Acta Cybernetica **25** (2021) 435–468.

Traquest Model — A Novel Model for ACID Concurrent Computations*

Dániel B. Rátai[ab], Zoltán Horváth[ac], Zoltán Porkoláb[ad], and Melinda Tóth[ae]

Abstract

Atomicity, consistency, isolation, and durability are essential properties of many distributed systems. They are often abbreviated as the ACID properties. Ensuring ACID comes with a price: it requires extra computing and network capacity to ensure that the atomic operations are done perfectly or they are rolled back.

When we have higher requirements on performance, we need to give up the ACID properties entirely or settle for eventual consistency. Since the ambiguity of the order of the events, such algorithms can get very complicated since they have to be prepared for any possible contingencies. Traquest model attempts to create a general concurrency model that can bring the ACID properties without sacrificing a too significant amount of performance.

Keywords: ACID, concurrency, consistency, atomicity, concurrency model, fault tolerance

1 Introduction

In the case of the microservices architecture [5] when we send a request, it can initiate some modifications in the global state in a transactional way. Microservices are mainly based on the request-response model. When the Request returns with no errors, that means the modifications in the global state are done, and the transaction is over. While if the response is an error, that means there were no

*The research was supported by the ÚNKP-20-3 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. The research is part of the "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme no. 2020-4.1.1.-TKP2020 (National Challenges Subprogramme) funding scheme.

[a]Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary
[b]E-mail: `danielratai@inf.elte.hu`, ORCID: `0000-0002-9587-571X`
[c]E-mail: `hz@inf.elte.hu`, ORCID: `0000-0001-9213-2681`
[d]E-mail: `gsd@inf.elte.hu`, ORCID: `0000-0001-6819-0224`
[e]E-mail: `toth_m@inf.elte.hu`, ORCID: `0000-0001-6300-7945`

DOI: `10.14232/actacyb.288765`

modifications in the global state at all. Requests can make other requests, and more complex transactions can be assembled.

In a request-response model, the ACID properties come at a high price. The service which calculates the response has to guarantee that any write operations arising must be synchronized, committed, and persisted before it can reply with a response. Of course, on the other hand, if ACID is not a requirement, the service can be very fast. In this case, the service can just read and write the state of the local server and answer to the client immediately. Later, the server can synchronize the writes if eventual consistency is a requirement, but this does not block or decelerate the original process.

However, it is not just the performance that can cause a problem. It is very challenging to ensure atomicity itself when we nest the services. Figure 1 shows a scenario where we have service A calling two other services, B and C. The order of the network events is marked on the figure. The client sends a request to service A, which sends requests to service B and C. Service B responds correctly, but C responds with an error. In this case, we can assume that C has rolled back correctly, but B should be rolled back as well. There is no mechanism to roll back a request in the request-response model after it has been responded. Therefore it is hard to chain more services properly when atomicity is a requirement. We can be sure to have a proper response if the happy path happens, but if there is an error arising at some of the chained requests, our system can get easily stuck into an invalid intermediate state.

It seems there is a hard dilemma between ACID properties and efficiency. The proposed Traquest model attempts to resolve this dilemma and therefore improve the efficiency of the ACID systems.

The phrase Traquest [28] comes from the words Request and Transaction. The core of the idea comes from the microservices architecture, and the Traquest model

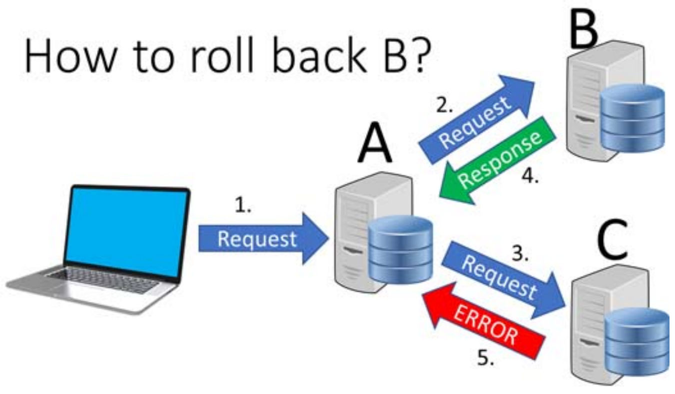


Figure 1: Nested rollback issue

is something similar to the request-response model. We can send requests to a Traquest, and the Traquest replies with an answer, but here the answer is not a simple response message, but rather an established parent-child connection between the two Traquests with a temporary response, a so-called Trasponse. When a Traquest gets a request, it can immediately carry out read and write operations on the local server. It can immediately reply with a Trasponse; however, of course, that still might take time to synchronize the effects of the operations with other servers. Therefore Trasponse is only a temporary response.

Before creating the Traquest model, we have examined many of the existing technologies and solutions, including multitier architectures, actor model based systems, different consistency protocols, and different papers discussing the limitations of distributed ACID systems. We have found that the current systems have the strict limitations we described above. We decided to investigate whether it is possible to create a system based on the idea that a response can have a temporary nature. The Traquest model was realized during this research process.

We created the concept of the Traquest model, and we also built an experimental prototype in TypeScript. Adjustments on the model might become necessary later as further, and more comprehensive implementations will be created in different programming languages. However, the current results show that the general concept of the Traquest model is viable. Traquests can provide ACID computations using magnitudes fewer network messages in some concurrency scenarios than the current technologies.

This paper is structured as follows. In Section 2, we give an explanation of the Traquest model. In Section 3, we discuss some state-of-the-art solutions and how the current technologies were used to solve problems related to the ACID properties. We explain further the Traquest model through an exemplary case and compare it to the current technologies. In Section 4, we will highlight the current challenges and further research directions. Finally, this paper concludes in Section 5.

2 The Traquest model

The request-response model is used on a local level as well and not only between different computing nodes. Asynchronous callback functions can behave equivalently. We can send the request content and the callback function as an argument, and the callback function can contain the response in an argument. This mechanism is often used to wrap network-based request responses, but for local asynchronous operations as well.

However, callbacks can get complicated when they are heavily used, and we want to handle exceptional scenarios. To this end in computer science, *Future*, *Promise*, *Delay*, and *Deferred* refer to constructs used for synchronizing program execution in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is not yet complete. The term *Promise* was proposed in 1976 by Daniel P. Friedman and David Wise [9] and Peter Hibbard called it *Eventual* [16]. A somewhat similar concept *Future* was introduced in 1977 in a paper by Henry Baker and Carl Hewitt [3].

Traquests behave most similarly to *Promises*; therefore, we use them as a baseline for the explanation. Traquests, just like Promises, are placeholders for a temporarily unknown value. Traquests, just like Promises, can be nested and depend on each other. However, once a Promise returns with a response, this response is final, and it cannot be modified afterwards. On the other hand, Traquests can be strongly bonded together to form a tree structure, a so-called Traquest tree. A Traquest tree creates the transaction, and if any Traquest fails in the Traquest tree, all the Traquests are failing. When the Traquests are failing, they are not just returning an error, but they are ensuring that if they created any modification, it would be appropriately rolled back so that the global state of the system will not be affected by half-done transactions. To be able to achieve this, Traquests are containing some additional mechanisms.

2.1 Structure

To understand how Traquests are working, first, we need to see the fundamental structure of the state of art Promises.

2.1.1 Promises

Figure 2 shows the fundamental structure of Promises. Deferred describes a yet unfinished work which is the asynchronous process that has to be done to get the

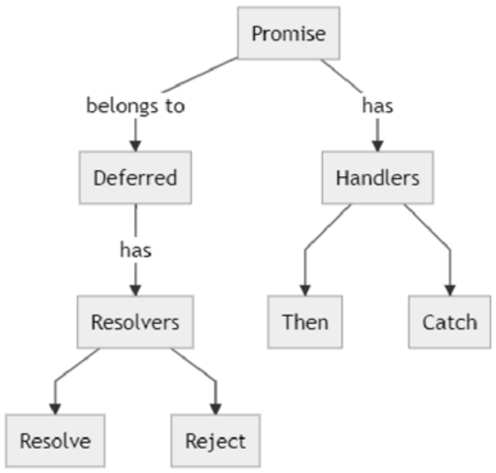


Figure 2: Promise structure

final value of the Promise. A Promise belongs to a Deferred. When the Deferred finishes, it can call different resolvers depending on whether the execution was successful or some exceptions were arising. If the execution was successful, the Deferred calls the Resolve resolver; otherwise, it calls the Reject resolver. The Promise itself is the placeholder of the yet unknown value. It has two handlers to handle the event when the unknown value becomes known. The Then handler is responsible for handling the successful resolution of the Promise, and the Catch handler is for handling the exceptions.

