ct', ct'' \ :\ dt2ct[c, dtid[c]] \leq ct' \leq ct'' < tid[c] + sz[c] \ :\ cd[c, ct'] \equiv cd[c, ct''] \,) \\
\equiv & \left\{ \begin{array}{ll} 1^{st} \text{ conjunct:} & pdtid_0(c) \text{ is orthogonal to } S8.1\ldots4\text{, hence it holds} \\ 2^{nd} \text{ conjunct:} & \text{Use } (A8.5^1 \ \wedge\ \neg B_8) \Rightarrow (CT_9 = sct(tid[c], sz[c]))\text{, } A8.8^1 \text{ and } pdt2ct_0 \end{array} \right\} \\
 & true \\
]\!] &
\end{array}
$$

4.5.2.5 Invariance of $prqi_0$ (reused)

The only free variable occurring in $prqi_0$ is $rqi[S]$ in the antecedent. Hence $S5$, $S7$ and $S8$ are orthogonal.

- $S6$, first alternative: use $c \in C$ (and $cw[c, ct] \in S$): follows from declaration of c and ct, cw respectively.

4.5.2.6 Invariance of $prqi_1$ (reused)

Free variables occurring in $prqi_1$ are $rqi[S]$ in the antecedent and $cw[C]$, $RQO[C]$, $RPO[C]$ and $CD[C]$ in the consequent. Hence $S7$ is orthogonal;

- $S5$: from $A5.1^1$ and $pRQO_1$ we obtain $\langle \forall ct' \ :\ tid[c] \leq ct' \ :\ srqi(c, s, ct') = \emptyset \rangle$.
- $S6$:
  - 1^{st} alternative: apply substitutions and use $A6.3^1$.
  - 2^{nd} alternative: use $A6.4^1$.
- $S8.2$: use $A8.4^1$ and $A8.5^1$.

4.5.2.7 Invariance of $prpi_0$ (reused)

The only variable occurring in $prpi_0$ is $rpi[C]$ in the antecedent. Hence $S5$ and $S8$ are orthogonal.

- $S6$, 2^{nd} alternative: use $c \in C$ and $cw[c, ct] \in S$: follows from declaration of c and ct, cw respectively.
- $S7.3$: widening.

4.5.2.8 Invariance of $prpi_1$ (reused)

Variables occurring in $prpi_1$ are $rpi[C]$ in the antecedent and $tid[S]$, $ct2st[C]$, $rqi[S]$, $RQO[C]$, $RPO[C]$, and $CD[C]$ in the consequent. Hence only $S5$ is orthogonal.

- $S6$:

  - 1st alternative: use $A6.4^1$.
  - 2nd alternative: use $A6.3^1, A6.4^1$ and the fact that the message added to *rpi*$[c]$ is a timeout.

- $S7.3$: widening.
- $S8.2$: use $A8.4^1$ and $A8.5^1$.

4.5.2.9 Invariance of pffi_0 (reused)

The only variable occurring in pffi_0 is $\mathit{ffi}[S]$ in the antecedent. Hence only $S8$ is not orthogonal.

- $S8$: use $c \in C$ (and $cw[c, ct] \in S$): follows from declaration of c and ct, cw respectively.

4.5.2.10 Invariance of pffi_1 (reused)

Variables occurring in pffi_1 are $\mathit{ffi}[S]$ in the antecedent and $RQO[C]$, $RPO[C]$, $CD[C]$, $cd[C]$ and $rpi[C]$ in the consequent. Hence we have as only orthogonal statement $S5$. By respectively $A6.3^1$, $A7.3^1$ and $A8.3^1$ we have that $S6.2$, $S7.3$ and $S8.2$ have a precondition implying $gv(c, ct) \neq (1, 1, 1)$. Using pffi_1, we can derive for $6 \leq i \leq 8$ as precondition to $S6.2$ (either alternative), $S7.3$ and $S8.2$ respectively:

$$\neg\langle\, \exists m, s \ :\ m \in \mathit{ffi}[s] \ :\ m.fa = c \ \wedge\ m.tid \in CT_i \,\rangle\,.$$

Hence $S8.3$ is the only potentially disturbing statement of the client component.

- $S8$: use $A8.3^1$, $A8.4$, $A8.5^1$ and substitution.

4.5.2.11 Invariance of pcw_0 (reused)

Variables occurring in pcw_0 are $rqi[S]$ and $rpi[C]$ in the consequent. Hence statements $S5$ and $S8$ are orthogonal.

- $S6$: either alternative: use $A6.4^1$;
- $S7$: widening.

4.5.2.12 Invariance of $pct2st_0$ (reused)

Variables occurring in $pct2st_0$ are RQO in the antecedent, and $ct2st[C]$ and $tid[S]$ in the consequent. The only non-orthogonal statement is $S6.2$ (either alternative), which only widens $pct2st_0$: use $A6.3^1$.

4.5.2.13 Invariance of $pRQO_1$ (reused)

Variables occurring in $pRQO_1$ are $RQO[C]$ in the antecedent and $rqi[S]$ and $rpi[C]$ in the consequent. By the conjunct $RQO[c, ct] = 0$ in the antecedent we may disregard statements $S7$,$S8$ (use $A7.3^1$ and $A8.3^1$).This leaves only $S6$:

- $S6$: either alternative: use $A6.3^1$ and definition of the statement: widening.

4.5.2.14 Invariance of pcd_0 (reused)

The only free variable occurring in pcd_0 is $cd[S]$ in the consequent whence all statements of client c are orthogonal.

4.5.2.15 Invariance of pCD_1 (reused)

Variables occurring in pCD_1 are $CD[C]$ and $cd[C]$ in the antecedent and cd, $tid[C]$ and $ct2st$ in the consequent. Hence only $S8$ is non-orthogonal.

- $S8$: use $A8.3^1$ and definition of the statement: widening.

4.5.2.16 Invariance of pgv_0^* (changed)

Because we changed pgv_0 to pgv_0^* in the n-R/R protocol, we must also show that no service s can disturb it. Variables occurring in pgv_0 are $RQO[C]$, $RPO[C]$ and $CD[C]$ in the antecedent, and $tid[C]$ and $sz[c]$ in the consequent. Hence only $S5.1$, $S6$, $S8$ and $S14$ are non-orthogonal;

- $S5.1$: use $A5.2$.
- $S6$: apply substitutions; use $A6.5^1$.
- $S8$: note that $S8_c$ is only non-orthogonal to $pgv_0^*(c)$ and that $A5_c$ is postcondition to $S8_c$. As at $A5_c$, $pgv_0^*(c)$ is a corollary of $A5.1_c$, it is immediate that $S8_c$ cannot have disturbed it.
- $S14$: apply substitutions; use $A14.1^1$ (widening).

4.6 Progress (requirement 2)

In this section we prove that progress is guaranteed in the 1-R/R protocol, which covers requirement 2 of the problem statement. The structure of this section is similar to section 3.9. Precise definitions of our progress requirements are given in section 4.6.3. We also prove lemmas 7, 8 and 9 in section 4.6.3. Lemma 7 states that clients cannot deadlock. Lemma 8 states that a service can neither live- nor deadlock in its innermost repetition. Lemma 9 states that a service s can only deadlock on $S13.1$ if no client c ever chooses to perform a transaction with s (in which case it is 'idle'). We conclude that these lemmas indeed cover requirement 2 in section 4.6.4.

We need a few extra assertions in order to prove the lemmas 7, 8 and 9. Those are provided in sections 4.6.1 and 3.9.2 (reused).

4.6.1 Additional assertions in the client component

We need the following additional assertions in the client components to prove that progress is guaranteed in the n-R/R protocol;

4.6.1.1 Assertion 6

- **D6.1:** $\langle \forall ct\ :\ ct \in CT_7\ :\ |srpi(c, cw[c, ct], ct)| = 1 \vee gv(c, ct) = (1, 0, 0)\rangle$
  local correctness:

  - initially: apply substitutions: we obtain $CT_7 = \emptyset$.
  - repeatedly: apply substitutions: use $A6.3^1$, $A6.4^1$ and $A6.5$.

- **D6.2:**

$$\begin{array}{l}\langle\ \ \forall ct\ :\ ct \in CT_7\ : \\ \qquad |srpi(c, cw[c, ct], ct)| = 1\ \vee \\ \qquad (rd[cw[c, ct]]\ \wedge\ rqi[cw[c, ct]] = \{\langle c, cw[c, ct], ct, false\rangle\})\ \vee \\ \qquad ct2st[c, ct] = tid[cw[c, ct]] \\ \rangle\end{array}$$

  local correctness:

  - initially: apply substitutions: we obtain $CT_7 = \emptyset$.
  - repeatedly: for the first two disjuncts: apply substitutions, use $A6.3^1$, $A6.4^1$ and $A6.5$. For the third disjunct: use $\vee-$ weakening.

4.6.1.2 Assertion 7

- **D7.1:** equals $D6.1$
  local correctness:

  - initially: use $D6.1$.
  - repeatedly: apply substitutions: widening.

- **D7.2:** $D6.2$
  local correctness:

  - initially: use $D6.2$.
  - repeatedly: apply substitutions: widening.

4.6.1.3 Global Correctness

No client $c' \neq c$ can disturb $D6$ or $D7$: the argument given in section 4.4.2 also applies to $D6$ and $D7$.

The only statement in the service component non-orthogonal to $D6.1$ is $S14_s$, but only if $c_s = c \wedge ct_s \in CT_7$. Splitting off $ct = ct_s$ from $D6.1$ we find that in such cases the second disjunct is made $false$, while the first is made $true$ by $A14.2^2$. Note that $D6.1 \equiv D7.1$ whence the result applies to $D7.1$ as well.

As $D6.2 \equiv D7.2$, we treat both assertions in a single proof. Assume $CT_7 \neq \emptyset$, and split off any case for ct from $D6.2$ (or $D7.2$):

$$\begin{aligned}
&|srpi(c, cw[c, ct], ct)| = 1 \vee \\
&(rd[cw[c, ct]] \wedge rqi[cw[c, ct]] = \{\langle c, cw[c, ct], ct, false\rangle\}) \vee \\
&ct2st[c, ct] = tid[cw[c, ct]]
\end{aligned}$$

It suffices to show that no service s can disturb the split-off above.

