
Distributed Atomic Transactions over RESTful Services

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics
Major in Software Design

presented by
Masiar Babazadeh

under the supervision of
Prof. Dr. Cesare Pautasso

June 2012

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Masiar Babazadeh
Lugano, 17 June 2012

Abstract

ABSTRACT

Acknowledgements

Contents

Contents	vii
List of Figures	ix
List of Tables	xi
1 Introduction	1
2 TCC	3
2.1 Why REST needs Transactions	3
2.1.1 A Small Example	3
2.2 Current solutions	4
2.2.1 REST-*	4
2.2.2 ATOM Pub/Sub	5
2.3 TCC	5
2.3.1 Generalization: Try-Cancel/Confirm	6
2.3.2 The protocol	7
2.3.3 Guaranteeing atomicity	9
3 Architecture	11
3.1 General Overview	11
3.2 Logical View	11
3.3 Process View	13
3.4 Development View	14
3.5 Physical View	14
3.6 Use Cases	15
3.6.1 Immediate Purchase	16
3.6.2 Deletion	16
3.6.3 Timeout Before Confirmation	17
3.6.4 Timeout During Confirmation	17
4 Component Design and Interfaces	19
4.1 Client Side	19
4.1.1 Design	19
4.1.2 Interfaces	20
4.2 Service Side	20

4.2.1	Design	20
4.2.2	Interfaces	20
4.3	Transaction Manager	21
4.3.1	Design	21
4.3.2	Interfaces	21
5	Detailed Protocol Design	23
6	Implementation	25
6.1	General TCC Implementation	25
6.1.1	Google Chrome Extension	25
6.1.2	Node Service	27
6.1.3	Node Transaction Manager	28
6.2	Transaction Manager integrated in Browser Extension	29
6.2.1	Google Chrome Extension	30
6.3	TCC used in server interaction	30
6.3.1	Introduction to S	30
6.3.2	TCC Library	31
6.3.3	Transaction Manager Library	31

Figures

2.1	State machine representing a generic resource that implements the Try-Cancel/Confirm protocol	7
2.2	Protocol architecture in the Happy Path case	8
2.3	Example of a book reservation.	9
3.1	Logical View of the system.	12
3.2	Process View of the system.	13
3.3	Development View of the system.	15
3.4	Physical View of the system.	16
6.1	Freedom of choice by the user for the transaction manager.	26
6.2	Transactions in progress for the logged user.	26

Tables

Chapter 1

Introduction

Chapter 2

TCC

2.1 Why REST needs Transactions

In the REST community there is dissension whether or not transaction support is needed and possible. This section will show why REST needs transactions through a business case, it will then outline the current solutions and their problems and finally will show the Try-Cancel/Confirm (TCC) pattern.

2.1.1 A Small Example

Suppose we want to buy the first two books of a fantasy series. We want the both of them right away, we don't want to wait when finishing the first one for the second one to arrive, nor wasting time ordering the second one while reading the first. We found them on two different websites: amazon.com and bookfinder.com, the first one is available only at Amazon, the second one only on Bookfinder. Let's assume both of the stores have the same hypermedia contract for buying (to simplify things, without loss of generality, since the composite service must be aware of all the hypermedia contracts involved). The REST implementation of the store information and buying process could be sketched as follows. Clients can inquiry for the availability of a book in a store at the URI: `/books/book-title`. For example, the request `GET /books/A-Game-of-Thrones` will return a hyperlink to buy the the specified book or none (e.g. `204 No Content`) if the book is not available anymore. A `POST` request to the same URL with a payload referencing such book will create a new purchase resource and redirect the client to it by sending a hyperlink identifying it, for example `/buy/{book-title}/{id}`. The body of the request might contain some information about the chosen book (i.e., `<book title="Games of Thrones" author="George R. R. Martin">`). The purchase resource can be updated with additional information by performing a `PUT` on `/buy/{book-title}/{id}/`.

Now that we know the interface of the services, we are ready to dive into a user story that will motivate the use of transactions in REST.

As a bookworm, I would like to buy the first two books of the Game of Thrones series. Both of the books are available online on two different webstores.

Since it's the webstore's responsibility to satisfy my need, a very naive implementation,

without a transaction model for REST, would be the following:

1. GET `amazon.com/books/A-Game-of-Thrones`
2. POST `amazon.com/buy/A-Game-of-Thrones`
3. GET `bookfinder.com/books/A-Clash-of-Kings`
4. POST `bookfinder.com/buy/A-Clash-of-Kings`

What can happen is that after we buy the first book (step 2) the second book won't be available anymore (step 3). In this way we end up having just the first book but not the second. If we try to reorder the requests as follows:

1. GET `amazon.com/books/A-Game-of-Thrones`
2. GET `bookfinder.com/books/A-Clash-of-Kings`
3. POST `amazon.com/buy/A-Game-of-Thrones`
4. POST `bookfinder.com/buy/A-Clash-of-Kings`

we can still end up in the same situation, because both in step 1 and 2 may return availability for the chosen books, but the subsequent requests may fail due to concurrent purchase (imagine there is only one book left and while we are between step 2 and 3 someone already sent a **PUT** request purchasing the book before us).

The idea behind TCC is to make step 3 and 4 tentative, so that they can be confirmed later, thus making the whole process atomic and assuring that it will happen as a whole or not happen at all.

The next section will illustrate the current solutions that we can find to solve this problem and why they are not suitable for distributed atomic transactions over REST.

2.2 Current solutions

This section will focus on two solutions that can be found to solve this problem. The first one is REST-*, recently appeared on the book "REST in Practice". The second one is ATOM Pub/Sub.

2.2.1 REST-*

The JBoss REST-* initiative aims to provide various quality of services guarantees for RESTful web services, in the same way as WS-* has done for web services. REST-* follows an approach very similar to the one used by TIP or WS-AT: to make the invocation transactional, a context is added to each invocation. The problem is that the receiving service has to understand the context in order to participate in the transaction, thus bringing tight coupling, something that TCC tries to avoid.

2.2.2 ATOM Pub/Sub

This approach uses, as the name suggests, a publish/subscribe mechanism based on feeds. The transaction coordinator publishes updates on the transaction outcome and each participant listens to what it can be interested in. This is feasible up to the point that each participant knows the feed it has to listen to and understands the semantics of the published updates. Again this shows a tight coupling, which is not present in TCC as in our case the participants has to know nothing besides their own business contract. Moreover, the ATOM Pub/Sub solution implies that the coordinator cannot inquiry a participant about its final outcome (it could be interested if we take into account heuristic decisions). This is odd, since a coordinator has all the reasons to be interested in the final outcome of a transaction.

2.3 TCC

Now that current solutions have been illustrated, the reader has a glimpse on what we have now and what we want to achieve. With another use case this section will show how the TCC protocol works from the point of view of the two parties involved.

Suppose I want to buy the first and the second book of a fantasy series. I want to be able to confirm the purchase when I'm done. Purchases that are not confirmed are not billed to by account.

The confirmation should be business-specific. Let's assume that a confirmation link is returned by the RESTful API of the webstore, for example in response to a `GET /buy/book-title/id` the service would return something like `<book title id> <payment uri="/payment/X"> </book>`. Thus the purchase can be confirmed by performing a request `PUT /payment/X <MASTERCARD ...>`.