2.1.2 Traquests

Figure 3 show the fundamentals structure of Traquests. A Traquest also belongs to a Deferred that, similarly to Promises, describes a yet unfinished work. Traquests have handlers just as Promises to handle the event when the Deferred returns. However, Traquests has a third significant component as well, the Binding mechanism. The binding can permanently bind together Traquests in a parent-child tree structure. This binding holds until the whole atomic transaction finishes. Like that, a Traquest tree can act as a single entity, and it can form a complete atomic transaction, which can be distributed to many computing nodes.

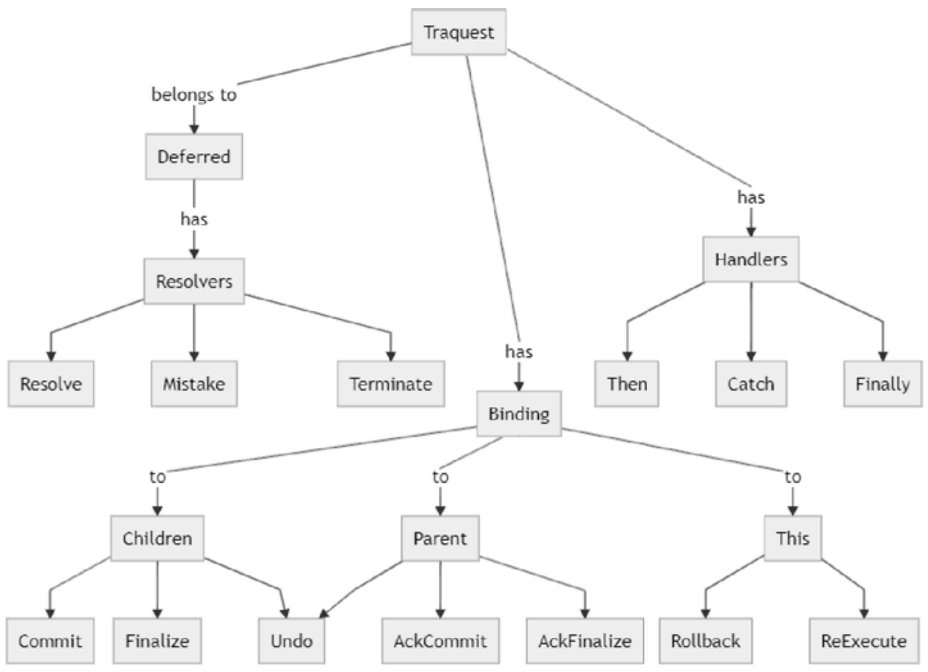


Figure 3: Traquest structure

The Deferred has the following resolvers. The Response resolver is the same as the Resolve resolver at the Promises. This should be executed when the asynchronous Deferred process successfully finishes with the proper value. The Mistake resolver is slightly different from the Reject resolver of the promises. The Mistake is called when a temporary failure happens. If there is a chance that the failure has occurred only because of the wrong order of the asynchronous operations, then Mistake should be triggered. Mistakes can be undone later, and the Traquests might rerun in proper order. The Terminate resolver is used in case of final failures. This resolver terminates the whole Traquest tree and tries to roll back all the Traquests in the Traquest tree.

The Then handler is the same as the Then handler of the Promises. The Catch handler is similar to the Catch handler of the Promises, but it is used explicitly for the mistakes. It can also avoid spreading up the Mistake to parent Traquests or let it spread further. The Finally handler is called no matter if the Traquest was properly committing or it was terminated.

The Binding mechanism of the Traquests has the following concepts:

Parent-child binding – When a Traquest had been created, the reference to the parent Traquest should be defined. If it is not defined, that means the created Traquest will be the root of the Traquest tree.

Undo – A mistake happens when an exception occurs because the Traquests are executed out of order. However, it can happen that the Traquest has already responded with a seemingly correct response, and an out-of-order conflict turns out only later. In this case, an undoing mechanism can be executed, which rolls back the necessary Traquests on the affected branch of the Traquest tree and re-executes them.

Rollback – A callback is provided for the case when the Traquest needs to revert the changes it has made so far. If the Traquest did not create any changes directly to the global state, just by calling other Traquests, this part could be omitted because the rollbacks spread automatically on the Traquest tree.

Finalizing – It is a mechanism used when all the Traquest in the tree have returned, and the result of the Traquests can be finalized. This happens completely hidden and automatically when all the Traquests in the tree have returned.

Committing – It is a mechanism used when all the Traquests in the tree have been finalized, and a final commit can be initiated. This happens completely hidden and automatically. The Committing mechanism combined with the Finalizing mechanism gives a similar process to the two-phase commit protocol [35]; however, there are differences because the Finalizing and Finalized states are handling the potential rollbacking Tail Traquests as well.

2.2 States

Promises and Traquests have different states throughout their life-cycle, which describes their behaviour.

2.2.1 Promises

Figure 4 is a state diagram that shows the possible states of a Promise. In the case of the Promises, we have three very simple states. We have an Unfulfilled or Pending state while the Deferred process is running, and the Promise value is not known. From this state, the Promise can step only to Fulfilled or Rejected state. This happens when the Deferred process finishes depending on whether an exception was arising or not.

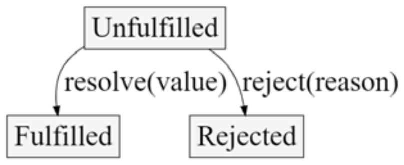


Figure 4: Promise state diagram

2.2.2 Traquests

Figure 5 shows the simplified state diagram of the Traquests. Traquests have to handle more complex scenarios; therefore, they can have more different states. For the sake of simplicity, the Terminated state and some state transitions are not

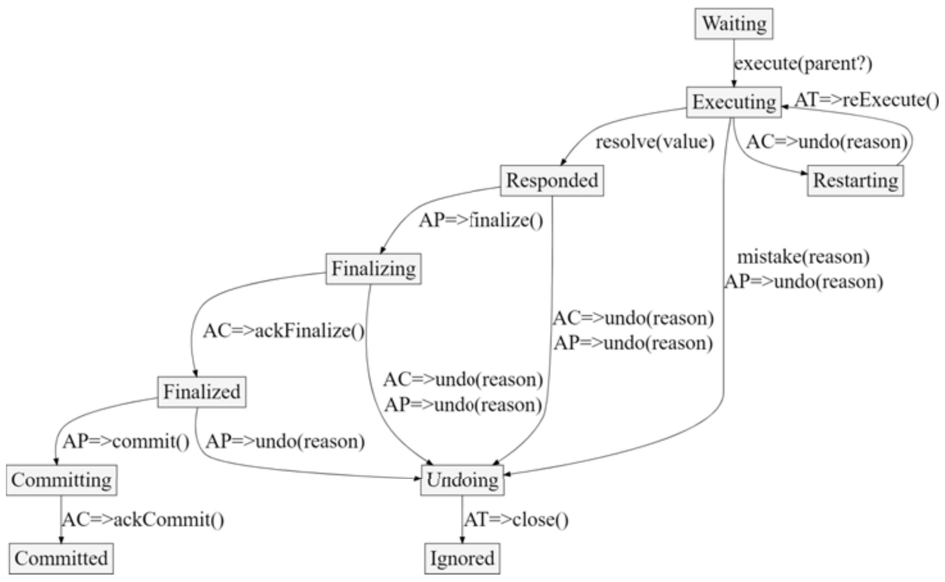


Figure 5: Simplified Traquest state diagram

shown here, only the most relevant ones which are necessary to understand the working mechanism of the Traquests. In Figure 5 the explanation for the state transition prefixes are the followings:

- **AC**: Automatically triggered by the child Traquest
- **AP**: Automatically triggered by the parent Traquest
- **AT**: Automatically triggered by the current ("this") Traquest
- **No prefix**: Manually triggered in the Deferred process

As one can see, Traquests are much more complex and have much more states than Promises have. However, Traquests have only two main manual resolvers, just like the Promises. When we have nested Promises, and an exception occurs, we should ensure manually that the proper Reject resolver is called, and it is handled at the parent Promise [31]. Furthermore, the parent Promise should manually escalate the exception further by calling its own Reject resolver. Traquests, on the other hand, are bound together and escalate the Mistakes automatically. A traditional exception in the Deferred process can be automatically converted into a Mistake, or a Mistake can automatically come from a child Traquest. In either case, no manual intervention is needed. Most of the time, it is enough to define only the happy path in a Deferred process and call the Resolve resolver. Interestingly this means that even though Traquests are more complex than Promises, still using Traquests can be even more convenient. To understand more how Taquests work, let us investigate them state by state in more detail.