- The first disjunct is only non-orthogonal to $S14_s$, but only if $c_s = c \wedge ct_s \in CT_7$ hold: $cw[c, ct] = s$ can be obtained from $A14.1^2_s$. But then we have disjointness by $A14.2^2_s$.
- The second disjunct is only non-orthogonal to $S13_{cw[c,ct_c]}$ and $S18_{cw[c,ct_c]}$, the last of which only causes widening. $S13_{cw[c,ct_c]}$ makes the second disjunct $false$ while the third disjunct is made $true$. Indeed: m_1 can only be chosen such that $m_1 = \langle c, cw[c, ct_c], ct_c, false\rangle$, i.e. $c_s = c \wedge ct_s = ct$ hold at $A14_{cw[c,ct_c]}$.
- The third disjunct is non-orthogonal to $S13.3_s$, but only if $c_s = c \wedge ct_s \in CT_7$ hold as precondition: in this case the disjunct follows from $A14.1^{2,3}_s$. Also, $S18_s$ is non-orthogonal if $cw[c, ct_c] = s$. From $ct_c \in CT_7$, $A6.2$ and $A6.5^1$ it follows that there exists no ct', $ct' \in CT_7/\{ct_c\}$, such that $cw[c, ct'] = s$. (For $D7.2$: use $A7.2$ and $A7.5^1$ instead of $A6.2$ and $A6.5^1$). By $A18.2$ it must be the case that $c_s = c \wedge ct_s = ct_c$ holds. But then we obtain disjointness by table 3.8.

4.6.2 Additional assertions in the service component

The additional assertions needed in the service component are those given in section 3.9.2. It remains only to prove absence of disturbance of $D16.1$ and $D17.1$ by clients. As $D16.1$ equals $D17.1$, we discuss global correctness of both assertions in a single proof.

Only statements $S6_c$ and $S8_c$ are non-orthogonal to $B16.1$ and $B17.1$, but only if $c = c_s \wedge ct_c = ct_s$. Note that for $i \in 6, 7, 8$, we always have $ct_c \in CT_{i_c}$ as precondition to any sub-statement of statement i_c. From table 3.8 we obtain that control points $A6$ and $A16$ or $A17$ are disjoint under this condition.

For $S8_c$ we have widening of the second disjunct of $D16.1$ and $D17.1$ if $c_s = c \wedge ct_c = tid[c]$.

4.6.3 Absence of Live- or deadlock

The definitions of live- and deadlock given in section 3.9.3 also apply in this section. Note that variant functions were provided in section 4.3.1 for the repetitions of statements $S6$, $S7$ and $S8$. Hence there is no danger in the clients similar to the danger of livelock in the services: for clients there is only danger of deadlock. Table 4.9 summarizes the possible statements that can live- or deadlock, and the corresponding globally correct co-assertions.

Deadlock at:	*Co-assertion:*
$S7.1_c$	$CT_7 \neq \emptyset \ \wedge \ \langle \forall ct \ : \ ct \in CT_7 \ : \ srpi(c, cw[c, ct], ct) = \emptyset \rangle$
$S13.1_s$	$rqi[s] = \emptyset$
$S16.1_s$	$ffi[s] = \emptyset$
Livelock at:	*Co-assertion:*
$S16_s$ and $s17_s$	$sffi(c_s, s, ct_s) = \emptyset \ \wedge \ \neg B_{17}$

Table 4.9: Possibly deadlocking statements and corresponding assertions

Given the definitions above, we further examine a client c that deadlocked. From table 4.9 we obtain this is only possible at $S7.1_c$, and that at this control point, the following holds:

$$CT_7 \neq \emptyset \ \wedge \ \langle \forall ct \ : \ ct \in CT_7 \ : \ srpi(c, cw[c, ct], ct) = \emptyset \rangle \tag{4.4}$$

Using $D7.1$, we obtain;

$$\langle \forall ct \ : \ ct \in CT_7 \ : \ gv(c, ct) = (1, 0, 0) \rangle \tag{4.5}$$

Using (4.5) and $D7.2$, we obtain;

$$\begin{array}{l} \langle \ \ \forall ct \ : \ ct \in CT_7 \ : \\ \qquad (rd[cw[c, ct]] \ \wedge \ rqi[cw[c, ct]] = \{\langle c, cw[c, ct], ct, false \rangle\}) \ \vee \\ \qquad ct2st[c, ct] = tid[cw[c, ct]] \\ \rangle \end{array} \tag{4.6}$$

Lemma 7. *Clients cannot deadlock.*

Proof. Let c deadlock on $S7.1$. By (4.4) we have $CT_7 \neq \emptyset$: split off an arbitrary case ct - $ct \in CT_7$ - from (4.6), and abbreviate $cw[c, ct]$ by s;

$$(rd[s] \ \wedge \ rqi[s] = \{\langle c, s, ct, false \rangle\}) \ \vee \ ct2st[c, ct] = tid[s] \tag{4.7}$$

It suffices to show that (4.7) is always eventually disturbed. We analyze both disjuncts;

- First we show that the second disjunct of (4.7) cannot hold by showing that if it would hold, then service s can not be at either of its control points;
  - Due to $A13.2$, service s cannot be at control point $S13$.
  - The conjunction of $ct2st[c, ct] = tid[s]$ and either $A14.4$, $A16.5$, $A17.7$ or $A18.5$ yields $ct_s = ct \ \wedge \ c_s = c$ if execution of s were at control points $A14$, $A16$, $A17$ or $A18$. In which case:
    - For control point $A14$, by execution of $S14$, s would disturb 4.4, contradicting that c deadlocks.
    - Control points $A16$, $A17$ and $A18$ are disjoint due to

$$((A16.1^1 \vee A17.1^1 \vee A18.1^1) \ \wedge \ (4.5)) \equiv false.$$

- Second, we show that the first disjunct of (4.7) cannot stably hold, for assume that it holds. From $A13.3$, $A14.5$, $A16.6$, $A17.8$ and $A18.6$ it follows that service s can only be at control point $A13$. By $rqi[s] = \{\langle c, s, ct, false \rangle\}$, service s cannot deadlock at $S13.1$ while c is at $A7$. Indeed, it must choose m_1 such that the following holds at holds at $A14$:

$$m_1 = \langle c, s, ct, false \rangle$$

  From $A14.2^1_s$ and $A14.5_s$ it follows that $S13_s$ disturbs first disjunct of (4.7).

□

Lemma 8. *A Service cannot live- or deadlock on in its innermost repetition.*

Proof. Let s be a service that live- or deadlocks in its innermost repetition (i.e. it either deadlocks at $A16$ or it livelocks). Note that by table 4.9 we have that $\mathit{sffi}(c, s, ct_s) = \emptyset$ must then be a system-invariant. Indeed;

- if the service *livelocks*, the system-invariant *equals* the corresponding assertions in 4.9.
- if the service deadlocks at $A16_s$, the system-invariant *follows* from the corresponding assertion in table 3.11.
- no component but s can remove anything from $\mathit{ffi}[s]$. Hence if $\mathit{sffi}(c, s, ct_s) \neq \emptyset$ holds at some point, it would contradict the assumption that s live- or deadlocks.

It follows that it suffices to show that $\mathit{sffi}(c_s, s, ct_s) = \emptyset$ cannot be a system-invariant.

By $\mathit{sffi}(c_s, s, ct_s) = \emptyset$ we have that $\neg B_{17}$ is a globally correct assertion at control point $A17_s$ if s *livelocks* in its innermost repetition. Combined with $D16.1$ and $D17.1$, we obtain that $gv(c_s, ct_s) = (1, 1, 0)$ is also a system invariant if s live- *or* deadlocks in its innermost repetition. Combined with $pgv_0^*(c_s)$, $A16.1^2$ and $A17.1^2$ we obtain that (4.8) defined below is also a system-invariant;

$$cw[c_s, ct_s] = s \ \wedge \ ct_s \in sct(c_s, tid[c_s], sz[c_s]) \ \wedge \ gv(c_s, ct_s) = (1, 1, 0) \qquad (4.8)$$

From $A5.2_{c_s}$, $A6.3_{c_s}$, $A7.3_{c_s}$ and $A8.3_{c_s}$ it follows client c_s must either be at control point $A6$, $A7$ or $A8$ and $ct_s \in CT_7 \cup CT_8$ holds. As by lemma 7, c_s cannot deadlock, and all of its innermost repetitions were proved to terminate in section 4.3, it must eventually reach control point $A5$ again. But $\left(A5.1_{c_s} \wedge (4.8)^3 \wedge ct_s \in \mathbb{N}\right) \equiv false$. Hence (4.8) cannot be a system-invariant, which contradicts that s live- or deadlocks in its innermost repetition. □

Lemma 9. *A service s can only deadlock on $S13.1$ if no client c ever chooses S'_c in $S5.1_c$, such that $s \in S'_c$.*

Proof. By contradiction: let s deadlock on $S13.1$ while some client c chooses S'_c in $S5.1_c$, such that $s \in S'_c$. Then by $A13.3$ and table 4.9 we have at $A13$: $rqi[s] = \emptyset \ \wedge \ rd[s]$. At $A6$, we have for some ct, $ct \in sct(tid[c], sz[c])$: $cw[c, ct] = s$. But execution of $S6.2$ with precondition $cw[c, ct] = s \ \wedge \ rqi[s] = \emptyset \ \wedge \ rd[s]$ disturbs $rqi[s] = \emptyset$. □

4.6.4 Conclusion

In this section we briefly show that lemmas 7, 8 and 9 indeed cover requirement 2 of the problem statement given on page 17. Clients cannot deadlock by lemma 4. The services chosen by a client for a distributed transaction can neither dead- nor livelock by lemmas 5 and 6 respectively. Hence all parties involved in a distributed transaction will always make progress. At some point, all have finished processing their part of the distributed transaction, at which point the distributed transaction has terminated.

Chapter 5

Main Findings

In this chapter we present the main findings in this thesis. Section 5.1 gives a management summary, section 5.2 gives a technical summary. In section 5.3 we present directions for further development of the IFSA application-bus and research concerning transaction integrity in IFSA.

5.1 Management Summary

The business processes within financial institutions typically require a great amount of information processing. Hence information technology is essential to the strategy of ING: it must be properly aligned to its business architecture. This is reflected in a recently introduced, pan-European IT architecture, called the ING Financial Services Architecture (IFSA).