The workflow of the webstore can be seen as follows:

Workflow Engine:

```
⇒GET amazon.com/books/A-Game-of-Thrones

⇐200 OK

⇒POST amazon.com/buy/A-Game-of-Thrones

⇐302 Found
  Content-Type: text/uri-list
  /buy/A-Game-of-Thrones/A
⇒GET bookfinder.com/books/A-Clash-of-Kings

⇐200 OK

⇒bookfinder.com/buy/A-Clash-of-Kings

⇐302 Found
  Content-Type: text/uri-list
  /buy/A-Game-of-Thrones/A
⇒GET amazon.com/buy/A-Game-of-Thrones
```

```

<=200 OK
  Content-Type: text/uri-list
  /payment/A
=>GET bookfinder.com/buy/A-Clash-of-Kings/B

```

```

<=200 OK
  Content-Type: text/uri-list
  /payment/B

```

```

Transaction Coordinator:
=>PUT amazon.com/payment/A

```

```

<=200 OK
=>PUT bookfinder.com/payment/B

```

```

<=200 OK

```

The first set of interactions can be driven by the workflow that composes the two services, while the final confirmations that conclude the transaction can be made by a transaction coordinator component that will execute the **PUT** requests.

What has been shown is the happy path, in which everything goes as expected and no one cries. What happens if, for example, step 4 fails? As mentioned before, the payment is not triggered until there is no confirmation. This setup avoids our original problem, without ending up in a situation in which I have one of the two books but not the other.

Let's refine even more the protocol and take a look of the problem from the point of view of the webstores. The last use case is the following:

As a webstore, I do not want to wait for a confirmation forever. I want to be able to autonomously cancel a pending booking after some timeout expires.

This use case brings an obvious problem: what happens if a user reserves some book but never confirms them? This might make the webstore lose money since there could be other people interested in the reserved goods. As result, we can add a cancellation event, that is triggered when a timeout expires (which is specific to the service). The REST implementation, then, can be adjusted as follows: **GET /buy/{book-title}/{id}** returns **<book title id> <payment uri="/payment/X" deadline="24h"> </book>**. The composing workflow service may use the deadline field as a hint on when the reservation will expiry and cancel itself.

2.3.1 Generalization: Try-Cancel/Confirm

The use cases presented are particular cases of the more general Try-Cancel/Confirm (TCC) protocol. As shown in [FIGURE], once a request is issued, it remains "tentative" (in the reserved state) until confirmed or cancelled (either by a timeout in the server or from a client).

This protocol meets the need of industry, which has shown that transactions need not to be invasive. This in particular refers to the complexity in the design that might be introduced to create services that can participate in a transaction. With TCC's approach, we have loose coupling among resources: participating services are unaware of the fact that

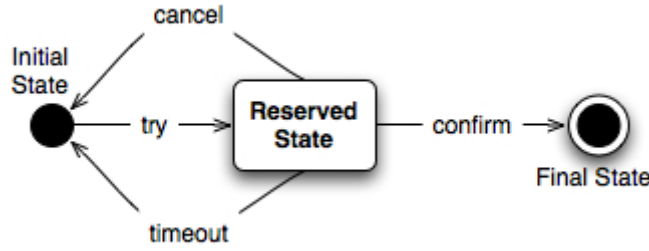


Figure 2.1. State machine representing a generic resource that implements the Try-Cancel/Confirm protocol

they are part of a global transaction. Indeed, the individual participating services do not need to have additional knowledge or implement any extra protocol besides the one they already support.

Also, with this solution we automatically avoid the use of an explicit transaction context, which is widely adopted in distributed transaction protocols. This is one of the most important requirements to ensure loose coupling.

2.3.2 The protocol

Before showing the protocol itself, let's define the transaction more formally:

Definition 1. A *REST-based transaction* T (e.g., purchasing two books) is a number of invocations R_i (e.g., purchasing individual books) across *REST-ful services* S_i (e.g., amazon.com and bookfinder.com) that need to either confirm altogether or cancel altogether. In other words: either all R_i succeed via an explicit confirmation $R_i, confirm$ (e.g., by paying for the book) , or all R_i cancel but nothing in between.

Happy path

1. A transactional workflow T goes about interacting with multiple distinct RESTful service APIs S_i
2. Interactions R_i may lead to a state transition of the participating service S_i identified by some URI - this URI corresponds to $R_i, confirm$
3. Once the workflow T successfully completes, the set of confirmation URIs and any required application-level payload is passed to a transaction service (or coordinator)
4. The transaction service then calls all of the $R_i, confirm$ with an idempotent PUT request on the corresponding URIs with the associated payloads

The protocol (shown in 2.2) guarantees atomicity because each service receives either a consistent request for cancel or to confirm. Moreover, all of them terminate their business transactions in the same way. Note that **PUT** and **DELETE** are idempotent actions, therefore we can assume $R_i, confirm$ is idempotent too.

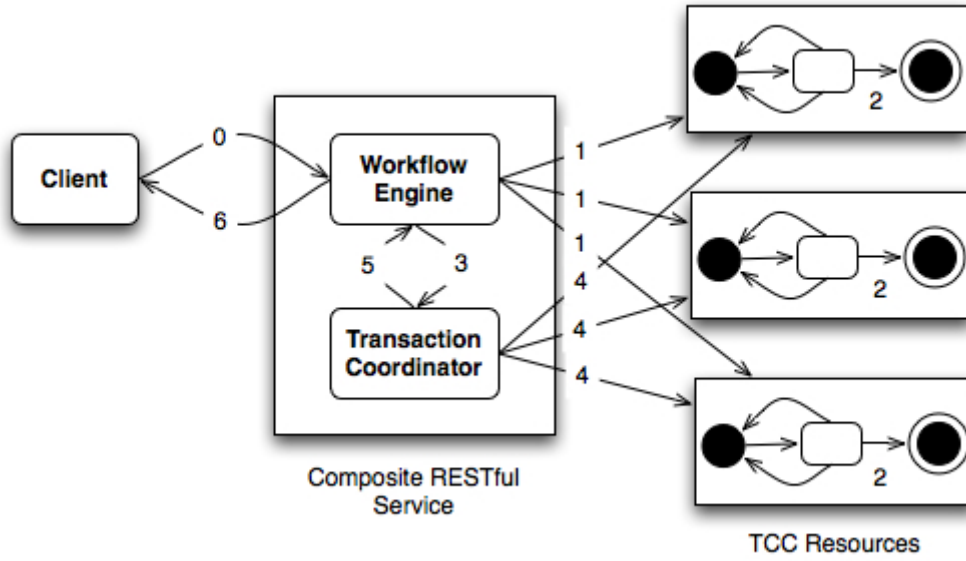


Figure 2.2. Protocol architecture in the Happy Path case

Sad path

The protocol looks very simple. How can this work in the presence of failure? This section will show what can go wrong and how can it recover from failures. We assume that each party is able to restore its own durable state, so we focus on the recoverability of the atomicity property across all parties.

We define recovery as follows:

1. Checking the state of a transaction after node failure followed by restart, or
2. Checking the state of a transaction triggered by timeout

The recovery is performed by both the coordinator service and the participant services. The first one is expected to recover all the transactions he knows: if there were some pending transactions (waiting for a confirm action) we expect it to recover them all. On the other hand, the services want to recover the transactions because they want to release the reserved resources as early as possible.

As for the participant services, each participating service S_i does the following:

1. For recovery before step 2, do nothing.
2. For recovery after step 4: do nothing.
3. For recovery in between steps 2 and 4: execute $R_i, cancel$ autonomously (This can be triggered by a timeout).

For the coordinator things are a bit different. The most problematic failures are those happening during step 4. We want all the participants to arrive at the same end state for

transaction T ; step 4 involves multiple participants that are not aware that they are part of a transaction, so a failure here could be problematic.

1. For recovery before step 2, do nothing.
2. For recovery between steps 2 and 4: do nothing.
3. For recovery after step 4: do nothing.
4. For recovery during step 4: retry $R_i, confirm$ with each participating service S_i . Since $R_i, confirm$ are performed using idempotent methods, they may be retried as many times as necessary. Note: this requires the coordinator to durably log all participant information before starting step 4.