Waiting state: The Waiting state is the initialization state of the Traquest. In this state, the Traquest gets initialized by defining the Deferred process belonging to it. The Deferred process is not added automatically to the event-loop like in the case of the Promises [20]; instead, it has to be manually executed. During the execute call, we need to provide the parent Traquest optionally. If the parent Traquest is missing, the current Traquest will be the root of the Traquest tree; therefore, the current Traquest will represent the overall transaction.

Executing state: When the execute method is called on the Traquest, the Traquest begins to execute the Deferred process, and it steps into the Executing state. During this state, the Deferred process can create new child Traquests and bind them to the current Traquest. The transaction can grow this way recursively.

During the execution, the Traquest can be interrupted by parent or child Traquests. The interruption happens if a Mistake arises at the ascending or descending Traquests. If a parent Traquest has a Mistake, that means the current Traquest should be ignored, including its children. Therefore if the parent Traquest triggers an undo operation on the current Traquest, it changes its state to Undoing, and triggers an undo operation on all the descendant Traquests.

If a Mistake is coming from one of the children of the current Traquest, that means that the Mistake is still not escalated to upper levels in the Traquest tree, and the Traquest can try to re-execute itself. In this case, the Traqest steps into the Restarting state.

The Mistake can come not only from the bounded Traquests, but it can also happen locally in the Deferred process of the current Traquest. In this case, the Traquest should roll back its changes; therefore, it has to switch into Undoing state.

Suppose no Mistakes are coming from the bounded Traquests, and the current Traquest is not running into a Mistake either. In that case, the Deferred process should call the resolve operation with the proper resulting value when it finishes. When the resolve operation is called, that means the current Traquest has completed its desired asynchronous task, and it can respond with the required value; therefore, the Traquest should step into the Responded state.

Responded state: The Traquest can get into the Responded state only from the Executing state by the Deferred process calling the Resolve resolver. Before stepping to the Responded state, the Traquest sends the result value coming from the Resolve operation to the handler of the Traquest. The handler executes the callbacks bounded to the Traquest by the Then operation right after the Traquest steps into the Responded state. However, there is one exception. If the current Traquest is the root of the Traquest tree, then no handlers are called, and the Traquest steps into the Finalizing state immediately.

Otherwise, the Traquest waits in the responded state until the parent Traquest asks for a commit, or a Mistake is coming from any of the bounded Traquests. Parent Traquests obviously can still be in execution when the current Traquest responds. However, for synchronization purposes and keeping the system's overall consistency, children Traquests can still be in Executing state as well. We discuss this scenario in more detail in Section 2.6. This means that a Mistake can trigger an undo operation from any direction. When an undo operation is called on a Traquest, which is in a Responded state, it should not be able to commit anymore.

If no Mistakes are coming from any of the bounded Traquests, then the current Traquest waits until it gets a finalize operation from the parent Traquest. The Traquest can and should trigger a finalize on itself only if the current Traquest is the root Traquest.

Finalizing state: When a finalize operation is triggered on a current Traquest by the parent Traquest and the current Traquest is in the Responded state, the current Traquest switches to the Finalizing state and calls the finalize operation on its child Traquests. If there are no child Traquests, the current Traquest steps into Finalized state and sends an ackFinalize operation to the parent Traquest.

If the current Traquest is still in Executing state, it makes no action. In this case, when the Deferred process returns, the parent Traquest calls again the finalize operation on the current Traquest; hence the finalizing mechanism can continue. This scenario happens with the Tail Traquests, which we will discuss later in more detail.

An Undo operation can come from any bounded Traquests during the Finalizing state.

Finalized state: The Traquests step to Finalized state when they get an ackFinalize acknowledgement from all of their children. If the Traqest steps into the Finalized state, it calls an ackFinalize operation on its parent. If it has no parent – meaning that the current Traquest is the root of the Traquest tree – the

Traquest steps in to Committing state and sends a commit operation to all of its child Traquests.

Undo operation can come only from a parent Traquest during the Finalized state because this state means that no descendant Traquests are being in Executing state, which could serve as a root for launching the undoing chain.

Committing state: When a Traquest steps into the Committing state, it triggers a commit on its child Traquests and waits until it gets an acknowledgement from them. When all the children have responded with an acknowledgement using the ackCommit operation, the Traquest steps into the Committed state. If the Traquest has no children, it immediately steps to the Committed state without waiting for any other processes or Traquests.

Traquests cannot roll back if the Traquest has already stepped into the Committing state.

Committed state: When a Traquest gets acknowledgements from all the child Traquests, it steps into the Committed state. The Committed state is a final state, meaning that the life-cycle of the Traquest was finished.

At the Response state, we discussed that if the current Traquest is the root Traquest the callbacks bounded to the handlers are not called. The root Traquest calls any handlers only at the end of its life-cycle. Therefore, when the root Traquest enters the Committed state, it calls the callback bounded to the Then handler, responding the final result of the whole transaction.

Undoing state: When a Traquest enters the Undoing state, that means that all the modifications in the global state done by the current Traquest or its descendants should be rolled back. Traquests can enter the Undoing state from almost any state. The exceptional states are the Waiting, Committing, Committed, Ignored, and Terminated states. From Waiting, there would be no point in entering the Undoing state since without executing the Deferred process, there cannot be any modifications to be rolled back.

The Committing state is already part of the committing process. Here all the temporary values are finalized. Suppose any errors are happening at this phase. That means a more serious issue that can affect the consistency. Therefore rolling back cannot be an option from the Committing state.

The Committed, Ignored, and Terminated states are the final states of the Traqest, and they cannot be rolled back. In all the other states, except the five states mentioned above, stepping to the Undoing state is possible.

Stepping to the Undoing state can be triggered manually in the Executing state, during the execution of the Deferred process – we can use the mistake operation for this purpose – or it can be triggered automatically by the bounded Traquests. The bounded Traquests can use the undo operation for initiating a rollback and trigger the current Traquest to step into the Undoing state. However, when the current Traquest is still in Executing state, and an undo operation comes from its child, the Traquest can still be re-executed, and in this case, the Traquest will step into the Restarting state.

Traquests can spread up the undoing chain even if they already have responded. This is important because the undo operation means that the responded value is not valid anymore; therefore, the parent Traquest, which already consumed this value, should be conflicted.

Ignored state: The Traquest steps into the Ignored state from the Undoing state when it has finished calling all the undo operations on its bounded Traquests. The Ignored state is a final state, and it means the life-cycle of the Traquest has ended.

Restarting state: When the current Traquest gets an undo operation from its child Traquest while the current Traquest is still in the Executing state, the Deferred process of the current Traquest should be re-executed. In the Restarting state, an undo operation is called on each of the child Traquests, the rollback callback is called for the current Traquest – if it was defined – to roll back every state change that has been made so far. When the rollbacks are finished, a reExecute operation is called on the current Traquest, which sets back the state of the current Traquest to Executing, and re-executes the Deferred process.

Terminated state: The Terminated state was not discussed in depth so far, and it is missing from Figure 5, because it implies high complexity. Figure 10 in the Appendix shows all the possible states and state transitions a Traquest can have, including the Terminated state. The Terminate state can be initiated by calling the terminate operation of the Traquest. The Terminated state means that an unsolvable error has happened, and the system's consistency cannot be guaranteed.

When a Traquest enters the Terminated state, it tries to roll back itself and calls the terminate operation on all its bounded Traquests.

2.3 Timestamps

Traquests contain a logical timestamp of their creation to be able to resolve conflicts. The timestamp of a Traquest inherits all the timestamps of the ascendant Traqests. This means that the timestamp of a parent Traquest represents the logical time of its whole branch when compared with Traquests from other branches. The timestamp of an ascendent Traquest is always earlier than the timestamps of descendant Traquests. The order of the sibling Traquests is decided in the order of their creation.

To clarify how the order of the Traquests should be considered, Figure 6 shows two Traquest trees, and on the horizontal axis, their physical time of creation is represented. In this case, the physical order of the Traquests is the following: T1; T2; T3; T4; T5; T6; T7; T8; T9. However, since Traquests trees represent atomic operations, and the branches of the trees represent atomic sub-operations, the logical order of the Traquests cannot be equivalent to the physical order. The logical order of the Traquests in this particular case is the following: T1; T3; T6; T8; T4; T7; T2; T5; T9.