Three fundamental design-principles guided the design of IFSA;

- All intra-domain communication between applications is implemented using services via the central IFSA *application bus*, referred to as *bus* in the sequel. Services communicate by means of sending messages through the bus. The way in which messages are sent is standardized in *communication patterns*: IFSA defines the Request/Reply-pattern (R/R) and the Fire and Forget-pattern (F&F).
- Services are logically distributed over the domains of the business architecture.
- Services are *loosely coupled* [Kay03]: they can connect to, or disconnect from the network as in, for example, the internet.

The transition to a loosely coupled, service based architecture has many advantages and offers great potential for cost reductions. However, it also raises the issue of *distributed transaction integrity*. Basically, distributed transaction integrity can be defined by means of the following requirements;

1. All service-requesters (clients) and services involved in a distributed transaction agree on the outcome of a distributed transaction. That is, either all clients and services commit the distributed transaction, or all services abort.
2. None of the clients or services *deadlocks*. A client or service deadlocks if it reaches a state in which it is stuck forever (unless humans intervene).

The issue of distributed transaction integrity in IFSA is mainly due to the following reasons;

- Because the services are loosely coupled, they may be temporarily down at any time.
- The R/R-pattern offered by the application bus is *lossy*: it has the property that messages sent may be lost.

The most thorough approach to ensuring distributed transaction integrity is the development of *distributed transaction protocols*: sets of rules to which all parties involved in a distributed transaction must adhere. Hence the approach taken in this thesis was to design and formally verify a distributed transaction protocol called the n-R/R protocol.

The main properties of the n-R/R protocol are that;

- It guarantees distributed transaction integrity for distributed transactions that must terminate rather quickly, between a client and any number of services.
- It can to some extent deal with programming errors that lead to malformed service-requests, malformed service-replies, and errors during processing by service-requesters and/or services.
- It is perfectly suited to the loosely-coupled nature of IFSA: services (and service-requesters) may shutdown between transactions without causing the protocol to malfunction.

The design and verification of transaction protocols requires expertise often found only in universities or related institutions such as the Laboratory for Quality Software at the Technische Universiteit Eindhoven. A famous quote from Roger Needham is that *'Security protocols are three line programs that people still manage to get wrong'*. It can be asserted that the same holds for distributed transaction protocols. In fact, in chapter 1 we give two examples of distributed transaction protocols formerly considered within ING that do not guarantee distributed transaction integrity;

Replacing the R/R-pattern by the F&F-pattern: the R/R-pattern offered by the application bus is lossy, which is often a reason for the loss of transaction integrity. The F&F-pattern is not. Hence came the idea to replace R/R-messages by F&F-messages. In section 1.2 we show that;

- Whenever a service disconnects from the bus, it may result in service-requesters being stuck until the service reconnects to the bus. Hence this approach is ill-suited to the concept of loose-coupling.
- Some protocols featuring R/R-messaging can deadlock service-requesters and/or services if the R/R-messages are replaced by F&F-messages.

Compensating Transactions: whenever a service committed a distributed transaction while it should have aborted the transaction, it is instructed to execute a compensating transaction that 'undoes' the compensated-for transaction. In section 1.3.2 we show that;

- Compensation is impossible for transactions that involve *real-actions* such as dispensing money from an automated-teller machine.
- The problem to determine whether or not a compensating transaction actually restores transaction integrity is intractable. In practice this means that the processing power needed to solve the problem may exceed that of modern and future computers.
- The problem to define a compensating transaction that restores transaction integrity is also intractable.

The consequences of using a faulty protocol in IFSA can be dramatic. Often, faults cannot be repaired (for example, because transactions involve real-actions), or, repairing them may result in huge amounts of man- and machine-hours, possible financial burden, and a loss of goodwill. It can be expected that the costs incurred by the formal verification of a protocol outweigh the costs incurred by the use of an incorrect protocol, especially in an architecture as large as IFSA. Indeed, once a protocol has been designed and verified, it can be reused throughout IFSA without any added costs.

Although the n-R/R protocol is suited to a wide range of applications, there are also applications to which it is not. Hence it may be considered to develop and verify additional transaction-protocols to cover a wider range of applications. In section 5.3 we present some directions to this end.

5.2 Technical Summary

In this section we provide a technical summary. This section complements the management summary given in section 5.1: we do not repeat definitions or abbreviations already given there. Sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4 summarize chapters 1, 2, 3 and 4 respectively. In section 5.2.5 we summarize how the requirements of the problem-statement given in chapter 1 are met.

5.2.1 Chapter 1

In chapter 1 we define IFSA as a loosely-coupled network of clients and services that communicate by means of the IFSA-application-bus. In order to be able to reason about distributed transaction integrity, we assumed each client and service to have a local database. Clients and services can only read or update their own database, by means of local database transactions. The distributed database was defined as the multiset of all local databases. We explain the two messaging-patterns offered by the bus in section 1.1.2.

The F&F-pattern provides for one-way, unordered, reliable, asynchronous communication, and features only F&F-messages. The only scenario possible when sending a F&F-message from a client to a service was given in figure 1.3 on page 15, the unordered nature of F&F-messaging was demonstrated in figure 1.4 on page 15. F&F-messaging is expensive in the sense that the associated overhead is much greater than that of R/R-messaging.

The R/R-pattern provides for two-way, unordered communication and distinguishes between requests and replies. Typically a client sends a request to a service and expects a reply from the service. However, it is also possible that the client receives a timeout from the bus instead of a reply from the service. Hence the R/R-pattern was said to be *lossy*. We reduced the possible scenarios for R/R-messaging to the three scenarios given in figure 1.2 on page 14.

In defining the problem statement, we defined an example distributed-update-transaction using only R/R-messaging on the basis of the scenarios given in figure 1.2. In particular, we assumed the request in each scenario of figure 1.2 to contain an instruction for the service to perform an update to its local database by means of a local database transaction, and the reply to contain information with which the client must update its database, also, by means of a local database transaction. The impossibility for the client to discriminate between scenarios 2 and 3 of figure 1.2, and for the service to discriminate between scenarios 1 and 3 formed the basis for our problem-statement.

However problematic in distributed update-transactions, it was argued that R/R-messaging also has important benefits over F&F-messaging in distributed transactions between one client and several services. Indeed, replacing R/R-messaging by F&F-messaging was shown to introduce deadlock in some cases, or, it proved to be ill-suited to the loosely-coupled nature of IFSA.

Hence the exact problem-statement was to find a solution for the problems connected to using R/R-messaging when performing distributed update-transactions between one client and any number of services. The solution should respect the loosely-coupled nature of IFSA, use only functionality provided by the bus, and ensure that the four requirements listed in the first two columns of table 5.1 are met. (We explain the other columns of table 5.1 in the sequel.)

Requirement		is covered by	in section(s)
1	*Each distributed transaction always ends such that all parties involved commit their local database transaction, or all parties involved abort their local database transaction*	invariants I_0 and $pdtid_0$	4.2.1 and 4.5
2	*Distributed transactions always terminate*	lemmas 7, 8 and 9	4.6.4
3	*It is impossible that the transaction system enters a state wherein each transaction is always aborted*	remark	2.5.2
4	*A good balance exists between price and performance of the protocol*	remark	2.5.3

Table 5.1: Requirements traceability matrix

We considered the following candidate-solutions to the problem statement;

- Resending of requests by the client until a reply is received (instead of a timeout), discussed in section 1.3.1. It was argued that this approach is ill-suited to the loosely-coupled nature of IFSA.

- Compensating transactions, discussed in section 1.3.2, was a solution formerly considered within ING until we showed it to be problematic. The main idea is that whenever a party in a distributed transaction committed its local database transaction while another party aborted, the party that committed is instructed to perform a compensating transaction that undoes the effects of the compensated-for transaction.

  The discussion was largely based on the formal framework provided in [KLS90] for compensating transactions on local databases. The paper defines soundness of histories of transactions on (local-)databases as the main integrity criterion. We ourselves showed the decision-problems of determining soundness after compensation ($HistSound$), and, defining a compensating transaction that ensures soundness ($ComSound_{1*}$), to be intractable in theorems 1 and 2 respectively. As those problems would have to be solved in many applications that rely on compensating transactions to achieve distributed transaction integrity, we rejected compensating-transactions as a general solution to our problem-statement. It was also noted that reliably instructing a client or service to perform a compensating transaction is problematic using only R/R-messaging.

- Allowing additional F&F-messaging, discussed in section 1.3.3. Given the drawbacks of the other proposed solutions, allowing F&F-messaging in *addition* to R/R-messaging seemed the only viable solution left. In chapter 2 we present a protocol that uses such additional F&F-messages: the n-R/R protocol. It allows distributed transactions between 1 client and n different services, $1 \leq n$. The integer n is called the *size* of the distributed transactions.

5.2.2 Chapter 2

In chapter 2 we present the n-R/R protocol, the main properties of which are that;

- It requires n requests, n replies and n F&F-messages for distributed transactions with size n.
- It suits the loosely-coupled nature of IFSA: it allows clients and services to disconnect from the bus in between transactions without causing the protocol to malfunction.
- The decision to commit or abort a distributed transaction is taken by means of a voting process: a distributed transaction can only be committed if the client and all services involved unanimously vote to do so, otherwise it is aborted.
- The protocol can to some extent deal with programming errors that lead to malformed requests, malformed replies, or errors during processing by the business applications run by clients or services.
- The protocol is similar to the 2-phase-commit protocol (see for example: [SKS98]). The main difference between the 2-phase-commit protocol and the n-R/R protocol are;
  - The 2-phase-commit protocol requires $3n$ messages to be sent *in addition* to the messages needed to perform the actual transaction. The messages contain only votes or commit/abort-instructions. The n-R/R protocol requires $3n$ messages *in total.*
  - The n-R/R protocol is specified in terms of the communication-primitives offered by the bus. We use those primitives in a very specific manner. In section 2.5.2 we showed that the n-R/R protocol can deadlock clients and services if we replace all R/R-messaging by F&F-messaging. The protocol thus obtained is also similar to the 2-phase-commit protocol. This shows that implementing a known protocol in terms of bus-primitives is not a trivial matter!
- The protocol is intended for distributed transactions that terminate rather quickly (in the range of a few seconds).