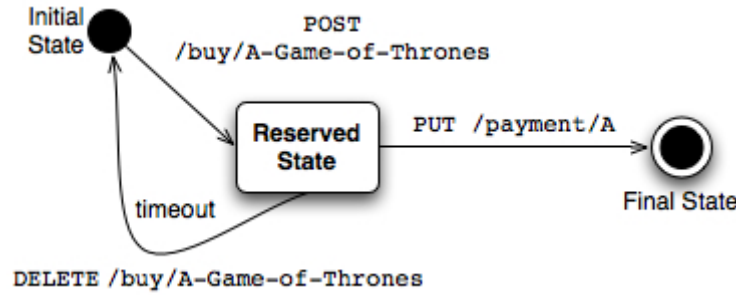


Figure 2.3. Example of a book reservation.

2.3.3 Guaranteeing atomicity

Even with failures, we can still (eventually) preserve atomicity. In fact, given enough time, the global transaction T will be finally confirmed everywhere or cancelled everywhere (or nothing happened). If we take a look at the protocol steps, we can prove that either all R_i are confirmed or all are cancelled. More in details:

1. If there are no failures, then steps 1-4 run through and each R_i will have been confirmed.
2. For any failures before step 2, no R_i exists, meaning that nothing has happened.
3. For any failures during or after step 2 but before step 4: all R_i will eventually be cancelled autonomously by each S_i .
4. For any failures during step 4: the coordinator will retry each $R_i, confirm$ until it succeeds. Because confirmation is idempotent, this will eventually succeed.
5. For any failures after step 4: all $R_i, confirm$ have been done, so we already have atomicity and no action is required.

Saddest path

Although everything seems fine and the protocol seems bullet-proof, there is one weak point in our proof of atomicity. During step 4, while the coordinator is confirming transactions, it may happen that some services S_i , that still has transactions that need to be confirmed, time out and cancel transactions on their own. In the worst case this means that some transactions will be confirmed and some will cancel, effectively breaking atomicity. This is what is called a *heuristic exception*.

In the world of atomicity, there has been a lot of interesting work, but the most important result is that a perfect solution is not possible. In practice, there will always be a possibility that at least one participating service (or node) is unaware of the outcome of the global distributed transaction.

To avoid this problem it's very important to tune correctly the timeouts at the services side. If the timeout is too small (that is, resources are kept in a reserved state for a small amount of time) a client will find hard to confirm something in that small time slot, and may happen to have timeouts before and during step 4. On the other hand services want to have a timeout as small as possible, because they don't want resources to be kept in a reserved state for too long. This is mostly a matter of tuning, depending on which resources is the service offering and how many clients may be interested in those. We will explore this dimension in a later chapter.

The next chapter will dive into the architecture of how TCC can be implemented. It will especially focus on the $4 + 1$ architectural views.

Chapter 3

Architecture

3.1 General Overview

There are three main ways in which we can see TCC at work. One could be what has been shown in the examples of the previous chapter: a client that want to purchase something and has to deal with a TCC-compliant service. In this case we would have a client extended with a TCC transaction manager and a server that can perform TCC transactions. A generalization of this concept brings a second possibility of implementing TCC: we can assume the transaction manager as a more general service that lays on the cloud, thus the client will only be extended to the point in which the transactions in progress will be dispatched to the transaction manager. When client is done and wants to confirm (or delete), it will dispatch the required actions to the transaction manager.

Finally, a third possibility is the interaction among servers, thus the client itself becomes a server and requires resources from another server in a TCC fashion.

During the development of my thesis, these dimensions were all explored. In this chapter we will see the architecture of the system; for the most part I show the views for the generalized TCC (with the transaction manager on the cloud) because it's the most interesting among all.

3.2 Logical View

The logical view of a system decomposes the system structure into software components and connectors. It's used to map functionality, use cases and requirements onto the components. It concerns functionality and it's target audience are primarily users and developers.

The browser extensions has primarily two components: a GUI component which helps the user in tasks like logging in or registering into the transaction manager service and the extension filter. The task of the extension filter is to filter out confirmation links from HTTP headers received when reserving an item on a TCC service. As we can see the GUI component is directly connected to the login/registration component, which is used to register or log in into the transaction manager and to the transaction manager itself, to see current transactions in progress. It is not connected to the TCC service because it's not the GUI itself that dialogues with it, but the browser.

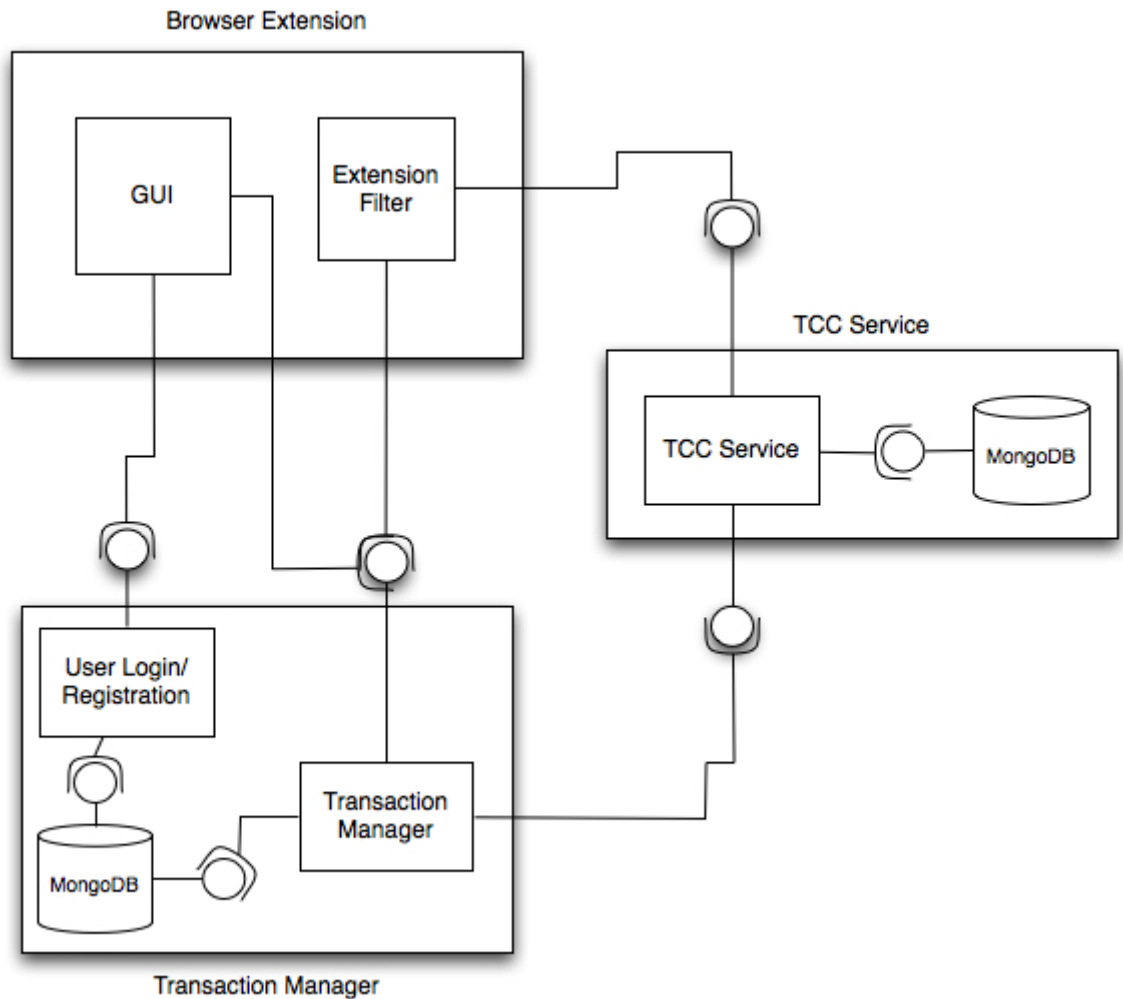


Figure 3.1. Logical View of the system.