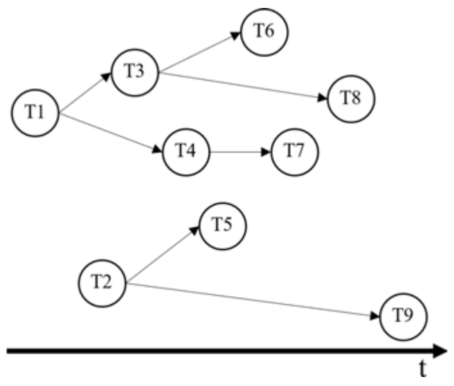


Figure 6: Order of Traquests

2.3.1 Current logical timestamps

There are already several solutions for creating timestamps and clocks in a distributed environment when more processes are running parallelly. The Lamport timestamp [27] algorithm or the Vector clocks [37] algorithm are designed to handle timing issues when more processes are existing at the same time which can interact with each other. In the case of the Lamport timestamp, creating the timestamp is very fast and easy but compare two timestamps is very hard and costs a lot of computation time. For the comparison, we also must decide if there is a joining path between the processes. It is not possible to use it for the Traquests. With the vector clocks, the result is similar. However, at the vector clocks, creating the timestamp is highly costly because we need as many dimensions for the clock as many processes we have. In our case, every Traquest is a process, and there can be millions of them or even more. Furthermore, Traquests can be created in real-time, which means at the point where we should give a timestamp for a Traquest, we do not even know how many processes should we consider to create a Vector Clock based timestamp. Therefore, the Vector clock works neither for the Traquest model.

2.3.2 Hierarchical timestamp

We need a hierarchical time stamping mechanism which is a more special case. The naive algorithm for creating a required hierarchical timestamp would be simply using an array with integers. All the Traquest can count how many children they have already, and whenever a child is created, they increase the counter; therefore, each child knows where they are in the queue of the order. The root Traquests can get their number from a global counter, from the Unix time, or a combination of the two. The array used as a timestamp can store all the ancestor's order numbers and also its own order number in the last record. Figure 7 shows the naive algorithm timestamps for the Traquest tree example presented in Figure 6.

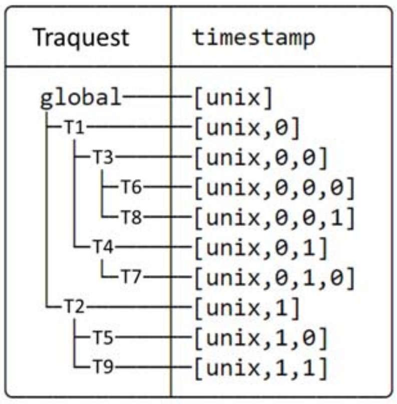


Figure 7: Naive hierarchical timestamps

In the example, we supposed that all the Traquests were created very close in physical time; therefore, they all have the same Unix time in the first record. After the global timestamp, all the timestamps contain their ancestors' timestamps; they are just extending it with their own order number. For flat hierarchies, it works well because there are only a few levels, and the length of the timestamp is short. However, the timestamps can get very long when the hierarchy is tall.

To address this issue, we have created a new timestamp algorithm that can reduce the necessary size of the timestamp drastically. Explaining our research on the optimized timestamps in more detail would be complex, and it is out of the scope of the current paper. Therefore, we rely on the naive timestamps for the explanation of the Traquest model.

2.4 Data protectors

We introduced that Traquests are forming a tree structure. However, a tree structure by itself would never result in any conflicts, which would be the core of the Traquest model to handle them effectively. Conflicts are happening when two different processes are trying to read or write the same part of the global state. To this end, Data protectors were constructed. Data protectors are entities responsible for managing a given segment of the global state. They protect the given global state particle from conflicting reads and writes.

The goal of each branch of the Traquest tree is to interact somehow with the global state. Therefore, each branch, at some point, ends up in a Data protector. Traquests can call CRUD [33] operations on the Data Protectors. When a Traquest calls a CRUD operation to a Data protector, the Data protector generates a new Traquest containing the operation and replies with the generated Traquest. This new Traquest can be bounded to the original Traquest as a child; therefore, it becomes part of the Traquest tree.

When more Traquests are using the same Data protector, the Data protector can use the logical timestamps of the Traquests to decide which read or write operation should be answered first. If a Traquest with an earlier timestamp comes after a Traquest with a later timestamp has been already responded to, the Data protector can call the undo mechanism of the already responded Traquest and serve the newly requesting Traquest. Therefore, the Data protector can efficiently resolve any conflicts.

Furthermore, because all the conflicts are recognized and resolved at the Data protectors, most of the conflict resolving features of the Traquests are used only by the Data protectors themselves. Data Protectors can trigger a Mistake if they have conflicting Traquests, and they can define the callback for the Rollback. This way, the increased complexity of the Traquests can be mostly hidden from the developers, and they do not need to care about the failure handling parts at all, except taking care of the Finally handler of the root Traquest. As a result, using Traquests can be as straightforward as using Promises or even more.

Data Protectors are the only entities who can contact directly to the global state storage. Therefore, the way we physically store the data can be abstracted away. Data Protectors can store the state using any databases, the local storage, the memory, or even any mixture of these solutions. Using traditional databases can be helpful to provide compatibility with other systems; however, in this case, we should be aware of the risk of corrupting the consistency of the global state. The most efficient solution is to use the local storage or memory for storing the state.

2.5 Consistency and Fault tolerance

Traquests interact with each other using serializable data constructs only. Therefore, Traquests can be located on different computing notes as well, and they still can interact. Fault tolerance requires the replication of the different particles of the global state to several computing nodes. Traquests are perfect for creating replicas of a desired global state particle and consistently managing them. It is enough to add new Traquest tree branches to each write operation that replicates the operation on different computing nodes. Thanks to the atomic property of the Traquest tree, the state will always remain consistent. For the read operations, we do not need such replication since the writes are already ensuring consistency.

2.6 Tail Traquests and network buffering

Tail Traquest is not a new separate feature in the Traquest model; it is instead a useful design pattern. Tail Traquests are simply Traquests where the Then handler of the Traquest is not defined. This implies that the parent Traquest of a Tail Traquest can finish its Deferred process without waiting for the current Traquest to finish with its task. This also means that the resulting value of a Traquest is independent of its child Tail Traquests.

Tail Traquests are primarily helpful for synchronizing the modifications in the global state made by the Traquest tree with other servers in a consistent way for reaching fault tolerance. Because Tail Traquests are not blocking the execution of the core logic of the Traquest tree, the overall Traquest tree can run very fast. Suppose every part of the global state that needs to be used is replicated locally. In that case, the whole logic of the Traquest tree can even execute in-memory time, and the network messages for the synchronizations are buffered automatically. They can be synchronized lazily in only two round trip messages for finalizing and committing. This optimization can happen even in the case of very complex algorithms, when there are many global state reads and writes depending on each other.

The communication of the Traquests between different computing nodes and the buffering can be separated and managed automatically. Therefore, the protocol for communication is abstracted away. The background implementation can use TCP [17], UDP [39], REST [21], WebSocket [8], WebRTC [18], long polling [19], SSE [36] or any other technologies what the infrastructure allows. This leaves many possibilities for optimizations. For example, if two nodes are communicating less frequently, they can use REST calls. However, if there are two nodes with frequent communication between them, they can switch to WebSocket. This way, the Traquest model attempts to create a new layer on top of the OSI layers [14], where the network communication itself can be abstracted.

2.7 A basic exemplary case

We have discussed how the basics of the Traquest model are working. To have a deeper understanding, let us examine how we can increment a simple integer value in the global state.

2.7.1 Basic scenario with no conflicts

To examine a basic scenario with no conflicts, let us discuss a simple incrementation of a value in the global state. In the Appendix Figure 11 illustrates such an example of an incrementation. The incremented global state variable is named i. The global process creates a Traquest for the transactional incrementation. A ”T” prefix marks the Traquests, and their postfix is their logical timestamp. ”DP_i” is the Data Protector of the i variable, and ”Storage_i” is the physical location of the i variable. The ”Storage_i” can be a local storage, it can be stored directly in the memory, or it can represent any kind of database as well. The sequence diagram notes are marking the actual states of the Traquests. The global process creates the ”T1” Traquest, and the other two Traquests are generated by the Data Protector, one for reading the i variable and one for updating it. The diagram shows the operations between the entities. The initial value of the i variable is 10.

2.7.2 Conflict resolving

To examine how Traquests behave in a conflicting scenario, let us continue with a similar incrementation, but in this case, we have two conflicting global processes (e.g., two threads). In the Appendix Figures 12a and 12b illustrate such an example of a conflicting incrementation. The first global process begins an incrementation on variable i, and the second process begins a read on the same i variable only a little later. In this case, we have two Traquest trees with T1 and T2 Traquests at the root. The T1 tree performs a read on variable i, next the T2 tree performs a read, and after that, T1 performs the write with the incremented value. This is a conflicting scenario because the T2 tree should read the value of i only after the T1 has completely finished with the incrementation.