The client- and service-programs of the n-R/R protocol are discussed in sections 2.2 and 2.3 respectively. In section 2.4 we give some example scenarios of executions of the protocol. Section 2.5 discusses mostly implementation details, we also provide arguments that show that requirements 3 and 4 are met by the n-R/R protocol (see the last two rows of table 5.1). Section 2.5.5 discusses the limitations of the protocol.

5.2.3 Chapter 3

In chapter 3 we verify that requirements 1 and 2 of the problem statement are met by the 1-R/R protocol: a special case of the n-R/R protocol that only allows transactions with size 1. The main motivation for first examining this special case was the complexity of our proof requirements.

In section 3.3 we give a formal model of the messaging-primitives offered by the application bus in the Guarded Command Language (GCL, [FvG99]). Note that this model may also be used to design other protocols.

The client- and service-programs given in chapter 2 are translated to the GCL in sections 3.4.2 and 3.4.3 respectively. Requirement 1 of the problem statement is translated to the system-invariant I_0 in section 3.5.1. Sections 3.6, 3.7 and 3.8 are devoted to proving properties of the protocol that enable us to prove that the system invariant I_0 is indeed maintained by the 1-R/R protocol. In section 3.9 we prove that the 1-R/R protocol meets requirement 2.

5.2.4 Chapter 4

In chapter 4 we verify that requirements 1 and 2 of the problem statement are met by the n-R/R protocol for *any* positive n. In doing so we reuse as much as possible from the results in chapter 3. The client-program of the 1-R/R protocol given in section 3.4.2 is extended to allow for transactions with any positive size n in section 4.1.4. The service program given in section 3.4.3 is reused without modification.

Requirement 1 of the problem statement is translated to the reused system-invariant I_0 and the new system-invariant $pdtid_0$ in section 4.2.1. Sections 4.3, 4.4 and 4.5 are devoted to proving properties of the protocol that enable us to prove that the system invariants I_0 and $pdtid_0$ are indeed maintained by the n-R/R protocol. In section 4.6 we prove that the n-R/R protocol meets requirement 2.

5.2.5 Conclusion

Table 5.1 is a requirements-traceability matrix. Entries in the last two columns show exactly where and how it is remarked or formally proved that a requirement given in the first two columns is met. For requirements 1 and 2 we gave a formal proof. For requirement 3 we provide guidelines in section 2.5.2 that if followed, lead to it being met in all probability. The efficiency of the n-R/R protocol is discussed in section 2.5.3, it is also argued there that requirement 4 is met.

Recall from section 5.2.2 that the n-R/R protocol also respects the loosely-coupled nature of IFSA, and that it only uses the functionality offered by the IFSA-bus. Hence we consider all requirements of the problem-statement to be met under the required preconditions.

5.3 Directions for further research and development

In this section we present some directions for further development of the IFSA application-bus and research concerning transaction integrity in IFSA.

The following directions concern further development of the IFSA application-bus;

- The method for receiving requests or F&F-messages `IFSAGetMessage` (see section A.4.1), returns any available request or F&F-message. The BAI provides no functionality that lets a service specifically receive a request instead of a F&F-message, or vice versa. This could be a useful feature in future versions of the bus. The absence of this feature led to the requirement in the n-R/R protocol that a service has two connections (one only for R/R-messaging and one only for F&F-messaging, see section 2.3.2).

- As discussed in section 2.3.2, the BAI provides no functionality that lets a service identify the sender of messages even though this information is available at the layer below the BAI. This information is often needed for distributed transaction protocols to function. There are two solutions to this;

  - Expose the sender of a message by means of a new BAI-method.

  - Implement distributed transaction protocols at the layer below the BAI and expose only the methods needed for the distributed transaction protocols to operate by means of new BAI-methods.

The following directions concern further research concerning transaction integrity in IFSA;

- It might be useful to extend the n-R/R protocol to allow for nested transactions: transactions in which services must also act as service-requester (client). We provide some sketches to this end below.

  An example scenario for a nested distributed transaction is given in figure 5.1. Two services are involved in the scenario: the client transacts with service 1, service 1 must transact with service 2 in order to be able to send a reply to the client. The messages exchanged between the client and service 1 are exactly those of a size-1 distributed transaction of the n-R/R protocol. The same holds for the messages exchanged between services 1 and 2, the difference being that service 1 acts as *client* in its communication with service 2.

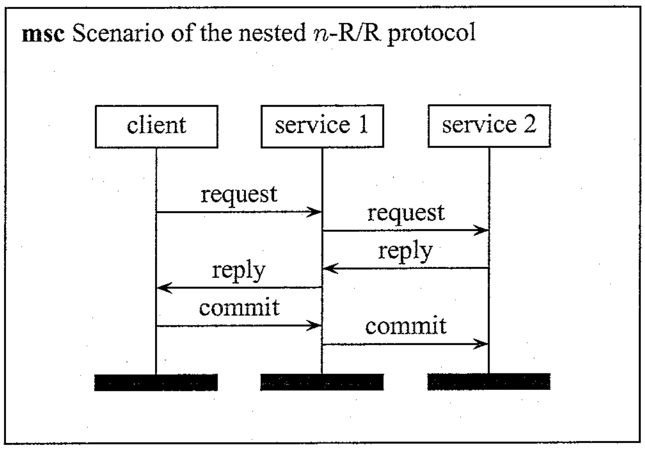

Figure 5.1: Scenario of the nested n-R/R protocol

  Only the service-program of the n-R/R protocol given in section section 2.3 would have to be adapted to allow for nested transactions. A flow diagram of the adapted service-program is given in figure 5.2. The interpretation of the grey boxes is the same as the boxes with corresponding labels in the client program of the n-R/R protocol (see section 2.2 and figure 2.1). The interpretation of the white colored boxes is the same as the boxes with corresponding labels in the service program of the n-R/R protocol (see section 2.3 and figure 2.2).

  Some remarks are in place. First of all, it is evident that the risk of receiving a timeout (or reply containing a vote to abort) increases with the level of nesting. Secondly, the informal description of the nested-n-R/R protocol given above should not be taken as a specification but merely as a starting point for further research!

- It may be desirable to develop other distributed transaction-protocols in addition to the n-R/R protocol because some applications may not be suited to the n-R/R protocol. We explain this by giving an example. Recall from section 2.5.2 that in the n-R/R protocol, the variable `MAX_TIMEOUT` roughly indicates the maximum amount of time a transaction may take. It was argued in section 2.5.2 that this amount should not be set too high. Some applications may call for *long-lived distributed transactions*: distributed transactions that can take hours or even days to complete. It is evident that the n-R/R protocol is not suited to such applications.

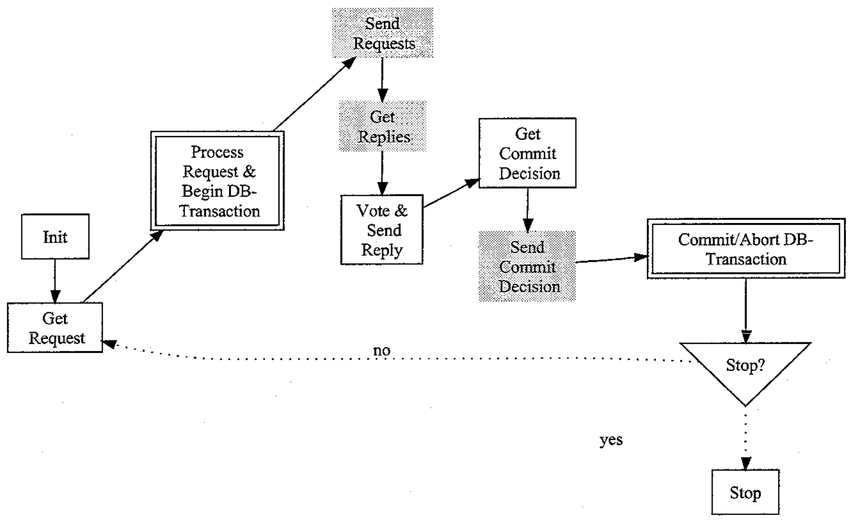

Figure 5.2: Service-program of the nested n-R/R protocol: flow-diagram

- The n-R/R protocol has been designed for single-instance, single-threaded services. It is however not difficult to extend the protocol such that it can function with multi-instance and/or multi-threaded services: services that can process multiple transactions simultaneously. Verifying that a multi-threaded version of the n-R/R protocol maintains transactional-integrity as required by our problem-statement given on page 17 seems challenging though.
- In section 2.5.4 we gave some guidelines for dealing with exceptions caused by malfunctioning of the bus. However, we did not prove that following these guidelines restores distributed transaction integrity. It might also be useful to extend and verify the protocol such that it allows clients and/or services to recover from crashes during a transaction.
- In section 2.5.2 it was shown that replacing all R/R-messaging by F&F-messaging in the n-R/R protocol introduces the danger of deadlock. It would be interesting to investigate whether or not the functionality offered by our protocol can be provided by a protocol that uses only and at most $3n$ F&F-messages - or generally, only and at most $3n$ reliable, asynchronous and unordered messages - in distributed transactions with size n, without resorting to the use of timeouts.

Appendix A

The business application interface

A.1 Introduction

The business application interface (BAI) of the IFSA bus is the only interface accessible to the business-application programmer. Because of its importance to this project and for the sake of unity of this document, some parts of the interface that concern communication are treated in detail in this appendix. The sections that follow also explain some terminology and concepts used throughout the bus documentation.

Note that the BAI is available for multiple programming languages and platforms such as C, Java, .NET, CICS/COBOL, etc., on UNIX, Windows, etc. The differences between these implementations are minor. This appendix is based on the C-interface described in [it03].

Section A.2 discusses basic concepts and how connections with the bus can be made. Section A.3 explains how messages can be sent, section A.4 explains how messages can be received. Section A.5 discusses *units of work*. Section A.6 discusses the various faults and exceptions that may occur in communication.

A.2 Common BAI concepts

Every BAI-method returns a *completion-code*, the corresponding BAI type is the enumeration type `IFSARETCODE` = {`IFSACC_OK`, `IFSACC_WARNING`, `IFSACC_FAILED`}. Associated with a completion-code is a *reason-code*, the corresponding BAI type is `IFSAREASON`. It is listed below what the meaning of the different completion-codes is;

- `IFSACC_OK`: the invocation completed successfully, the corresponding reason-code has a default value `IFSARC_DEFAULT`.