The TCC service is composed by a big homonym component and a MongoDB database. The component's task here is to work as any possibly service that offers resources, with the only difference that when a **POST** request arrives for a resource, it does all the work to make it TCC compliant (explained in chapter 2). When answering with a confirmation link, the latter is intercepted (filtered) by the extension filter, which sends it to the transaction manager that will store it.

The transaction manager is composed by a small piece which deals with registrations and logins, and a big component which receives confirmation links and stores them. When questioned, the transaction manager component sends back the list of current transactions in progress to the GUI. When a confirmation request is asked by the GUI, the transaction manager confirms the transaction directly to the TCC service component using a **PUT** request on the confirmation link.

For what concerns the logical view for an integrated transaction manager into the extension

itself, this may turn as simply as a local storage for the extension that keeps track of the current transactions in progress. A registration/login component will not be needed since the transaction manager has to deal only with the transactions of the current user. In the case of servers interacting using TCC we have the same exact pattern except that the extension is not a browser extension per se but a server.

3.3 Process View

The process view models the dynamic aspects of the architecture: it defines which are the active components, which are the current threads of control, if there are multiple distributed processes in the system, what is the behavior of the various parts of the system and how all of them communicate. It concerns functionality and performance and the target audience are mainly developers.

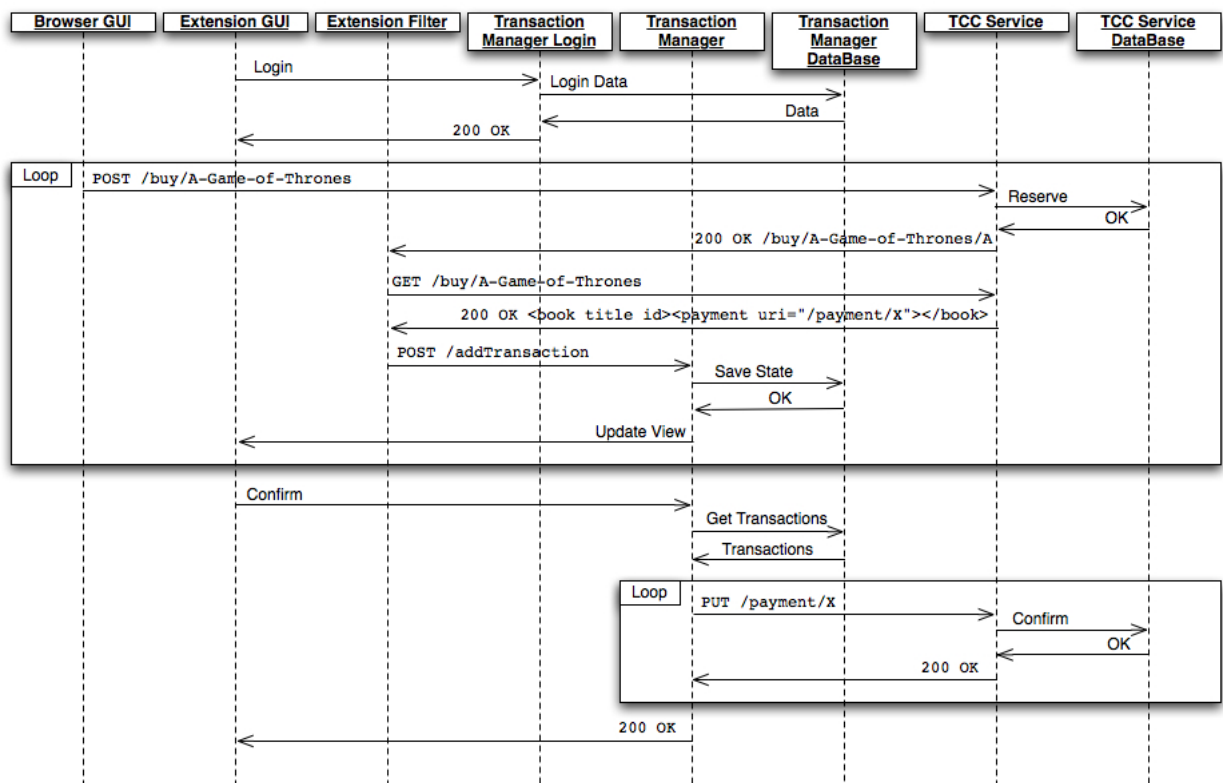


Figure 3.2. Process View of the system.

In figure 3.2 we can see the previously described components in a general working flow. The first thing that we want to happen is the login from the extension to the transaction manager. Once this is done, the reservation loop can start. Basically what happens is that from the browser GUI, the user may **POST** a reservation for an item. The TCC service sets the item as reserved, creates the resource and sends back the confirmation link to the

browser. The link is subsequently intercepted by the extension filter, which does **GET** on it to receive more information about the current transaction. As it receives them, they are sent to the transaction manager through a **POST** request. The transaction manager stores them and updates the view that shows to the client the current transactions in progress. This loop may go on for a while, the user may reserve as many items as he wants. When done, the user can confirm from the extension GUI the various transactions. A **PUT** request is forwarded to the transaction manager, which gathers all the transaction for that user and starts a second loop that confirms, through a **PUT** request on the various confirmation links, the transactions to the TCC service(s). Once done the transaction manager can notify the client that the confirmation happened successfully (or not).

As for what concerns the two variants of TCC, this workflow doesn't change. In the case of server interacting with another server there won't be any browser GUI, and if the transaction manager is local then login is not necessary either, but except for that the order of the actions and the actions done are the same.

3.4 Development View

The development view of a system is the static organization of the software code artifacts (for example packages, modules, binaries, etc.). Basically it maps the code with the logical view and it concerns reusability and portability. Target audience are usually developers.

As we can see in figure 3.3 for what concerns the browser extension, the GUI is done in plain HTML while the extension filter that intercepts the confirmation links is done in JavaScript. The service as well as the transaction manager are developed in JavaScript using as framework Node.js. Node.js is a server side framework to write servers in JavaScript. The choice of using it instead of any other framework in any other language was taken because creating a service with all the functionalities that were needed to implement TCC was very simple and straightforward. Same goes for the transaction manager and its functionalities. The two databases were implemented in MongoDB.

Of course any kind of choice regarding client and server sides is possible, given the correct implementation of the TCC protocol.

3.5 Physical View

The physical view defines the hardware environment (hosts, networks, storage, etc.) where the software will be deployed. Different hardware configurations may be used to provide different qualities. It concerns performance, scalability, availability, reliability and the target audience are operations.