Figures 12a and 12b in the Appendix show that despite the conflict, the global processes get only the correct result at the end, and the correct final global state value is persisted on the storage. Examining more the "DP_i" Data Protector, it is also visible that the Data Protector reads and writes to the storage only once. It is interesting if we consider that it had to serve several CRUD operations coming from the Traquests. Data Protectors can effectively aggregate those CRUD operations and reduce the number of CRUD operations necessary to call on the storage itself. This has higher importance if we consider that the storage is a component that can be located on different computing nodes; therefore, calling an operation on the storage can be the slowest element of the overall process.

3 Related work

3.1 Current technologies

There are many technologies for providing ACID concurrent systems. Hereby we discuss the most relevant and most widely used directions.

3.1.1 Multitier architectures

The "Layers" architectural pattern has been described in various publications [6], and it is the most widely used pattern in the case of enterprise web applications. When the Business layer executes the desired algorithm, it continuously has to access the Data access layer to read the global state and write back the changed state. When the algorithm requires only a few iterations depending on each other with the Data access layer, this causes no problem. On the other hand, each read and write requires a roundtrip on the network when there are several depending steps. Although many databases – e.g., most of the SQL databases – can easily handle atomic transactions, the number of the necessary roundtrips implicates a massive limitation in the overall performance. This is a strict limitation in any architecture where we separate the location where we execute the business logic from the location where we store the global state.

To give an example, one might select any use cases where there are several dependent reads or writes to any databases to compare the multitier architectures. One of those examples is the Geographical Information System using large point clouds. For this purpose traditional PostgreSQL [23] is often used. Using space partitioning algorithms is a necessity to be able to manage the point cloud. Octree [22] is such an algorithm that can let us manage the points efficiently and easily.

Figure 8 shows an example when a new point is added to an Octree. We assumed that adding the new point requires searching down the Octree structure for ten levels. We assumed that the whole data set is replicated to two servers.

Database servers cannot execute algorithms in multitier architectures. They are only responsible for storing the data. The application is executed by the application server. Therefore, each node has to be first read to the application server from one of the database servers. Each node refers to its children; therefore, all the parents should be read before the child can be reached, and pipelining [29] cannot be used. Furthermore, to ensure atomicity and consistency, each node must be checked and locked on both servers. The example showed in Figure 8 is a simplified one. In real life, there can be much more and complex network messages between the servers. Still, even in this simplified case, we can count up to 44 network messages between the servers.

This example shows that any solutions built on using databases can have strict performance limitations if the operations sent to the database are depending on each other. If they are independent, buffering and pipelining can be used, and many queries can be sent within a single RTT (Round Trip Time) over the network. In this case, the number of network messages can be $O(1)$.

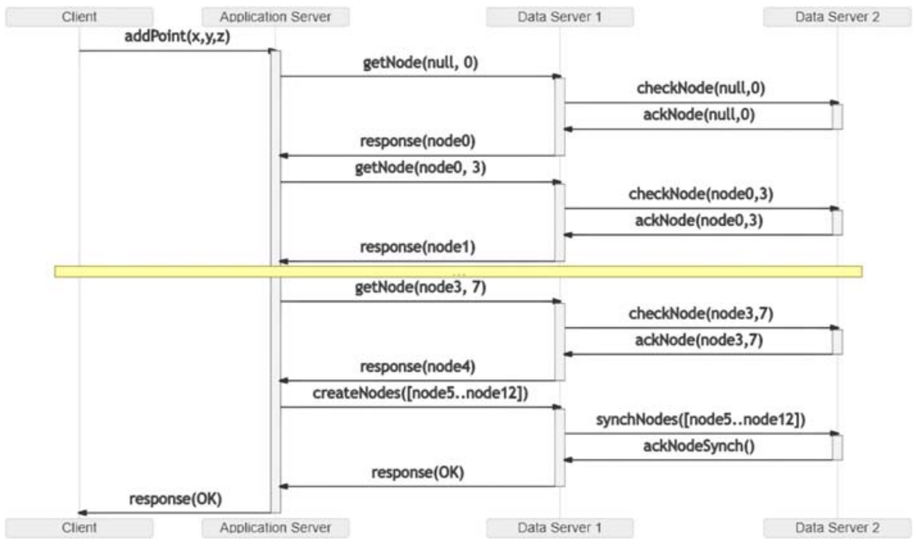


Figure 8: Add point to an octree in a multitier architecture

However, in many cases, we need to know the result of a query to be able to send the following query to the database. In such cases, the number of network messages grows on a $O(n)$ scale.

3.1.2 Serverless architecture

Serverless is one of the most trending architecture types nowadays. Serverless computing has emerged as a new compelling paradigm for the deployment of applications and services. It represents an evolution of cloud programming models, abstractions, and platforms and is a testament to the maturity and wide adoption of cloud technologies [4].

The serverless architecture is built on using stateless cloud functions in a managed way. The developer does not need to manage any server-side infrastructures. The number of cloud functions can scale horizontally automatically. These cloud functions can call other managed database systems to reach out to the application's global state. Serverless is getting more and more traction, and all the leading cloud providers are offering their serverless solutions: AWS Lambda [2], Google Cloud Functions [11], or Azure Functions [24].

However, in serverless architectures, the cloud functions have strict performance limitations thanks to the stateless nature of the cloud functions. The cloud functions cannot store any part of the global state. They need to call a database, a distributed file system, or other external services for that purpose. Therefore the cloud functions cannot be executed in-memory time, and they need to communicate on the network to finish their task. This means that from a performance perspective, they share the same limitations with the multitier architectures.

3.1.3 Actor model

Carl Hewitt first described the actor model in 1973 [15]. Actors can execute business logic and store state simultaneously; therefore, they do not share the limitations of the classical multitier architectures. Actors are excellent for solving problems where we have many independent processes that can work in isolation and only interact with other Actors through message passing. This model fits many problems. However, unfortunately, the actor model is not a favorable model for implementing truly shared state [34], when we need to have a consensus and a stable view of state across many components. Actors can get information about each other only through messages; therefore, it is hard to maintain atomicity.

The Actor model is a more general concept. Many algorithms can be implemented using Actors. Therefore we cannot directly compare the Actor model itself with the Traquest model because it depends on the algorithm that we create using the Actor model. However, there are standardized solutions for creating atomic transactions using the Actor model. The Akka toolkit suggests Transactors [34] for creating transactions. Under the hood, it uses a CommitBarrier, similar to a Java CountDownLatch [26] which is a blocking mechanism. Therefore, Transactors also have no specific timestamping mechanism. They have to lock and await each change in the global state; therefore, we would not have fewer network messages with the Transactors than what multitier architectures have.

3.1.4 Consistency protocols

The Traquest model can ensure atomicity and is also a promising way to ensure consistency. Therefore, hereby we take the most relevant consistency protocols [32] under investigation respective to the Traquest model.

Continuous consistency: Continuous consistency ensures that the numerical deviation of a specific global state particle does not go above a certain threshold on the different computing nodes. This can be applied only in the case of numerical values, and it ensures only an approximate consistency; therefore, it is out of scope for the Traquest model.

Primary-Based Protocols: Primary-Based protocols provide proper consistency for an arbitrary type of data. To keep the data consistent, they have to synchronize each write at least with the primary server. For instance, in such a case, the algorithm has to be blocked until a read it depends on gets a confirmation from the primary server. This requires many iterations of roundtrip messages; therefore, Primary-Based Protocol implies a strict limitation in the performance.

Quorum-Based Protocols: Quorum-Based Protocols have very similar limitations to Primary-Based Protocols. Each read and write operation must be confirmed by other computing nodes before the executed algorithm can rely on the operation and step forward. The only exception is the Read-One, Write-All scheme. In this case, it is enough to read the local state of the data; however, it requires even more messages to write synchronizations. This scenario can only be suitable in the case of very read-heavy applications.

The mentioned Primary-Based and Quorum-Based protocols share the same problems with the Transactors and multitier solutions. Each read or write should be crosschecked with other servers before we can rely on the locally stored data, and the locks are blocking the process. Therefore, these protocols cannot reduce the necessary network messages either.

3.2 Traquest model compared to the current technologies

In the following, we will discuss the exemplary GIS-octree case described in Section 3.1.1 to understand more and compare the Traquests.

With the Traquest model, we do not need a persistence layer because the Traquests themselves can already ensure the ACID properties. The data protectors can store the global state on the local storage or even in the memory. Therefore, there is no need for network communication for the reads in the case of an optimal topology. Moreover, write operations do not need network communication either, only for synchronization, which can be buffered and postponed. This way, all the network events can be done in only three RTTs. This results in $O(1)$ – or even less depending on the proportion of reads and writes – number of network messages even for operations that depend on each other.