- `IFSACC_WARNING`: the invocation completed but raised a warning, the corresponding reason-code contains a description of the warning. The warning must be specifically handled in most cases. The reason-codes differ per method in this case.

- `IFSACC_FAILED`: a rather serious failure occurred, the corresponding reason-code contains a description of the failure.

If warnings can be returned by a method they are discussed. Also failures that can be recovered from are discussed. Irrecoverable failures are not discussed in detail.

A.2.1 Connections and Service slots

Each application that wants to use the bus must connect to it first, once the connection is no longer needed it must be destroyed in order to release the resources associated with the connection. The corresponding BAI-methods are `IFSAConnectEx` and `IFSADisConnect` respectively. A connection is represented by a connection-handler of the type `IFSAHCONN`.

For each connection, it is possible to create an arbitrary number of *service slots*, that are to be destroyed when not needed anymore. The corresponding BAI-methods are `IFSAAllocateServiceSlot` and `IFSAReleaseServiceSlot` respectively. Each service slot enables communication with a single service that is to be specified upon creation. So if communication is needed with n services, n service slots are needed. The BAI type for service-slot-handlers is `IFSAHSRVSLOT`.

A.2.2 (Un)registering a service

The bus *repository* is an internal store (invisible to the programmer) that keeps track of the services that are accessible through the bus and their addresses. Services must be registered before they can be used and unregistered if unavailable, in order to register a service, a connection is needed. The corresponding BAI-methods are `IFSARegisterService` and `IFSAUnRegisterService` respectively. The BAI type for services is `IFSAHSRV`.

A.3 Sending messages or requests

The IFSA specification provides two *patterns* to exchange messages. With each pattern are associated a certain message type and means to send and receive messages. Both patterns provide unordered communication only: there is no guaranteed order in which messages are received.

A.3.1 The 'Fire and Forget' pattern

The Fire and Forget (F&F) messages, simply referred to as *messages* in the sequel if no confusion is possible may have a size of up to 4 megabytes. Their BAI type is `IFSAHMSG`. There is a (very) high probability that the message arrives, there are however (extreme) cases where this is not so, we deal with those cases at a later stage. There is no such thing as a connection associated with the subsequent sending of multiple messages: the messages arrive at some time at the destination service.

A.3.1.1 Creating F&F-messages

Messages are created and destroyed by means of the BAI-methods `IFSACreateFireAndForget` and `IFSADestroyMessage` respectively, the latter of which simply releases the memory associated with a message (this method must also be used for destroying replies, treated later).

Table A.1 gives the details for `IFSACreateFireAndForget`. A value 'in' in the column 'Input/Output' means that the argument in the corresponding row is an input parameter, a value 'out' an output parameter. The value 'in/out' means that before invoking the method, a variable of the corresponding type must have be declared and passed as an argument. This variable is then modified (or assigned a value) by the invocation. These conventions are used throughout this appendix.

Argument	Input/Output	Type	Description
`serviceSlot`	in	`IFSAHSRVSLOT`	The service-slot handler, determines the destination
`message`	in/out	`IFSAHMSG`	The message handler
`payload`	in	`char[]`	The *payload* (contents) of the message
`reasonCode`	int/out	`IFSAREASON`	A description of the failure or warning (if any)
`completionCode`	out	`IFSARETCODE`	Completion-code

Table A.1: The `IFSACreateFireAndForget` method

A.3.1.2 Sending F&F-messages

Sending of messages is done by means of the BAI-method `IFSASend`, the details of which are given in table A.3.1.2.

Argument	Input/Output	Type	Description
`message`	in/out	`IFSAHMSG`	The message to be sent
`reasonCode`	int/out	`IFSAREASON`	A description of the failure or warning (if any)
`completionCode`	out	`IFSARETCODE`	Completion-code

Table A.2: The `IFSASend` method

The different values for `completionCode` have the following interpretation;

- `IFSACC_WARNING`: with one of the following reason-codes;
  - `IFSARC_PAYLOAD_TOO_LARGE`: the message was sent although the payload is too large; the application contains a programming error.

Note that `IFSASend` must also be used for sending requests or replies (discussed in the sequel).

A.3.2 The 'Request/Reply' pattern

The 'Request/Reply' (R/R) pattern distinguishes *requests* and *replies*. A request is sent to a destination service after which a unique reply from the destination service is expected. One can see this as a connection between the client and service that lasts for two communications. Replies are identified by their corresponding request in the BAI. In contrast with the F&F pattern, upon creating a request, a parameter `timeOut` must be specified. There exist internal configuration parameters `MIN_TIMEOUT`, `MAX_TIMEOUT`, (`MIN_TIMEOUT` < `MAX_TIMEOUT`), that cannot be changed by the programmer, that specify the minimum and maximum allowed respectively. If the value of `timeOut` exceeds these boundaries it is changed to equal the closest boundary. When a request is sent, the system internally calculates the following parameters;

- `systemTime` ='the system time of the bus upon *sending* the request'.
- `absoluteTimeout` = `systemTime` + `timeOut`,

- `expiryTime` = `F(timeOut)`: where `F` is a monotonously increasing function, such that `timeOut` < `F(absoluteExpiryTime)`,
- `absoluteExpiryTime` = `systemTime` + `expiryTime`.

After `absoluteExpiryTime`, the reception of a reply is impossible, in some cases this is already so after `absoluteTimeout`. The exact meaning of `absoluteTimeout` and `absoluteExpiryTime` is explained when discussing the reception of requests. Requests and replies may have a payload of up to 32 kilobytes, their BAI types are equal to that of F&F-messages.

A.3.3 Creating requests

Requests are created and destroyed by means of the BAI-methods `IFSACreateRequest` and `IFSADestroyMessage` respectively, the latter of which simply releases the memory associated with a message (this method must also be used for destroying replies, treated in section A.3.4). Table A.3 gives the details for `IFSACreateRequest`.

Argument	Input/Output	Type	Description
`serviceSlot`	in	IFSAHSRVSLOT	The service-slot handler, determines the destination
`timeOut`	in	`int`	The timeout value
`request`	in/out	IFSAHMSG	The request handler
`payload`	in	`char`[]	The payload of the request
`reasonCode`	int/out	IFSAREASON	A description of the failure or warning (if any)
`completionCode`	out	IFSARETCODE	Completion-code

Table A.3: The `IFSACreateRequest` method

A.3.4 Creating replies

Replies are created and destroyed by means of the BAI-methods `IFSACreateReply` and `IFSADestroyMessage` respectively. Table A.4 gives the details for `IFSACreateReply`.

Argument	Input/Output	Type	Description
`request`	in/out	IFSAHMSG	The request for which the reply is
`reply`	in/out	IFSAHMSG	The reply
`payload`	in	`char`[]	The payload of the reply
`reasonCode`	int/out	IFSAREASON	A description of the failure or warning (if any)
`completionCode`	out	IFSARETCODE	Completion-code

Table A.4: The `IFSACreateReply` method

A.3.5 Sending requests and replies

Sending of requests and replies must be done as sending F&F-messages because their BAI types are equal.

A.4 Receiving messages

The following subsections treat the methods for receiving F&F-messages, requests and replies.

A.4.1 The `IFSAGetMessage` method

`IFSAGetMessage` is a BAI-method to receive either requests or F&F-messages. The method is intended for services. This method can get any message addressed to the specified connection. The receiving service must inspect a received message in order to determine the type of the message (either a reply or a F&F-message). The details of `IFSAGetMessage` are given in table A.5.

Argument	Input/Output	Type	Description
`connection`	in	`IFSAHCONN`	The connection to use
`waitDelay`	in/out	`int`	Wait time in milliseconds if no message is available
`message`	in/out	`IFSAHMSG`	The received F&F-message or reply
`payload`	in/out	`char[]`	The payload of the message or reply
`reasonCode`	int/out	`IFSAREASON`	A description of the failure or warning (if any)
`completionCode`	out	`IFSARETCODE`	Completion-code

Table A.5: The `IFSAGetMessage` method

The values of `reasonCode` that can be returned have the following interpretations;

- `completionCode` = `IFSACC_WARNING` $\wedge$ `reasonCode` = `IFSARC_NO_MESSAGE_AVAILABLE`: No message was received within `waitDelay` milliseconds.
- `completionCode` = `IFSACC_WARNING` $\wedge$ `reasonCode` = `IFSARC_POISON_MESSAGE`: There was a F&F-message waiting, but that message had already been offered to the service several times within a unit of work that was aborted too many times (see section A.5). The returned `message` is valid whence all context information can be retrieved. The payload however is `NULL`.

A.4.2 The `IFSAGetReply` method

The `IFSAGetReply` method can be used to receive a pending reply corresponding to a specified request. The details of `IFSAGetReply` are given in table A.6. The method may take up to `timeOut` milliseconds to return, where `timeOut` is the parameter specified upon invocation of `IFSCreateRequest` for creating the corresponding request.

The value `AbsoluteTimeOut` associated with a reply determines until when and invocation of `IFSAGetReply` may wait for a reply. If the method invocation must wait and the reply has not arrived at `AbsoluteTimeOut`, then the reply times out. Once a reply has timed out, it cannot be received anymore, it is deleted by the bus. Irrespective of the BAI-method used for receiving a reply, a reply is deleted by the bus if it has not yet been received or timed out (and thus subsequently deleted) at `AbsoluteExpiryTime`. If `IFSAGetReply` is invoked after `AbsoluteTimeOut` but before `AbsoluteExpiryTime` while the reply is available, the reply can still be received. Note that the exact waiting behavior of `IFSAGetReply` is invisible to the application programmer.

The possible values for `completionCode` must be interpreted as follows;

Argument	Input/Output	Type	Description
`request`	in	`IFSAHMSG`	The request for which a reply is expected
`reply`	in/out	`IFSAHMSG`	The reply
`payload`	in/out	`char[]`	The payload of the reply
`reasonCode`	int/out	`IFSAREASON`	A description of the failure or warning (if any)
`completionCode`	out	`IFSARETCODE`	Completion-code

Table A.6: The `IFSAGetReply` method

- `completionCode` = `IFSACC_WARNING`: a timeout occurred, `reply` is not a valid reply, the payload is `NULL`. Note that any failure from the sending application or the bus leads to a timeout.