As shown in figure 3.4, HTTPS is the main channel through which communication is performed. I shall repeat that this configuration corresponds to the general TCC case described in 3.1. The browser can communicate to the TCC service with HTTP too, but when doing the **POST** request, and receiving the result back (filtered by the extension), it's important that it's set to HTTPS, to avoid eavesdropping of the link. If the link gets eavesdropped, for an attacker it could be easy to confirm some reservations even if the reserver wants to cancel it afterwards. Same goes for the communication between the extension and the transaction manager (in which confirmation links are sent). As for the

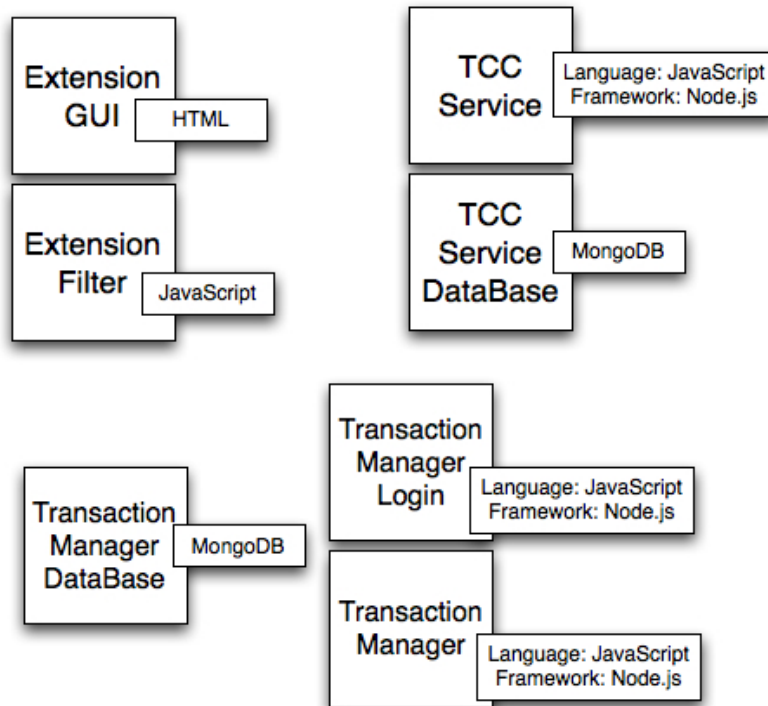


Figure 3.3. Development View of the system.

confirmation event we can assume that in the worst case, an attacker may want to intercept the various **PUT** request, send back a bogus result and send a **DELETE** to the confirmation link to cancel the transaction, instead of confirming it, thus again the use of HTTPS. For what concerns a physical view for the other cases, again the use of HTTPS is encouraged to avoid man-in-the-middle attacks and eavesdropping if the protocol is used on a web-based service. As for the case in which different servers works on the same network or same machine, the use of a secure transfer protocol is highly suggested.

3.6 Use Cases

A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system. The use cases this section is going to dive through will be useful to understand the flow of actions in the TCC protocol from the point of view of the user.

Note that these use case scenarios only describe the general TCC case, and do not explain the case in which the transaction manager is on the machine. This choice has been made because the general TCC case is also the most complicate in terms of elements in play, thus results more interesting to see.

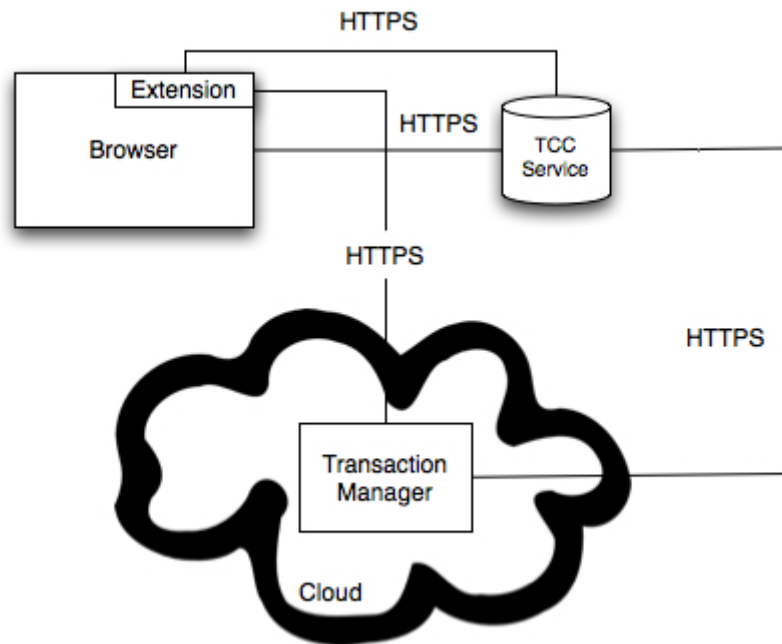


Figure 3.4. Physical View of the system.

3.6.1 Immediate Purchase

This use case explores the best case for user and for the sellers: a user performs a couple of reservations from the same website and confirms them right away. The service is happy because the user confirms as soon as possible, and the user is too because transactions won't timeout before he confirms them.

As pre-condition, the user should be registered to a transaction manager service that manages his transactions. There are no post-conditions.

Scenario: the user logs in into the transaction manager service using the browser extension installed in his browser. Then he starts navigating the website of a airline agency. He wants to book a flight composed of two connecting flights from two different airlines. In the first website, he finds the first flight and books it. As soon as the booking is completed, in the browser extension the reservation appears stating the name of the airline agency, the booked flight name and the remaining time to confirm the reservation.

Shortly later, the user moves to the second airline agency and finds eventually the second flight. After booking it, the reservation appears right after the first one in the browser extension. Now the user is assured that he will take those two flights, so he clicks on the confirmation button on the browser extension, that shortly after tells him that the transactions happened successfully.

3.6.2 Deletion

The following use case scenario explores the situation in which a user reserves something but doesn't find what he was searching for in another website. Since the purchase of only

one of the two items he wanted to buy is meaningless, he deletes the order.

As pre-condition again, the user has to be registered to the transaction manager service. There are no post-conditions.

Scenario: the user logs in into the transaction manager service through the browser extension. First, he navigates to a website selling books: he wants to buy two books from the same fantasy series. In the current website he cannot find the first one, but manages to reserve the second one. In the browser extension now he can see his reservation. Later, he navigates to two different websites, but cannot find the first book of the series. Since he doesn't want to buy the second book without having the first one, he opens the browser extension and cancels the current transaction for the second book. The webstore makes the book available again and the user closes the pages.

3.6.3 Timeout Before Confirmation

In this use case scenario, the user will wait too much for confirming the reservations done. In this case the items will be available in the store again and he has to go through the reservation process once more to confirm them.

As for what concerns pre- and post-conditions they are the same of the previous scenario.

Scenario: the user logs in into the transaction manager service through the browser extension. Then navigates to a movie theater website. He wants to book a seat for a movie, and then book another seat for another movie right after the first one ended, in the same movie theater. The user easily finds both the movies and manages to reserve the two seats for the two movies. Right after he receives a phone call and forgets about the transaction in progress. He leaves the house. In the meantime transactions time out, the seats get available again and eventually all the seats for both shows get booked. As the user returns, he notices from the browser extension that the reservations are timed up, so he can't confirm them.

3.6.4 Timeout During Confirmation

This use case scenario explores the worst case, in which the user confirms some transactions but during confirmation some reservations time out so he end up having some transaction confirmed and some timeout.

As for what concerns pre- and post-conditions they are the same of the previous scenario.

Scenario: the user logs in into the transaction manager service through the browser extension. He navigates to three different webstores that sell PC components. On one of them he can find the best motherboards at the cheapest price, so he reserves one. On the second website he reserves two RAM banks and on the third one, he reserves the latest processor in terms of computing power. In the end he has on the browser extension three transactions with relative names and timeouts. After a while he decides to purchase and he realizes he has very few seconds left to confirm the transactions in progress. He decides anyway to confirm them. Some bad thing happens during the confirmation phase and some transaction got timeout before the confirmation arrives. The user ends up having bought two RAM banks and a processor but no motherboard.

Notice that this case is the *heuristic exception* mentioned earlier, in the end of chapter 2. This is an unrecoverable state, so the user has to navigate back to the motherboard website

and hope the motherboard he wants is not out-of-stock.

The next chapter will illustrate the components design and interface.

Chapter 4

Component Design and Interfaces

This chapter is focused on the general design of the components used in the implementation of TCC as well as their interfaces. It is divided into the three main parts of the general TCC implementation (see section 3.1): the client side (browser extension), the service side (any webstore implementing the TCC protocol) and the transaction manager. The sections are divided in two parts: the design part, which focuses on how the various part of the implementation of the TCC protocol have been designed to cooperate together, and the interface part which targets the means by which those components may be addressed.