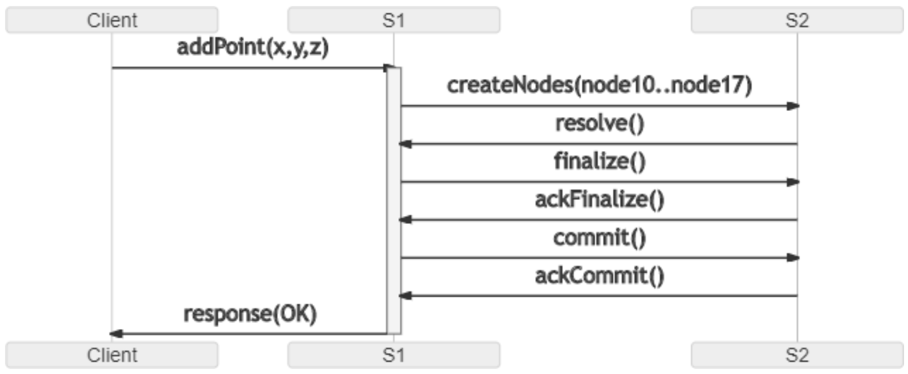


Figure 9: Add point to an octree in a Traquest based architecture

Figure 9 shows the same use case that was presented in Figure 8, but it uses the Traquest model. We do not need to use separate application and data servers since the Traquest model has constructions for both processing and storing. We have two servers for storing the replicated data on two different physical computing devices, just like the multitier example. As we discussed earlier, Traquests do not need to synchronize the read operations because if a conflict arises, the responded value can be later undone. Furthermore, the writes can also be aggregated and buffered and be sent in one message on the network. In this particular case, there are only six network messages between the servers. Compared with the optimistic estimation of 44 for the multitier architectures, this is a significant reduction. This difference is even bigger if there are more dependent iterative operations necessary for the database. For example, suppose we would have a graph as an example, and we wanted to find the shortest distance between two points. In that case, there could be thousands of dependent operations to the database, but with the Traquest model, we would still need only six network messages.

Here we need to emphasize that this optimal scenario is valid only if the necessary particles of the global state have replications locally. These numbers also depend on the actual infrastructure topology, the used components, the concrete use case, and many other factors.

However, even if not every data is available locally, the execution of the necessary Traquest tree branches can be delegated to other servers since all the servers are running the Traquest environment. Therefore the location of the processing can move to the data location and not backwards. This means much less communication over the network, even in this case.

Suppose we have a worst-case scenario, where the data stored over the servers is randomly fragmented. In that case, the number of the messages in the Traquest model can grow on a $O(n)$ scale, which is the best-case scenario for the multitier architectures. However, this would mean already an unrealistic and completely randomized worst-case topology. In general, we can say that Traquests has the potential to reduce the network load by magnitudes.

Table 1: Comparing the Traquest model

Case #	Circumstances				Response type		
	Concurrency	**Dependency**	**Topology**	**Operations**	**Final**	**Temporary**	
1	Optimistic	Independent	Ideal	Read only	$O(1)$	$O(0)$	
2	Optimistic	Independent	Ideal	Read & Write	$O(1)$	$O(1)$	
3	Optimistic	Independent	Random	Read only	$O(n)$	$O(n)$	
4	Optimistic	Independent	Random	Read & Write	$O(n)$	$O(n)$	
5	Optimistic	Dependent	Ideal	Read only	$O(n)$	$O(0)$	!
6	Optimistic	Dependent	Ideal	Read & Write	$O(n)$	$O(1)$	
7	Optimistic	Dependent	Random	Read only	$O(n)$	$O(n)$	
8	Optimistic	Dependent	Random	Read & Write	$O(n)$	$O(n)$	
9	Pessimistic	Independent	Ideal	Read only	$O(n)$	$O(0)$	!
10	Pessimistic	Independent	Ideal	Read & Write	$O(n)$	$O(n)$	
11	Pessimistic	Independent	Random	Read only	$O(n)$	$O(n)$	
12	Pessimistic	Independent	Random	Read & Write	$O(n)$	$O(n^2)$	
13	Pessimistic	Dependent	Ideal	Read only	$O(n)$	$O(0)$	!
14	Pessimistic	Dependent	Ideal	Read & Write	$O(n)$	$O(n)$	
15	Pessimistic	Dependent	Random	Read only	$O(n)$	$O(n)$	
16	Pessimistic	Dependent	Random	Read & Write	$O(n)$	$O(n^2)$	

Comparing the Traquest model with the existing technologies and models is challenging because there can be many different distributed scenarios. Table 1 shows a more complex comparison of the Traquest model. We showed four different kinds of circumstances that can highly influence the properties of a distributed system. The "Concurrency" column shows whether the examined distributed system has an optimistic or pessimistic concurrency scenario. We consider a scenario optimistic when there are no conflicts or the global state operations are executed in proper order or affect different records of the global state. In a pessimistic scenario, all the global state operations are executed in reverse order on the same global state records; therefore, we have the highest possible amount of conflicts. Realistically, most the distributed systems are much closer to the optimistic scenario.

The "Dependency" column shows whether the global state operations are dependent on each other. They are dependent if a global state operation has to await a previous one to be executed. For instance, incrementing a value is a dependent operation since the value first has to be read to increment it.

The "Topology" column shows whether the topology of global state particles is ideal or not. The topology is ideal if all the corresponding global state particles are stored on the same servers; therefore, the processes can reach them at one step. The topology is random if the global state particles are spread across the servers without considering the probability of them being used together. For instance, consistent hashing used for sharding by many of the most popular databases (AWS DynamoDB [7], Redis [30], etc.) creates such a random topology.

The "Operations" column shows whether the examined processes are write-heavy or they only contain read operations.

There are two "Response type" columns in the table. The Traquest model is built on introducing the temporary responses. As we described earlier, the Traquest model can collect all the conflicts and resolves them lazily in a buffered way. The column "Final" refers basically to all the currently existing technologies (multitier, serverless, actor, etc.) where the response to a request is final; therefore, all the conflict resolution must be finished in advance. It is usually done by using a locking mechanism. The "Temporary" column refers practically to the Traquest model as it introduces the temporary responses. However, we do not exclude the possibility that later new models or concepts will arise and utilize this idea as well.

The limitation in the performance of a distributed system is usually coming from the overall time needed to get network messages from one computing node to the other. A larger amount of data can be sent over the network relatively fast in a buffered way, but when there are more messages, each message has a roundtrip time and an overhead. Also, in-memory calculations are magnitudes faster than network communication. Therefore, to compare the Traquest model, we choose the necessary number of messages compared to the number of global state operations as the primary indicator for the performance of the distributed system.

The rows in the table describe different cases respective to the different circumstances. The last two columns show how the necessary number of network messages is growing as we increase the global state operations. The cells with green background mean easier circumstances or better performance, and the red background table cells have the opposite meaning. Table cells with white background mean no significant difference between traditional architectures and the Traquest model. The exclamation marks at the end of Case 5, 9, and 13 highlight that the Traquest model performs better not only by one but also by two magnitudes.

When the global state operations come strictly in order, they are independent, and the topology is ideal, the traditional databases can use pipelining. In this ideal scenario, the read and write operations can be sent over one single network message. This implies that at Case 1 and 2, the number of messages can scale on a $O(1)$ level ideally. In all the other cases, there is at least one network message per global state operation necessary. When the operations come entirely out of order and have a pessimistic concurrency scenario, traditional architectures using final responses still need only $O(n)$ messages. However, they also need to use locking mechanisms actively. This slows down the execution significantly. We did not show this effect in the table because we assumed that the servers could utilize this freed resource for other tasks or processes.

With the Traquest model, the processing and storing of the data can be performed on the same server, supposing we have read operations only and the topology is ideal. Therefore, in Case 1, 5, 9, and 13, there is no need for messages on the network at all. This means the number of the messages scales on a $O(0)$ scale. In an ideal case, when there are writes, the Traquest model can immediately process the writes in-memory time and postpones the conflict resolving and data replication lazily to buffer them to one single message. The committing mechanism runs in a buffered way as well, meaning that Case 2 and 6 can scale on a $O(1)$ level. In the worst cases, the Traquest model needs $O(n)$ network messages, except when we need to consider pessimistic concurrency. When we have pessimistic concurrency, random topology, and write-heavy global state operations, the Traquest model can require $O(n^2)$ number of messages. There is a big difference between read and write operations because reads do not generate conflicts thanks to the ability of the Data protectors to store the history of the global state records and serve the read operations, respectively.

We can conclude that the Traquest model performs better than the traditional architectures when considering some level of topology optimization and a more optimistic concurrency scenario. In this case, the advantage can reach several magnitudes. The Traquest model is not ideal when there is a limited amount of global state records getting many concurrent, and conflicting writes or the topology of the global state records is random.