A.4.3 The `IFSAGetAnyReply` method

The `IFSAGetAnyReply` method can be used to receive any pending reply for the specified connection. The details of `IFSAGetAnyReply` are given in table A.7. The method may take up to `timeOut` milliseconds to return, where `timeOut` is the *minimum* over all values of `timeOut` specified upon invocation of `IFSCreateRequest` for the associated pending requests.

Argument	Input/Output	Type	Description
`connection`	in	`IFSAHCONN`	The connection to use
`request`	in/out	`IFSAHMSG`	The request for which a reply or timeout is returned
`reply`	in	`IFSAHMSG`	The reply returned
`payload`	in	`char[]`	The payload of the reply
`reasonCode`	int/out	`IFSAREASON`	A description of the failure or warning (if any)
`completionCode`	out	`IFSARETCODE`	Completion-code

Table A.7: The `IFSAGetReply` method

The possible values for `completionCode` and `reasonCode` must be interpreted as follows;

- `completionCode` = `IFSACC_FAILED` $\wedge$ `reasonCode` = `IFSARC_NO_PENDING_REQUEST`: the invoking application sent no requests for which a reply (or timeout) was not yet returned.

- `completionCode` = `IFSACC_FAILED` $\wedge$ `reasonCode` = `IFSARC_TIME_OUT`: a timeout occurred. A request that was sent previously is not collected in time by the service, it is not processed in time, or the corresponding reply was not received in time. Note that if the sender of a reply crashes just before sending, or the bus crashes before the reply is collected, the result is also a timeout. The returned request is valid, which makes identification possible of the reply that timed out.

If several replies are expected, only one reply or timeout is returned by `IFSAGetAnyReply` during each invocation.

A.5 Units of work

It is possible to send or receive multiple F&F-messages in a *unit of work* (UOW). A UOW has an effect on F&F-messaging similar to that of a database transaction on a database. Database transactions are used to group a number of actions that modify the database into an atomic action that is either performed (if the transaction is comitted), or not (if the transaction is aborted). A unit of work is used to group a number of send- and receive actions of F&F-messages of a single client or service into an atomic action that is either performed (if the UOW is comitted), or not (if the UOW is rolled back).

A UOW is started by means of the BAI-method `IFSABeginUOW` and ended by either `IFSACommitUOW` or `IFSARollbackUOW`. The scope of a UOW is that of the connection. Any request or reply sent or received is never part of a UOW, although requests and replies may be sent or received independent of the actions of a UOW. F&F-messages sent or received within a UOW are either all sent and received (in case the UOW ends with `IFSACommitUOW`), or none of them are (in case the UOW ends with `IFSARollbackUOW`).

The messages that were received within a UOW that is rolled back, are kept by the bus for retrieval at a later stage. Messages that that were sent within a UOW that is rolled back are never offered to the destination service: the bus simply deletes the message. If a F&F-message is received outside the scope of a UOW, it can not be received again.

It is possible to configure a parameter called `MaxBackout` within the bus BAI (this is invisible to the programmer) that specifies how many times the retrieval of a F&F-message may be rolled back. If this value is exceeded for a F&F-message, the message becomes a *poison*-message. Upon receiving a poison-message, a warning is raised (see section A.4.1).

Argument	Input/Output	Type	Description
`connection`	in	`IFSAHCONN`	The connection to use
`reasonCode`	int/out	`IFSAREASON`	A description of the failure or warning (if any)
`completionCode`	out	`IFSARETCODE`	Completion-code

Table A.8: The `IFSABeginUOW`, `IFSAEndUOW`, `IFSARollbackUOW` methods

Table A.8 lists the arguments for `IFSABeginUOW`, `IFSAEndUOW` and `IFSARollbackUOW`. If `IFSABeginUOW` is invoked if a UOW is already active, then the method returns with `completionCode` = `IFSA_FAIL` $\wedge$ `reasonCode` = `IFSARC_UOW_IN_PROGRESS`; the active UOW must be committed or rolled back first.

A.6 Considerations

In this section we discuss the various faults and exceptions that may occur in communication;

- It is possible F&F-messages arrive more than once because of crashing and recovery of the bus, independent of whether or not UOWs are used.
- It is possible that requests, replies or F&F-messages do not arrive at their destination service because one of the following reasons;
  - A failure or crash occurred in a business application or a disaster struck the infrastructure hosting the bus. The details are listed in table A.9.
  - A request expired. The details are listed in table A.10 .
  - A message was rejected by the bus. The details are listed in table A.11.

Note that F&F-messaging can be considered reliable if units of work are used: only extreme disasters or severe programming-errors may prevent a F&F-message from reaching its destination.

Message type	Failure point	Explanation
F&F	Failure during `IFSASend`	It may be expected that the sender notices this by inspecting completion-codes and that it sends the message again.
F&F	Crash of a part of the bus	Disaster recovery plans have been put in place to recover from disasters. Depending on the magnitude of the disaster, messages in transit can be lost.
F&F	failure during `IFSAGetMessage`	If the receiver crashes during the invocation of `IFSAGetMessage`, the message is lost. This can be prevented by using a UOW.
Request	failure during `IFSASend`	See under F&F
Request	crash of a part of the bus	This depends on which part of the bus crashed. The infrastructure of the bus is complex and there are many possibilities.If the request is indeed lost by the crash, then `IFSAGetReply` returns with an error. Which specific error is dependent on the nature of the crash. `IFSAGetAnyReply` shows no special behaviour. If a reply to this request already exists, it is ignored. Anyway, no warning or failure is returned for this request.
Request	failure during `IFSAGetMessage`	If the receiver crashes during the invocation of `IFSAGetMessage`, the message is lost. This cannot be prevented. Eventually, the sender receives a time-out when trying to receive the reply.
Reply	-	Basically the same as under request: the only difference being that the receiver has received the request in this case and has done some work.The sender however cannot know whether the failure occurred before or after the receiver received the request.

Table A.9: Undelivered messages due to system failure

Message type	Explanation
Request	It takes longer than the specified expiry value to reach the service (e.g.: an extremely short value for `timeOut` is specified). In this case the intended receiver cannot notice anything: the request is never offered. The sender receives a timeout when the reply is received.
Reply	The time lapse between the invocation of the `IFSASend` routine and the corresponding `IFSAGetReply` or `IFSAGetAnyReply` is larger than the specified expiry. Either the sender took too long to create the reply or the path for the reply or the request was too long, that makes no difference for the result. The sender receives a timeout when the reply is received.

Table A.10: Undelivered messages due to expiry

Message type	Failure point	Explanation
Any	Message header	The header of the message is not IFSA compliant. The intended receiver is not informed of this. In case of a request, the sender eventually receives a timeout.
Request or F&F	Service	The service addressed by this message is not registered by the receiver. Neither are the receiver nor the sender informed of this. In case of a request, it eventually times out.
Reply	Coupled request	The reply is not coupled to a pending request (for example because the request is already destroyed). The client that originally sent the request does not notice the rejected reply.
Any	Module	Some module used by the BAI encountered an error while the message was received. The application receives a notice.
F&F	Retry mode	The message is offered to the receiver several times in a Unit of Work, but every time the UOW was rolled back. If the message is rolled back as often as the Backout Threshold (a setting for the service) indicates, then it is in retry mode and it is rejected. The receiver receives a warning with reason-code `IFSARC_POISON_MESSAGE`.

Table A.11: Undelivered messages due to rejection

Appendix B

Functions, assertions, invariants and main-program

In this appendix we give all auxiliary functions, assertions, invariants and the main-program in a compact format to avoid page-turning in verification of the proofs given in chapters 3 and 4.

All auxiliary functions are given in table B.1 in alphabetic order, for $c \in C$, $s \in S$, $ct \in \mathbb{N}$, $st \in \mathbb{N} \cup \{-1\}$, $K \in \mathbb{N}$ and $M \in 2^{Message}$. All system- and repetition invariants are given in table B.2. Note that some auxiliary functions and/or auxiliary are only used in the 1-R/R (pgv_0), or only in the n-R/R protocol (*allRep*,*sffi*, scw, sct, pgv_0^*, $pgv_0^*(c)$, $pdtid_0$, $pdtid_0(c)$, $pdt2ct_0$ and $pdt2ct_1$).

The assertions of clients and services in the 1-R/R protocol are given in table B.3. The assertions of clients and services in the n-R/R protocol are given in table B.4. Note that the services' program texts and assertions are the same in the 1-R/R and n-R/R protocols.

The main-program of the 1-R/R and n-R/R protocols is given on page 133. Assertions $A0$ and $A1$ have been included in the program text.

We advise readers interested in chapter 3 to keep copies of pages 56, 57, 130, 131 and 133 at hand while reading that chapter in order to avoid a lot of page turning. Readers interested in chapter 4 are advised to keep copies of pages 57, 88, 130, 132 and 133 at hand.

		Auxiliary functions
$allRep(c, CT, M)$	$=$	$\lvert CT\rvert = \lvert M\rvert \wedge CT = \{m.tid \mid m \in M\} \wedge$ $\langle \forall m : m \in M : m.fa = cw[m.ta, m.tid] \wedge m.ta = c\rangle$
$gv(c, ct)$	$=$	$(RQO[c, ct],\ RPO[c, ct],\ CD[c, ct])$
$sct(ct_0, K)$	$=$	$\{ct \mid ct_0 \leq ct < ct_0 + K\}$
$scw(c, ct_0, K)$	$=$	$\{cw[c, ct] \mid ct \in sct(ct_0, K)\}$
$sffi(c, s, ct)$	$=$	$\{m \mid m \in ffi[s] \wedge m.fa = c \wedge m.tid = ct\}$
$srpi(c, s, ct)$	$=$	$\{m \mid m \in rpi[c] \wedge m.fa = s \wedge m.tid = ct\}$
$srqi(c, s, ct)$	$=$	$\{m \mid m \in rqi[s] \wedge m.fa = c \wedge m.tid = ct\}$
$tra(c, s, ct, st)$	$\equiv$	$cw[c, ct] = s \wedge ct2st[c, ct] = st \wedge 0 \leq ct < tid[c] \wedge -1 \leq st < tid[s]$