4.1 Client Side

The client side is composed by a browser extended with an extension application that can manage to filter confirmation links and send them to the transaction manager. The components this section is going to focus on are the GUI and the filter, situated in the browser extension, and not on the coordination between the browser and its extension, nor the communication protocols used by the browser to contact the service and the transaction manager.

4.1.1 Design

As mentioned in the introduction of this section, the client side is designed with two main components: the GUI and the filter. The GUI's task is primarily to let the user see what's happening. At first, it needs to login to the transaction manager and store the cookie. Once this is done, the GUI will show the current transactions in progress for the logged user. From the GUI, the user can trigger the confirmation of the current transactions in progress or the deletion of one of them. So basically this component encapsulates everything that concerns the control of the transaction manager.

For what concerns the filter, this is a small component that filters the headers of every request done by the browser. If the filter detects a fingerprint sent by a service TCC-capable, then it searches for the confirmation link that should be stored in the header. Once the link is taken, it performs a **GET** request to receive additional information about the transaction, and sends them to the transaction manager. It's highly recommended to

performs these two steps asynchronously, to avoid the user to perceive response time and delays.

4.1.2 Interfaces

There are no real interfaces to which somebody or something can attach to both components. In fact, the both of them connects to some other interface, but they do not have one to be connected to (see section 3.2 and especially image 3.1). It may seem strange that the two components do not interact together. The reason is that they simply don't need to. The GUI logs in into the service and receives back the current transactions in progress. Those are filled one after the other by the filter component, that filters the confirmation links, gets the additional information about the transactions in progress and send everything to the transaction manager.

4.2 Service Side

The service side is a simple website that offers something to the users, that can be books, items, or plane flights. This section omits everything related to the design of such websites, but focuses its attention on how the service is modeled to become TCC-capable. We will see how it is designed and which interfaces gives it access to.

4.2.1 Design

To be capable of doing TCC transactions, the service has to have the power to "reserve" the item, instead of selling them right away. This can be done by creating a new state for the items that is neither bought nor available. The idea is that after receiving a **POST** request, the service creates a new resource that is "hid" by the means of an id. Ideally, this resource can only be accessed by the user that performed the reservation, since he is the one receiving the link to it (confirmation link). This resource should be susceptible to **GET**, **PUT** and **DELETE** methods, in fact with a **GET** request it returns some information (for example the timeout limit, the payment link, etc.), with a **DELETE** request, it cancel itself as resource (effectively canceling the reservation) and with a **PUT** request, it confirms the purchase and removes itself.

Of course, the work to create this new resource is not that simple. The service provider in fact may want to keep track of the current transaction status by setting a timeout, which corresponds to the timeout given to the user, which when expired makes the item back available. These timeouts have to subsequently made persistent by storing the current transactions in progress into a database, to recover from failures. This is just one way to recover current transactions from failures, from the point of view of the service side, other means are welcome, as long as they provide persistence and recovery of the transactions and timeouts.

4.2.2 Interfaces

The interface given by the TCC service are those given by any RESTful service on the cloud. The filter situated on the browser extension does not really "connects" to it, but uses the result of an HTTP method to check the possibility to execute the TCC protocol.

For what concerns the transaction manager, it uses again the interfaces given by REST to contact the service and either confirm the transactions or delete one of them.

4.3 Transaction Manager

The transaction manager is a service on the cloud; its work is to receive transactions in progress by users registered to it. Users may later confirm those transactions or delete them. The work of the transaction manager is to keep track of every transaction for any user, keep track of their timeouts and execute requests as the user needs.

4.3.1 Design

This last part is composed by two pieces, a small piece capable of registering or logging in users and the big part which stores transactions and executes methods on the service.

The login / registration component may be a simple web page with a form, or a RESTful interface that accepts **POST** requests which sends either a login or a registration form. In either of the two cases, this component should communicate to the database and register users or check for the login.

For what concerns the most interesting part, it has to be capable of receiving chunks of data which represent ongoing transactions. It has to store them keeping into account all the information received plus the user that sent the request. Once the same user requires back all his transactions in progress (that is from the GUI), the transaction manager has to retrieve all of them back from the database and send them to the user. Lastly, when the user decides what to do (either delete some or all of the transactions or confirm them), the transaction manager has to be capable of sending to the service either a **DELETE** request or many different **PUT** requests to many different services.

Also in this case, persistence and recoverability are key factors. The database storage helps when the transaction manager has a failure, since it can recover every ongoing transaction for every user. The same thing happen when it comes to a failure during the confirmation phase (see section 2.3.3), since the **PUT** operation is idempotent, the transaction manager can retry that phase on and on until the transactions are all confirmed.

4.3.2 Interfaces

The interface to dialogue with the login / registration component can be, as previously stated, a simple web page with forms as well as a RESTful interface that accepts **POST** requests containing the data to login a user or register him.

As for the transaction manager, the GUI of the browser extension needs data to be filled, so again a RESTful interface is given to give access to the current transactions in progress. Especially in this case a **GET** request is needed to ask for the ongoing transactions, while a **DELETE** request provided with the id of the transaction will trigger a **DELETE** for a transaction. Instead, a **PUT** request will trigger the confirmation phase.

Chapter 5

Detailed Protocol Design

Chapter 6

Implementation

This chapter dives into what has been done for the implementation of the TCC protocol. The chapter is divided into three main sections that describe the three ways in which TCC can be implemented. These sections are divided subsequently into three subsections defining the TCC actors. Each part describes in details how it has been implemented, including programming languages and libraries used.

For what concerns the browser application, I developed an extension for Google Chrome. The reason behind this choice lays in the readiness with which extensions are developed. Moreover, I was already familiar with JavaScript, language in which Chrome extensions are programmed, thus I thought it might have been a very good choice. I also documented on FireFox extensions, but I didn't liked the way programmers are meant to program extensions. Thus, the implementation of the two forms of extensions found hereafter are Google Chrome extensions.

6.1 General TCC Implementation

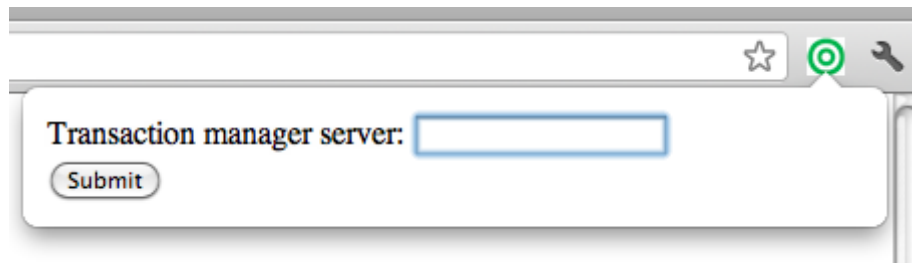
The general TCC implementation has in play a browser extension, some services implementing the TCC protocol and a server acting as transaction manager. This section describes in details how this implementation has been done.

6.1.1 Google Chrome Extension

This Google Chrome extension implies the use of a background page. This means that while navigating, there is always a background JavaScript file executing some methods. In this case, the method being executed is the filter method which was discussed in 4.1.1. Moreover, this extension also uses a popup page. The Google Chrome API let the programmers create popup pages, which are plain HTML, that can contain any kind of information: they are capable of getting data about the currently visualized page, as well as doing HTTP requests to gather data from somewhere else. The use I made of it is the latter, in this particular case I'm gathering information from the transaction manager.

GUI

For what concerns the GUI, everything is stored in the popup page. At the beginning the popup asks with a form which transaction manager server the user wants to use. When a URL is provided, the GUI contacts the server through a GET request done in AJAX but returning the whole page instead of plain XML or JSON. The reason behind this choice is the use of a simple library on the transaction manager; this will be fully explained later.