3.3 Comprehensive solutions in the literature

3.3.1 Consistency Choices in Distributed Systems

Gotsman et al. [12] give a comprehensive overview of the consistency issues and the compromises that should be considered when designing a distributed system. It argues that only the necessary level of consistency should be used to avoid unnecessary loss in the performance. Using different levels of consistency in the same distributed system in such a mixed way is called hybrid consistency. The authors have designed a modular methodology to help developers decide the necessary level of consistency, and they presented proof for the methodology.

Despite the comprehensiveness of the paper, the different consistency issues are all discussed on the database level. All the consistency issues are discussed as database-related issues. This pattern can be recognized in the literature in general since consistency problems are associated with the consistency of the global state, and most of the distributed systems are managing the global state in the persistence layer. On the other hand, Traquests can ensure consistency – and atomicity – on the data processing level, not only on the data storage level. This is a major difference that allows a significant leap forward in the potentially reachable performance.

The authors of the paper also discuss the usefulness of hybrid consistency strategies. Through the enhanced performance, the Traquest model can highly reduce the consistency level dilemma and reduce the necessity of using hybrid strategies. Nevertheless, the Traquest model still supports hybrid consistency implicitly. When we wish to create strong consistency, we can build a fully bound Traquest tree as an atomic operation, and all the state changes and replica synchronization steps will remain strongly consistent.

However, we are not always restricted to define the parent of a Traquest. This way, we can separate different branches from the Traquest tree, and we can create less consistent operations to gain more performance. Although, in the Traquest model, it can rarely have clear benefits because strong consistency can already perform equally fast. Separating Traquest trees can be beneficial when we need to face highly conflicting use cases. In such a pessimistic concurrency case, the continuous rollbacking of the branches could slow down the execution of the Traquest tree, and hybrid consistency can be an effective solution.

3.3.2 Performance of Transactional Distributed Systems

Eric Brewer introduced the idea that there is a fundamental tradeoff between consistency, availability, and network partition tolerance. This tradeoff, known as the CAP [10] theorem, has been widely discussed ever since. Some of the interest in CAP derives from the fact that it illustrates a general tradeoff in distributed computing: the impossibility of guaranteeing both safety and liveness in an unreliable distributed system.

Ahsan et al. [1] discuss the CAP theorem in more depth. The authors highlight some of the limitations in the practical usage of the CAP theorem, and they propose a new impossibility theorem called the CAT theorem.

The paper refers to Jim Gray's paper from 1996 [13] which showed that the rate of transaction aborts increases at least proportional to the square of the TPS (throughput) of the system, and the third to the fifth power of the number of actions in the transaction.

The Traquest model cannot violate the CAP theorem either, but it can provide a workaround. The Traquest model can inherently provide consistency and partition tolerance. Transaction availability is also basically provided thanks to the Tail Traquests. Availability is provided for the whole transaction as well; just the final commit of the root Traquest has to be awaited. We will have the correct result, and only the response time can be higher for the final commit if the writes need to be synchronized.

The workaround nature of the Traquest model for the CAP theorem confirms that alternative and more practical impossibility theorems can be essential. The CAT theorem stands for Contention, Abort Rate, and Throughput. The Traquest model tries to keep the abort rate minimal, thanks to introducing the temporary mistakes. If an exception arises, that would typically trigger the abortion of the whole transaction. Instead, in the Traquest model, only the affected Traquest tree branch gets aborted, rolled back, and re-executed. Therefore the Traquest model balances between Contention and Throughput.

In the Traquest model, resolving a conflict is relatively expensive, thanks to the undoing mechanism. Therefore, the Traquest model shows the most significant potential in the case of optimistic concurrency cases. This refers to cases where we can assume that most of the operations are not conflicting. We can say that regarding the CAT theorem, the Traquest model prefers Contention and low Abort Rate over Throughput. However, this decision is not architecturally predefined. When the system gets fewer conflicting concurrent operations, it automatically achieves higher throughput.

4 Self-reflection and further research

4.1 Current status

To be able to verify the Traquest model, we have built an experimental prototype in TypeScript. The Traquest model itself has no language dependencies. It can be implemented practically in almost any programming language. However, using TypeScript gave us more benefits in the current phase of the research. The concept of Promises is highly used in JavaScript. Typescript is a superset of JavaScript [25]; therefore, it simplified the experimentation process with Promises and Traquests. The flexibility of JavaScript and the strong typing, the strictness, and the expressivity of TypeScript made it an ideal solution. Furthermore, TypeScript compiles to JavaScript, which can be executed on the client and server side, enabling the experimentation with hybrid architectures balancing between cloud, edge, and peer to peer.

Building this experimental prototype helped clarify that the general concept of the Traquest model is feasible and viable. We could try the different state transitions, callbacks and write unit tests and integration tests to verify whether the Traquests behave as expected. The functional tests have worked as they were expected, and they gave a positive result. However, the non-functional tests gave a slower execution time than we were expecting. The complete correctness of the final version of the model can be verified only after more tests at a larger scale or a complete formalization of the model. This requires further research and optimization. However, we expect no significant changes in the general concept. Only slight modifications are expected for edge cases we could not consider in advance. These edge cases also might vary slightly depending on the programming language used for the implementation.

4.2 Local boilerplate

One of the main disadvantages of the Traquest model is that it increases the computing power necessary on a local level. Managing all the Traquests, calculating the timestamps, maintaining all the states can consume significant computing. The non-functional tests of the Traquest model clearly showed this issue. When we created only simple standalone Traquests, then 100,000 Traquests could be executed in 460 msec. However, when we had a more complex Traquest tree where we created a binary tree from the Traquests, we measured 990 msec to execute only 5,998 Traquests which means less than one Traquest/msec.

Thanks to the results of the non-functional tests, the current focus of our research is to optimize the Traquests performance.

Some parts of this increased computing resource consumption come from the programming language used for the prototype implementation. Namely, if we dynamically change the schema of an object in runtime, the V8 JavaScript engine changes to a much slower general implementation in the background. We measured that this deceleration can be even on a 100X times slower level. We have built many new concepts to mitigate this issue. To describe them in detail would be out of scope for the current paper, but we mention the most important ones. We have created a custom Promise implementation which we measured to be 30X times faster than the standard JavaScript Promise implementation. We also created a custom callback mechanism which we measured to be around 14X times faster than the standard JavaScript anonymous callbacks. We also prepared a second prototype of the Traquest model, avoiding object schema changes. With these optimizations, we could significantly accelerate the Traquests and execute 100,000 standalone Traquests in 54 msec.

Some parts of the increased computing resource are coming from a more general algorithmic level. We have also conducted more research that would be out of scope to discuss in more detail, but we mention the most significant achievements. One of the most resource-consuming parts of the Traquest model is the hierarchical timestamping mechanism. We highlighted this issue in Section 2.3.2. To address this, we created a new timestamp algorithm what we named Interval Timestamp, which uses a logical time interval to define inheriting timestamp relations instead of arrays used by the naive algorithm.

We plan to expand the research on aggregating the Traquests that can be executed sequentially. If a group of Traquests is executed by one single thread, that means there can be no conflicting operations from elsewhere; therefore, they could be handled as one aggregated Traquest. This optimization could reduce the execution boilerplate of the Traquests almost completely. Only the Traquest communicating on the network would break this aggregation and end up in physically new Traquests.

After this research phase, we will have a final Traquest implementation, and we will be able to test Traquests in larger quantities. This will let us create a more rigorous validation of the Traquest model and make more definitive performance tests comparing Traquests with architectures based on the currently existing technologies.

The Traquest model in its current status is already a unique approach utilizing the idea that a response to a request can contain temporary information. A general system can be built based on this principle. This system can postpone synchronization and conflict resolution phases lazily, enabling us much more buffering on the network and in-memory time speed even in the dependent operations where network communication is inevitable using the current technologies.

4.3 Formalizing

We tried to model every possible scenario that can happen with a Traquest tree to be sure about the correctness of the model. However, the Traquest model is a new concept, and in the future, building formal semantics and reasoning for the Traquest model using tools like the Coq [38] formal proof managing system would give us an absolute certainty about the correctness of the model.

5 Conclusion

Providing ACID properties can be crucial for many applications, but it requires a massive compromise in performance. The Traquest model is a proposed potential solution for this problem. We have discussed in detail the general concept of the Traquest model.

Traquests are unifying the location of the data storing and processing; they are using specialized timestamps and history tracking of the global state changes that can potentially cause conflicts. By creating temporary responses and building up a mechanism for rolling back the conflicting parts of a running distributed algorithm, Traquests can ensure atomicity in a very efficient way. The synchronization steps over the network for replication and fault tolerance do not block the business logic executed in the Traquests, giving a massive advantage in the performance. Furthermore, this lazy synchronization allows an effective buffering of the network messages; therefore, the number of the necessary network messages can be decreased by magnitudes.