Table B.1: Auxiliary functions in the 1-R/R and n-R/R protocols

	System Invariants
I_0 :	$\langle \forall c, s : c \in C \wedge s \in S : I_0(c, s)\rangle$
$I_0(c, s)$:	$\langle \forall ct, st : tra(c, s, ct, st) : cd[c, ct] \equiv cd[s, st]\rangle$
pCD_1 :	$\langle \forall c, s : c \in C \wedge s \in S \wedge CD[c, tid[c]] = 0 : \neg cd[c, tid[c]] \wedge \neg cd[s, ct2st[c, tid[c]]] \rangle$
pcd_0 :	$\langle \forall s, st : s \in S \wedge tid[s] \leq st : \neg cd[s, st]\rangle$
$pct2st_0$:	$\langle \forall c, s, ct : c \in C \wedge s \in S \wedge RQO[c, ct] = 0 : ct2st[c, ct] < tid[s] \rangle$
pcw_0 :	$\langle \forall c, s, ct : c \in C \wedge s \in S \wedge ct \in \mathbb{N} : \lvert srqi(c, s, ct) \cup srpi(c, s, ct)\rvert \leq 1\rangle$
$pdtid_0$:	$\langle \forall c : c \in C : pdtid_0(c) \rangle$
$pdtid_0(c)$:	$\langle \forall dt : 0 \leq dt < dtid[c] :$ $(\forall ct, ct' : dt2ct[c, dt] \leq ct \leq ct' < dt2ct[c, dt + 1] : cd[c, ct] \equiv cd[c, ct'])$ $\rangle$
$pdt2ct_0$:	$\langle \forall c : c \in C : tid[c] = dt2ct[c, dtid[c]] \rangle$
$pdt2ct_1$:	$\langle \forall c, dt : c \in C \wedge dtid[c] \leq dt : dt2ct[c, dt] \leq tid[c] \rangle$
$pffi_0$:	$\langle \forall m, s : s \in S \wedge m \in ffi[s] : m.fa \in C \wedge m.ta = s \rangle$
$pffi_1$:	$\langle \forall m, s, c, ct : s \in S \wedge m \in ffi[s] \wedge c = m.fa \wedge ct = m.tid :$ $gv(c, ct) = (1, 1, 1) \wedge cd[c, ct] = \neg m.to \wedge srpi(c, s, ct) = \emptyset$ $\rangle$
pgv_0	$\langle \forall c, ct : RQO[c, ct] = 1 \wedge CD[c, ct] = 0 : tid[c] = ct\rangle$
pgv_0^* :	$\langle \forall c, ct' : gv(c, ct') = (1, 0, 0) \vee gv(c, ct') = (1, 1, 0) : ct' \in sct(tid[c], sz[c])\rangle$
$pgv_0^*(c)$:	$\langle \forall ct' : gv(c, ct') = (1, 0, 0) \vee gv(c, ct') = (1, 1, 0) : ct' \in sct(tid[c], sz[c])$
$pRQO_1$:	$\langle \forall c, s, ct : c \in C \wedge s \in S \wedge ct \in \mathbb{N} \wedge RQO[c, ct] = 0 : srqi(c, s, ct) \cup srpi(c, s, ct) = \emptyset \rangle$
$prqi_0$:	$\langle \forall m, s : s \in S \wedge m \in rqi[s] : m.fa \in C \wedge m.ta = s \rangle$
$prqi_1$:	$\langle \forall m, s, c, ct : s \in S \wedge m \in rqi[s] \wedge c = m.fa \wedge ct = m.tid :$ $gv(c, ct) = (1, 0, 0) \wedge cw[c, ct] = s$ $\rangle$
$prqi_2$:	$\langle \forall s : s \in S : \lvert rqi[s]\rvert \leq 1\rangle$
$prpi_0$:	$\langle \forall m, c : c \in C \wedge m \in rpi[c] : m.fa \in S \wedge m.ta = c \rangle$
$prpi_1$:	$\langle \forall m, c, s, ct : c \in C \wedge m \in rpi[c] \wedge s = m.fa \wedge ct = m.tid :$ $gv(c, ct) = (1, 1, 0) \wedge (tid[s] = ct2st[c, ct] \vee m.to) \wedge srqi(c, s, ct) = \emptyset$ $\rangle$
	Repetition invariants of client c
$pRQO_0(c)$:	$\langle \forall ct : ct \in \mathbb{N} : RQO[c, ct] \leq 1 \wedge (ct < tid[c] \equiv RQO[c, ct] = 1)\rangle$
$pCD_0(c)$:	$\langle \forall ct : ct \in \mathbb{N} : CD[c, ct] \leq 1 \wedge (ct < tid[c] \equiv CD[c, ct] = 1)\rangle$
$pRPO_0(c)$:	$\langle \forall ct : ct \in \mathbb{N} : RPO[c, ct] \leq 1 \wedge (ct < tid[c] \equiv RPO[c, ct] = 1)\rangle$
	Repetition invariants of service s
$pct2st_1(s)$:	$\langle \forall c, ct : c \in C \wedge ct \in \mathbb{N} \wedge cw[c, ct] = s : ct2st[c, ct] < tid[s]\rangle$
$prpi_2(s)$:	$\langle \forall m, c : c \in C \wedge m \in rpi[c] \wedge m.fa = s : m.to\rangle$

Table B.2: System- and repetition-invariants in the 1-R/R and n-R/R protocols

		Assertions of client c (1-R/R protocol)
$A5.1$	:	$pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c)$
$A5.2$	:	$gv(c, tid[c]) = (0,0,0)$
$A6.1$	:	$pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c)$
$A6.2$	:	$cw[c, tid[c]] = s$
$A6.3$	:	$gv(c, tid[c]) = (0,0,0)$
$A6.4$	:	$srqi(c, s, tid[c]) \cup srpi(c, s, tid[c]) = \emptyset$
$A7.1$	:	$(\, pRQO_0(c) \ \wedge \ pRPO_0(c))\,(\, \mathbb{N} \ := \ \mathbb{N}/\{tid[c]\}) \ \wedge \ pCD_0(c)$
$A7.2$	:	$cw[c, tid[c]] = s$
$A7.3$	:	$gv(c, tid[c]) = (1,0,0) \vee gv(c, tid[c]) = (1,1,0)$
$A7.4$	:	$\lvert srqi(c, s, tid[c]) \cup srpi(c, s, tid[c]) \rvert \leq 1$
$A8.1$	:	$(\, pRQO_0(c) \ \wedge \ pRPO_0(c))\,(\, \mathbb{N} \ := \ \mathbb{N}/\{tid[c]\}) \ \wedge \ pCD_0(c)$
$A8.2$	:	$cw[c, tid[c]] = s$
$A8.3$	:	$gv(c, tid[c]) = (1,1,0) \ \wedge \ (\, tid[s] = ct2st[c, tid[c]] \ \vee \ m_2.to\,)$
$A8.4$	:	$srqi(c, s, tid[c]) = \emptyset \ \wedge \ srpi(c, s, tid[c]) = \emptyset$
		Assertions of service s (1-R/R protocol)
$A13.1$	:	$prpi_2(s)$
$A13.2$	:	$pct2st_1(s)$
$A13.3$	:	$rd[s]$
$A14.1$	:	$gv(c, ct) = (1,0,0) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A14.2$	:	$srqi(c, s, ct) = \emptyset \ \wedge \ srpi(c, s, ct) = \emptyset$
$A14.3$	:	$prpi_2(s)$
$A14.4$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A14.5$	:	$\neg rd[s]$
$A16.1$	:	$(\, gv(c, ct) = (1,1,0) \vee gv(c, ct) = (1,1,1)\,) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A16.2$	:	$srqi(c, s, ct) = \emptyset$
$A16.3$	:	$prpi_2(s)(C := C/\{c\})$
$A16.4$	:	$\langle \forall m \ : \ m \in rpi[c] \ \wedge \ \neg m.to \ \wedge \ m.fa = s \ : \ m.tid = ct \rangle$
$A16.5$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A16.6$	:	$\neg rd[s]$
$A17.1$	:	$[gv(c, ct) = (1,1,0) \vee gv(c, ct) = (1,1,1)] \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A17.2$	:	$srqi(c, s, ct) = \emptyset$
$A17.3$	:	$gv(m_3.fa, m_3.tid) = (1,1,1) \ \wedge \ cd[m_3.fa, m_3.tid] = \neg m_3.to \ \wedge \ srpi(m_3.fa, s, m_3.tid) = \emptyset$
$A17.4$	:	$m_3.fa \in C$
$A17.5$	:	$prpi_2(s)(C := C/\{c\})$
$A17.6$	:	$\langle \forall m \ : \ m \in rpi[c] \ \wedge \ \neg m.to \ \wedge \ m.fa = s \ : \ m.tid = ct \rangle$
$A17.7$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A17.8$	:	$\neg rd[s]$
$A18.1$	:	$gv(c, ct) = (1,1,1) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A18.2$	:	$srqi(c, s, ct) = \emptyset \ \wedge \ srpi(c, s, ct) = \emptyset$
$A18.3$	:	$cd[c, ct] = \neg m_3.to$
$A18.4$	:	$prpi_2(s)$
$A18.5$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A18.6$	:	$\neg rd[s]$