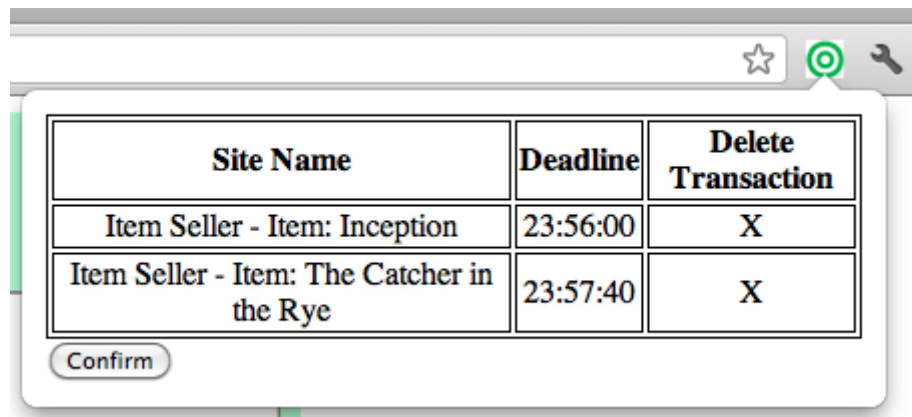


Transaction manager server:

Submit

Figure 6.1. Freedom of choice by the user for the transaction manager.

The server sends back the login / registration page, which is loaded in the popup and through which the user can login or register into the transaction manager service. Either of the two, the GUI loads the page containing the transaction in progress for the current logged user. If there are no transactions in progress a message informs the user of that. If instead there are transactions in progress, the user may confirm all of them clicking on a button or delete one or more by clicking on the delete button, provided for each of the transactions in progress.



Site Name	Deadline	Delete Transaction
Item Seller - Item: Inception	23:56:00	X
Item Seller - Item: The Catcher in the Rye	23:57:40	X

Confirm

Figure 6.2. Transactions in progress for the logged user.

One question that may arise is: how will the GUI react if a user performs a reservation without being connected or logged in to a transaction manager? What happens is that since the server receives a reservation, it performs it and sends back the confirmation link. The extension will notice that the user has not logged in into a transaction manager so it can notify it to the user. The item will yes kept reserved, but the confirmation link will be lost. This is not a problem, in fact as the timeout expires the item will be available again.

Filter

As mentioned in the introduction, the filter lays in the background page. Each time a request is performed and a result returned, the filter takes the result and checks if it contains a fingerprint of the TCC protocol. In that sense, the fingerprint is searched into the "Link" header of the HTTP response, and this fingerprint is `/XTCC?id=`. What comes after the equal sign is the id given to that particular transaction, while what comes before the fingerprint is the URI used to reserve the item. In this way the filter finds out a confirmation link, stores it and performs a **GET** request on the confirmation link through and AJAX call. This returns from the service more data regarding the transaction in JSON format. I have to specify that JSON format was asked in the header setting `application/json` in the "Accept" header. The server is also programmed to return XML, if needed.

On the callback of this **GET** request, the extension gathers the data and send it to the transaction manager through another asynchronous request (this time a **PUT**), which, if the user is logged in, will store the transaction on the cloud.

One may think "why not sending directly all the data when sending the response of the reservation?". The reason is that the server may want to avoid sending unnecessary data (for example the name of the item reserved, the name of the website on which the item was reserved, the date in which it happened, and so on), while the client may not be interested in it and just wanting to know the remaining time to complete the transaction and the confirmation link, as well as the payload reference.

An interesting case to focus on is when a reservation is triggered but the service appears to be down when doing the **GET** request. In that case, since **GET** is an idempotent method, I used a trick: until no response is received, the filter keeps sending **GET** requests doubling the time waited from one operation to the other each time one is performed. So for example the first time it will wait 1 second, then if no response is received it will wait 2 seconds, then 4 seconds, and so on. The reason is to avoid putting pressure on the service and avoiding infinite loops on **GET** requests by the Google Chrome extension. This trick is called *delayed request execution* and it's also used when, after receiving the additional data, the filter tries to send it to the transaction manager. If at that particular time the transaction manager is down or is experiencing major delays, the delayed request execution will delay step by step its **PUT** requests (which, again, are idempotent) until the transaction manager responds. Delayed request execution is used also in other contexts regarding TCC, especially in the transaction manager's work.

6.1.2 Node Service

The service I implemented is a very simple dummy item seller, which sells any kind of item. It is implemented with Node.js, the server side JavaScript framework. I've chosen to work with it because Google Chrome works all in JavaScript, so I preferred to keep the same programming language all over to avoid issues like translating data models from one language to the other. Moreover, it's really simple and fast to create a server, so that I could have quickly tested my TCC implementation.

This services has nothing different from any other webstore. It implements MongoDB as database, and makes use of Mongoose, a library that makes easy the interaction between the server and the database. The database is used to store the items sold by the webstore and the informations about current transactions in progress. The service also makes use of

Express.js, a high performance framework to simplify even more the creation of a RESTful service in Node.js.

When a **POST** request is received to reserve an item, the service creates a new resource, accessible through a URI provided with a unique id. This URI is what is called confirmation link and it responds to **GET**, **DELETE** and **PUT** requests. As soon as the resource is created, first it is added to the database, storing the timeout, the name of the item (or the item id) and the unique id created, and then the total amount of items of the kind of the one ordered is decreased by one. Then a timeout is setup: this timeout when expired will make the reserved item available again, thus effectively increasing the total amount of that item by one.

When a **GET** request is received on a transaction resource, first the server checks in the database if there is a reservation with that id, then if there is, it makes up either a JSON or an XML (depending on the "Accept" header) containing all the information about the item being reserved and the reservation, then sends it back as response. If there is no reservation with that id, the server just ignores the request, since it may be the case that the reservation timed out and was removed from the database.

When a **DELETE** request is received on a transaction resource, again the server checks in the database if that reservation still exists, and if it is the case, then it removes the database entry for that reservation and increases the total amount of the item reserved by one in the item database. This is what happens also in the case of a timeout. When a **PUT** request is received, the server checks if the reservation still exists, then if it is the case it removes the entry in the reservation database and starts the payment procedure. It is as simple as that since the total amount of items available was already decreased when the item was reserved and the server doesn't even need to remove the timeout in progress, because the timeout itself when expiring will check if the transaction is still there, and if it's not the case it means that either it has been confirmed or canceled, so no operation is performed. What happens when the service goes down? Thanks to the database storing all the information about the reservation in progress, the service can set up all the timeouts again as soon as it is up and running, effectively not losing anything. If a timeout expires while the service is down, as soon as it is in the process of setting timeouts up again, it will check if the timeout expired, and if it's the case it will follow the same procedure as when a timeout expired or a **DELETE** request is received.

6.1.3 Node Transaction Manager

The transaction manager I implemented is also very simple. Also in this case, MongoDB was used as database, as well as Mongoose and Express.js as library and framework for the implementation.

The transaction manager has a page where users can login or register to the service. This page is an HTML page, so users may login either from the browser and from the Google Chrome extension. Once logged in (or registered), users may see (again either from browser or Chrome extension) the current transaction in progress.

As explained in 6.1.1, the transaction manager has an interface to receive **PUT** requests from the Google Chrome extension. When a request is received, it creates an entry in the database with all the data received (i.e. confirmation link, timeout, etc.). It also sets up a timeout, like the service, which when expired removes the transaction from the database. In this way the transaction manager is always aware of which reservation are still in progress

and which are timeout (thus effectively updating the view).

When a **GET** request is received, if the user is logged in the transaction manager shows the current transaction in progress. From this page, the user can either confirm all transactions in progress or delete one (or more). If the user decides to delete a transaction, a **DELETE** request is triggered to the transaction manager with the id of the transaction. The transaction manager first checks if the reservation is still there (since it may have timed out in the mean time), then if found, it removes it from the database and sends a **DELETE** request to the confirmation link. If no response is received, it means that there is some problem with the service. In this case it makes use of the delayed request execution, which will guarantee that sooner or later the **DELETE** request will reach the service. Since also **DELETE** is an idempotent method, we do not have to worry if it has been received more than once at the service.

When a **PUT** request is done for a confirmation of all the transaction in progress, the transaction manager first gathers all the transaction in progress for the user that just sent a confirmation. Then it starts sending **PUT** requests on the confirmation links. Again, if one or more services are busy or down, the delayed request execution guarantees that sooner or later the **PUT** request will be received by the services.

For what concerns timeouts, the same holds also in the transaction manager: the timeout triggers automatically the delete process triggered by a **DELETE** method request. When confirming (or deleting) the timeout may be left there, since as it expires, it first checks that the transaction (for which it has been set up) is still there. If it is, delete process, if it isn't then nothing is performed.

One may wonder what happens if the transaction manager fails and goes down. For what concerns the receiving the **PUT** requests from clients, as specified in 6.1.1, the delayed request execution from the Google Chrome extension will guarantee that the transaction manager will receive the reservations as soon as it gets up and running. If the failure happens before the confirmation phase, then thanks to the database persistence, the transaction manager may setup again the timeouts and has all the information about the current transactions in progress.

In the worst (and most interesting) case, the transaction manager may fail during the confirmation phase. What has been done in this case is the following: as soon as transaction manager gathered all the transaction in progress for the user that triggered the confirmation, it sets each reservation as "pending" in the database. When a transaction is confirmed and a response is received, the entry in the database for that transaction is removed. If the transaction manager goes down during this process, when getting up it will notice that some reservation are in "pending" state. This means that it was confirming them but something happened. In this case it will start confirming all the pending transactions again.

6.2 Transaction Manager integrated in Browser Extension

This version of the TCC protocol has the transaction manager integrated into the Google Chrome extension. For what concerns the service, I used the same service described in 6.1.2 without changes. This may show how the service side of the protocol may stay invariant while we have different possibilities at the client side.

6.2.1 Google Chrome Extension

This particular version of the Google Chrome extension has the transaction manager integrated. This means that no service is needed and especially no login. The GUI is similar to what shown in figure 6.2, and if there are no transactions in progress, the GUI will notice it to the user.

Also the filter element is the same, but instead of sending a **PUT** request to a transaction manager on the cloud, it stores everything in the local storage. Local storage is a form of web storage on the client side. It is accessible through JavaScript and it's a simple key/-value pair storage. It is persistent, thus if there is a failure in the client side, or simply the client is shut down, the storage remains until the user wants to use it again.

Storing and retrieving information from the local storage works by just doing a couple of method calls, no database needed. This is a very powerful tool, because in this way the transaction manager has a very simple work. When transaction are asked (the popup of the extension is opened), it checks in the database if there are transactions in progress. If there are, it shows them. If a reservation is deleted, the transaction manager takes the confirmation link of that transaction and performs a **DELETE** request, and the transaction is removed from the local storage. If instead the confirmation is triggered, the transaction manager takes all the transactions from the database and executes **PUT** requests on their confirmation link.

For what concerns persistence in failures during confirmation phase, since the storage is a key/value pair which accepts strings, JavaScript objects are stored as plain JSON. To these object then I can add as many parameters as needed. During the confirmation phase the same "pending" state is added (as the one in 6.1.3), and when the transaction is confirmed, it is also removed from the local storage.

During confirmation and deletion, as well as when asking for more information to the service, the same principle of the delayed request execution is used, so until no response is received, it will try to execute the requests in a delayed fashion.

6.3 TCC used in server interaction

The previous sections have shown how the TCC protocol can be used to control transactional interactions between a server and a client. This idea can be extended and applied to the interaction between servers with the same results: when a server needs some resources from within itself (in large applications) or from other servers, it can use the same exact protocol, having the transaction manager local to avoid network delays. The result is a server application capable of taking different resources at once when needed without the worry of gathering just some of them but not all.

The following subsections will introduce how the TCC idea was extended to server applications. First there will be a short introduction to S, a new programming language, on which the TCC construct is applied. Then the following two subsections will describe the TCC library and the transaction manager library created to support TCC on S.

6.3.1 Introduction to S

S is a new domain-specific programming language developed at the Università della Svizzera Italiana. It targets server side scripting of high-performance RESTful web services. S brings

an innovative programming model, based on explicit and implicit parallelism control, which allows the service developer to specify which portion of the control-flow has to be executed in parallel. The choice of the best level of parallelism is then left to the runtime system, for each service. S compiles in Node.js, thus the executable are simple JavaScript files runnable by Node. What has been done in the context of S is the following: since S compiles in a Node.js executable file, I took what was a simple set of operations done on a service TCC-capable (see 6.1.2) and created, through S, a server that contacts the service and reserves a couple of items. What was created in pure S was just the **POST** request to the service. What I manually added later are a TCC library to support TCC transactions and a transaction manager library to support the managing of transactions in progress. The idea is to demonstrate how, applying a small change to the compiler, we can compile S and generate code that supports TCC just by adding the following two libraries.

6.3.2 TCC Library

This library is made of a constructor, an init method and a reserve method. The constructor returns a new object of the type TCC, which is the type defined by the library. The init method takes a transaction manager, a URI and a scheduler. The transaction manager is needed when a confirmation link is retrieved. The idea is that once this TCC object has filtered a confirmation link, it directly stores it into the transaction manager. The URI is the address to which we perform a **POST** request to reserve items on the service. The scheduler is a construct from S which is used to schedule subsequent actions when this one is complete (e.g. after this execution the scheduler will emit an event that will trigger the next execution).

The reserve method takes as parameter a event, which will be the input of the emit method of the scheduler, which will be called when the execution of the reservation is over. The reservation is done in a similar fashion to what the client in the Google Chrome extension does. First, it creates a **POST** request to the URI provided in the constructor. Then when the result is received, it is filtered to gather the confirmation link. With this, a hidden method is called to gather more information about the transaction (the **GET** request). I call this method "hidden" because it is not visible to the outside of the library. After the result of the **GET** request are gathered and ordered, they are set into the transaction manager. Finally, the scheduler emits the event passed as parameter.

If some error occurs during the reservation process, the scheduler emits the event and an error is logged. For what concerns failure resistance in this case, still nothing has been done. One idea could be to notify in some way the transaction manager object with the link of the resource that has not been able to contact. In this way we may set up an ideal system in which if more than an input number of reservation fails, the transaction manager does not confirm the transactions at the end.

6.3.3 Transaction Manager Library

The transaction manager library is made of a constructor, an init function, a function to store links, a function to delete transactions and one to confirm them all. The constructor and the init function have no parameter. The first one simply returns an object of the type TransactionManager. The second one creates the array of confirmation links. The function to store link takes as parameter an object containing the confirmation link as well

as other informations about the transaction in progress, the link is taken and added to the previously created array, then a timeout is created which, when fired, removes the object from the array. In this way when confirming, if one timed out, it will not be confirmed.

The function to delete transactions takes a confirmation link and performs a **DELETE** request on it in delayed request execution fashion. The same trick is used when performing a confirmation through the confirm method. It takes all the links stored in the array and performs a **PUT** request on them with the delayed request execution. If there is some timeout during confirmation it is kept into account.

One may notice that there is no persistence in all of this. Persistence in this case can be made possible by simply adding a database storage to the library, but the purpose of this study was to demonstrate how easily one can extend his own server with the TCC protocol. Many implementation are possible, the reader is invited to develop and explore his own.