Table B.3: Assertions in the 1-R/R protocol

		Assertions of client c (n-R/R protocol)
$A5.1$	:	$pRQO_0(c) \ \wedge \ pRPO_0(c) \ \wedge \ pCD_0(c)$
$A5.2$	:	$\langle \forall ct' : tid[c] \leq ct' : gv(c, ct') = (0,0,0) \rangle$
$A6.1$	:	$(\ pRQO_0(c) \ \wedge \ pRPO_0(c)\ )\ (\ \mathbb{N} \ := \ \mathbb{N}/CT_7\ ) \ \wedge \ pCD_0(c)$
$A6.2$	:	$\langle \forall ct' : ct' \in CT_6 \ : \ gv(c, ct') = (0,0,0) \ \rangle \ \wedge$
		$\langle \forall ct' : ct' \in CT_7 \ : \ gv(c, ct') = (1,0,0) \ \vee \ gv(c, ct') = (1,1,0) \ \rangle$
$A6.3$	:	$gv(c, tid[c]) = (0,0,0)$
$A6.4$	:	$\langle \forall ct' : ct' \in CT_6 \ : \ srqi(c, cw[c, ct'], ct') \ \cup \ srpi(c, cw[c, ct'], ct') = \emptyset \ \rangle \ \wedge$
		$\langle \forall ct' : ct' \in CT_7 \ : \lvert srqi(c, cw[c, ct'], ct') \cup srpi(c, cw[c, ct'], ct') \rvert \leq 1 \ \rangle$
$A6.5$	:	$CT_6 \cup CT_7 = sct(tid[c], sz[c]) \ \wedge \ \langle \forall i, j \ : \ 6 \leq i < j < 10 \ : \ CT_i \cap \ CT_j = \emptyset$
$A6.6$	:	$CT_8 = \emptyset \ \wedge CT_9 = \emptyset \ \wedge \ M_2 = \emptyset$
$A7.1$	:	$(\ pRQO_0(c) \ \wedge \ pRPO_0(c)\ )\ (\ \mathbb{N} \ := \ \mathbb{N}/sct(tid[c], sz[c])\ ) \ \wedge \ pCD_0(c)$
$A7.2$	:	$\lvert scw(c, tid[c], sz[c]) \rvert = \lvert sct(tid[c], sz) \rvert$
$A7.3$	:	$\langle \forall ct' : \ ct' \in CT_7 \ : \ gv(c, ct') = (1,0,0) \ \vee \ gv(c, ct') = (1,1,0) \ \rangle \ \wedge$
		$\langle \forall ct' : \ ct' \in CT_8 \ : \ gv(c, ct') = (1,1,0) \ \rangle$
$A7.4$	:	$\langle \forall ct' : \ ct' \in CT_7 \ : \ \lvert srqi(c, cw[c, ct'], ct') \cup srpi(c, cw[c, ct'], ct') \rvert \leq 1 \ \rangle \ \wedge$
		$\langle \forall ct' : \ ct' \in CT_8 \ : \ srqi(c, cw[c, ct'], ct') = \emptyset \wedge \ srpi(c, cw[c, ct'], ct') = \emptyset \ \rangle$
$A7.5$	:	$CT_7 \cup CT_8 = sct(tid[c], sz[c]) \ \wedge \ \langle \forall i, j \ : \ 6 \leq i < j < 10 \ : \ CT_i \cap \ CT_j = \emptyset \rangle$
$A7.6$	:	$CT_9 = \emptyset \ \wedge \ allRep(c, CT_8, M_2)$
$A7.7$	:	$\langle \forall m \ : \ m \in M_2 \ : \ ct2st[c, m.tid] = tid[m.fa] \ \vee \ m.to \ \rangle$
$A8.1$	:	$(\ pRQO_0(c) \ \wedge \ pRPO_0(c)\ )\ (\ \mathbb{N} \ := \ \mathbb{N}/sct(tid[c], sz[c])\ ) \ \wedge \ pCD_0(c)(\ \mathbb{N} \ := \ \mathbb{N}/CT_9\ )$
$A8.2$	:	$\lvert scw(c, tid[c], sz[c]) \rvert = \lvert sct(tid[c], sz[c]) \rvert$
$A8.3$	:	$\langle \forall ct' : \ ct' \in CT_8 \ : \ gv(c, ct') = (1,1,0) \rangle \ \wedge \ \langle \forall ct' : \ ct' \in CT_9 \ : \ gv(c, ct') = (1,1,1) \rangle$
$A8.4$	:	$\langle \forall ct' \ : \ ct' \in sct(tid[c], sz[c]) \ : \ srqi(c, cw[c, ct'], ct') \ \cup \ srpi(c, cw[c, ct'], ct') = \emptyset \rangle$
$A8.5$	:	$CT_8 \cup CT_9 = sct(tid[c], sz[c]) \ \wedge \ \langle \forall i, j \ : \ 6 \leq i < j < 10 \ : \ CT_i \cap \ CT_j = \emptyset \rangle$
$A8.6$	:	$allRep(\ c, \ sct(tid[c], sz[c]), \ M_2\ )$
$A8.7$	:	$\langle \forall m \ : \ m \in M_2 \ : \ ct2st[c, m.tid] = tid[m.fa] \ \vee \ m.to \ \rangle$
$A8.8$	:	$\langle \forall ct' \ : \ ct' \in CT_9 : \ cd[c, ct'] \equiv (\neg\, (\bigvee m : \ m \in M_2 \ : \ m.to) \wedge cv) \rangle \ \wedge$
		$\langle \forall ct' \ : \ ct' \in CT_9 : \ \neg cd\, [cw\, [c, ct']\, , ct2st\, [c, ct']] \rangle$
		Assertions of service s (n-R/R protocol)
$A13.1$	:	$prpi_2(s)$
$A13.2$	:	$pct2st_1(s)$
$A13.3$	:	$rd[s]$
$A14.1$	:	$gv(c, ct) = (1,0,0) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A14.2$	:	$srqi(c, s, ct) = \emptyset \ \wedge \ srpi(c, s, ct) = \emptyset$
$A14.3$	:	$prpi_2(s)$
$A14.4$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A14.5$	:	$\neg rd[s]$
$A16.1$	:	$(\ gv(c, ct) = (1,1,0) \vee gv(c, ct) = (1,1,1)\ ) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A16.2$	:	$srqi(c, s, ct) = \emptyset$
$A16.3$	:	$prpi_2(s)(C := C/\{c\})$
$A16.4$	:	$\langle \forall m \ : \ m \in rpi[c] \ \wedge \ \neg m.to \ \wedge \ m.fa = s \ : \ m.tid = ct \rangle$
$A16.5$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A16.6$	:	$\neg rd[s]$
$A17.1$	:	$[gv(c, ct) = (1,1,0) \vee gv(c, ct) = (1,1,1)] \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A17.2$	:	$srqi(c, s, ct) = \emptyset$
$A17.3$	:	$gv(m_3.fa, m_3.tid) = (1,1,1) \ \wedge \ cd[m_3.fa, m_3.tid] = \neg m_3.to \ \wedge \ srpi(m_3.fa, s, m_3.tid) = \emptyset$
$A17.4$	:	$m_3.fa \in C$
$A17.5$	:	$prpi_2(s)(C := C/\{c\})$
$A17.6$	:	$\langle \forall m \ : \ m \in rpi[c] \ \wedge \ \neg m.to \ \wedge \ m.fa = s \ : \ m.tid = ct \rangle$
$A17.7$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A17.8$	:	$\neg rd[s]$
$A18.1$	:	$gv(c, ct) = (1,1,1) \ \wedge \ cw[c, ct] = s \ \wedge \ ct2st[c, ct] = tid[s]$
$A18.2$	:	$srqi(c, s, ct) = \emptyset \ \wedge \ srpi(c, s, ct) = \emptyset$
$A18.3$	:	$cd[c, ct] = \neg m_3.to$
$A18.4$	:	$prpi_2(s)$
$A18.5$	:	$\langle \forall c', ct' \ : \ c' \in C \ \wedge \ ct' \in \mathbb{N} \ \wedge \ cw[c', ct'] = s \ \wedge \ \neg(c' = c \ \wedge \ ct' = ct) \ : \ ct2st[c', ct'] < tid[s] \rangle$
$A18.6$	:	$\neg rd[s]$

Table B.4: Assertions in the n-R/R protocol

Main Program

Below we give the main-program of the n-R/R protocol. By ignoring declarations of and assignments to the variables *dtid*, *dt2ct* and sz, the program-text can also be used for the 1-R/R protocol. Indeed, the assertions $A1.1$ and $A1.2$ are orthogonal to those statements.

```
|[ type $Message$ = record |[ $fa, ta : A;\ tid : \mathbb{N};\ to : \mathbb{B}$ ]|

   const $C$ = $\{2n | n \in \mathbb{N}\}$
   const $S$ = $\{2n+1 | n \in \mathbb{N}\}$
   const $A$ = $C \cup S$;

   var $tid$             : $Array[A]$              : $\mathbb{N}$;
   var $cd$              : $Array[A][-1..\infty)$  : $\mathbb{B}$;
   var $cw$              : $Array[C][\mathbb{N}]$  : $S$;
   var $ct2st$           : $Array[C][\mathbb{N}]$  : $[-1..\infty)$;

   var $rqi$             : $Array[S]$              : Set of $Message$;
   var $rpi$             : $Array[C]$              : Set of $Message$;
   var $ffi$             : $Array[S]$              : Set of $Message$;
   var $rd$              : $Array[S]$              : $\mathbb{B}$;

   var $CD, RQO, RPO$    : $Array[C][\mathbb{N}]$  : $\mathbb{N}$;

   var $dtid$            : $Array[C]$              : $\mathbb{N}$;
   var $dt2ct$           : $Array[C][\mathbb{N}]$  : $\mathbb{N}$;
   var $sz$              : $Array[C]$              : $\mathbb{N}$;


  { Assertion 0: $true$ }
  ⟨ $tid := \overrightarrow{0}$;
    $cd := \overrightarrow{false}$;
    $rd := \overrightarrow{true}$;
    $ct2st := \overrightarrow{-1}$;
    $rqi, rpi, ffi := \overrightarrow{\emptyset}, \overrightarrow{\emptyset}, \overrightarrow{\emptyset}$;
    $CD, RQO, RPO := \overrightarrow{0}, \overrightarrow{0}, \overrightarrow{0}$;
    $dtid, dt2ct := \vec{0}, \vec{0}$;
  ⟩

  { $A1.1 : \langle \forall c : c \in C : pRQO_0(c) \wedge pRPO_0(c) \wedge pCD_0(c) \rangle$ }
  { $A1.2 : \langle \forall s : s \in S : prpi_2(s) \wedge pct2st_1(s) \wedge rd[s] \rangle$ }
  $( \| c : c \in C : Client(c) ) \ \| \ ( \| s : s \in S : Service(s) )$

]|
```

Bibliography

[Bra97] S. Bradner. Key words for use in RFCs to indicate requirement levels. Technical Report RFC 2119, IETF, 1997.

[Fit03] Tony Fitzpatrick. *IFSA AND WEB SERVICES POSITION AND STRATEGY*, January 2003. IFSA version 2.0, Document 114.

[FvG99] W.H.J. Feijen and A.J.M. van Gasteren. *On a Method of Multiprogramming*. Springer-Verlag, 1999.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[it03] IFSA interface team. *IFSA APPLICATION BUS, TECHNICAL DESIGN C AND COBOL INTERFACE*, August 2003. IFSA version 2.0, Document 852.

[Kay03] Doug Kay. *Loosely Coupled: The Missing Pieces of Web Services*. Rds Associates Inc, 2003.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 95–106. Morgan Kaufmann, 1990.

[LKB77] J.K. Lenstra, A.H.G.R. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[SKS98] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., 1998.