In our investigated concrete use case, the classical multitier architecture required 44 messages between the servers, where the Traquest model only needed six. The difference can grow magnitudes as the complexity of the algorithm grows. We showed that classical architectures require at least $O(n)$ network messages as the number of operations grows in the case of depending operations. Simultaneously, the Traquest model can scale with only a $O(1)$ factor.

We showed how the Traquest model could mitigate the dilemma of choosing between the different consistency levels and how the Traquests can provide hybrid consistency.

We investigated the Traquest model through the CAP theorem and realized that the Traquest model does not violate but gives a workaround for the CAP theorem. We considered another impossibility theorem as well, called the CAT theorem, and identified the characteristics of the Traquest model respectively.

We discussed the main current challenges and future research directions to come over those limitations. The biggest challenge is to improve the local performance of the Traquests, and our suggested solution is the aggregation of those Traquests to reduce the boilerplate execution.

The presented Traquest model can be a good foundation for making distributed ACID computation magnitudes faster and easier.

References

[1] Ahsan, Shegufta Bakht and Gupta, Indranil. The cat theorem and performance of transactional distributed systems. In *Proceedings of the 4th Workshop on Distributed Cloud Computing*, DCC '16, New York, NY, USA, 2016. Association for Computing Machinery. DOI: `10.1145/2955193.2955205`.

[2] Amazon. AWS Lambda. `https://aws.amazon.com/lambda/`, 2021. Accessed: 2021-04-14.

[3] Baker, Henry and Hewitt, Carl. The incremental garbage collection of processes. In *SIGPLAN Notices 12, 8*, pages 55–59. ACM, 1977.

[4] Baldini, Ioana, Castro, Paul, Chang, Kerry, Cheng, Perry, Fink, Stephen, Ishakian, Vatche, Mitchell, Nick, Muthusamy, Vinod, Rabbah, Rodric, Slominski, Aleksander, and Suter, Philippe. *Serverless Computing: Current Trends and Open Problems.* In *Research Advances in Cloud Computing*, pages 1–20. Springer Singapore, Singapore, 2017. DOI: `10.1007/978-981-10-5026-8_1`.

[5] Bogner, Justus, Zimmermann, Alfred, and Wagner, Stefan. Analyzing the relevance of SOA patterns for microservice-based systems. In *10th Central European Workshop on Services and their Composition*, volume 2072, pages 9–16, 2018. `http://ceur-ws.org/Vol-2072/`.

[6] Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, and Stal, Michael. *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns.* Wiley, 1996.

[7] DeCandia, Giuseppe, Hastorun, Deniz, Jampani, Madan, Kakulapati, Gunavardhan, Lakshman, Avinash, Pilchin, Alex, Sivasubramanian, Swaminathan, Vosshall, Peter, and Vosshall, Werner. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007. DOI: `10.1145/1323293.1294281`.

[8] Fette, Ian and Melnikov, Alexey. The WebSocket Protocol. *RFC*, 6455:1–71, December 2011.

[9] Friedman, Daniel and Wise, David. The impact of applicative programming on multiprocessing. Technical report, Computer Science Department, Indiana University, Bloomington, 1976.

[10] Gilbert, S. and Lynch, N. Perspectives on the cap theorem. *Computer*, 45(02):30–36, 2012. DOI: `10.1109/MC.2011.389`.

[11] Google. Google Cloud Functions. `https://cloud.google.com/functions`, 2021. Accessed: 2021-04-14.

[12] Gotsman, Alexey, Yang, Hongseok, Ferreira, Carla, Najafzadeh, Mahsa, and Shapiro, Marc. 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems. *SIGPLAN Not.*, 51(1):371—384, 2016. DOI: `10.1145/2914770.2837625`.

[13] Gray, Jim, Helland, Pat, O'Neil, Patrick, and Shasha, Dennis. The dangers of replication and a solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 173–182. Association for Computing Machinery, 1996. DOI: `10.1145/233269.233330`.

[14] Henshall, John and Shaw, Sandy. *OSI explained: end-to-end computer communication standards.* Ellis Horwood Chichester, 1990.

[15] Hewitt, Carl, Bishop, Peter, and Steiger, Richard. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973.

[16] Hibbard, Peter. Parallel processing facilities. In Schuman, Stephen A., editor, *New Directions in Algorithmic Languages.* IRIA, 1976.

[17] Kozierok, Charles M. *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference.* William Pollock, 2005.

[18] Loreto, Salvatore and Romano, Simon Pietro. *Real-time communication with WebRTC: peer-to-peer in the browser.* O'Reilly Media, Inc., 2014.

[19] Loreto, Salvatore, Saint-Andre, P, Salsano, Sd, and Wilkins, G. Known issues and best practices for the use of long polling and streaming in bidirectional http. *Internet Engineering Task Force, Request for Comments*, 6202(2070-1721):32, 2011. DOI: `10.17487/RFC6202`.

[20] Madsen, Magnus, Lhoták, Ondřej, and Tip, Frank. A Model for Reasoning about JavaScript Promises. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. DOI: `10.1145/3133910`.

[21] Masse, Mark. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* O'Reilly Media, Inc., 2011.

[22] Meagher, Donald. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982. DOI: `10.1016/0146-664X(82)90104-6`.

[23] Meyer, T and Brunn, A. 3D Point Clouds in PostgreSQL/PostGIS for Applications in GIS and Geodesy. In *Proceedings of the 5th International Conference on GISAM - Volume 1*, pages 154–163. SciTePress, 2019. DOI: `10.5220/0007840901540163`.

[24] Microsoft. Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`, 2021. Accessed: 2021-04-14.

[25] Microsoft. TypeScript Documentation. `https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html`, 2021. Accessed: 2021-04-14.

[26] Oracle Corporation. Java documentation - Class CountDownLatch. `https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/CountDownLatch.html`, 2018. Accessed: 2021-04-14.

[27] Plakal, Manoj, Sorin, Daniel J., Condon, Anne E., and Hill, Mark D. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, page 67–76. ACM, 1998. DOI: `10.1145/277651.277672`.

[28] Ratai, Daniel Balazs, Horvath, Zoltan, Porkolab, Zoltan, and Toth, Melinda. Traquest model – a novel model for ACID concurrent computations. In *The 12th Conference of PhD Students in Computer Science – Proceedings*, pages 44–48. Institute of Informatics, University of Szeged, 2020.

[29] Redis Labs Ltd. Redis documentation – Using pipelining to speedup Redis queries. `https://redis.io/topics/pipelining`, 2020. Accessed: 2021-04-14.

[30] Redis Labs Ltd. Redis documentation – Partitioning: how to split data among multiple Redis instances. `https://redis.io/topics/partitioning`, 2021. Accessed: 2021-04-14.

[31] Sarieddine, Rami. *JavaScript Promises Essentials.* Packt Publishing Ltd., 2014.

[32] Tanenbaum, Andrew S. and Steen, Maarten Van. Consistency protocols. In *Distributed Systems – Principles and Paradigms*, pages 306–317. Prentice Hall, 2007.

[33] Truica, C., Radulescu, F., Boicea, A., and Bucur, I. Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. In *2015 20th International Conference on Control Systems and Computer Science*, pages 191–196, 2015. DOI: `10.1109/CSCS.2015.32`.

[34] Typesafe Inc. Transactors. `https://doc.akka.io/docs/akka/2.1/scala/transactors.html`, 2013. Accessed: 2021-04-14.

[35] Uyanık, H. and Ovatman, T. Enhancing Two Phase-Commit Protocol for Replicated State Machines. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 118–121, 2020. DOI: `10.1109/PDP50117.2020.00024`.

[36] Vinoski, S. Server-sent events with yaws. *IEEE Internet Computing*, 16(5):98–102, 2012. DOI: `10.1109/MIC.2012.117`.

[37] Zakeryfar, Maryam and Grogono, Peter. Static Analysis of Concurrent Programs by Adapted Vector Clock. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, page 58–66, New York, NY, USA, 2013. Association for Computing Machinery. DOI: `10.1145/2494444.2494476`.

[38] Zhang, Xiyue, Hong, Weijiang, Li, Yi, and Sun, Meng. Reasoning about connectors in Coq. In *International Workshop on Formal Aspects of Component Software*, pages 172–190. Springer, 2016.

[39] Zheng, Haitao and Boyce, Jill. An improved UDP protocol for video transmission over internet-to-wireless networks. *IEEE Transactions on Multimedia*, 3(3):356–365, 2001. DOI: `10.1109/6046.944478`.

Appendix

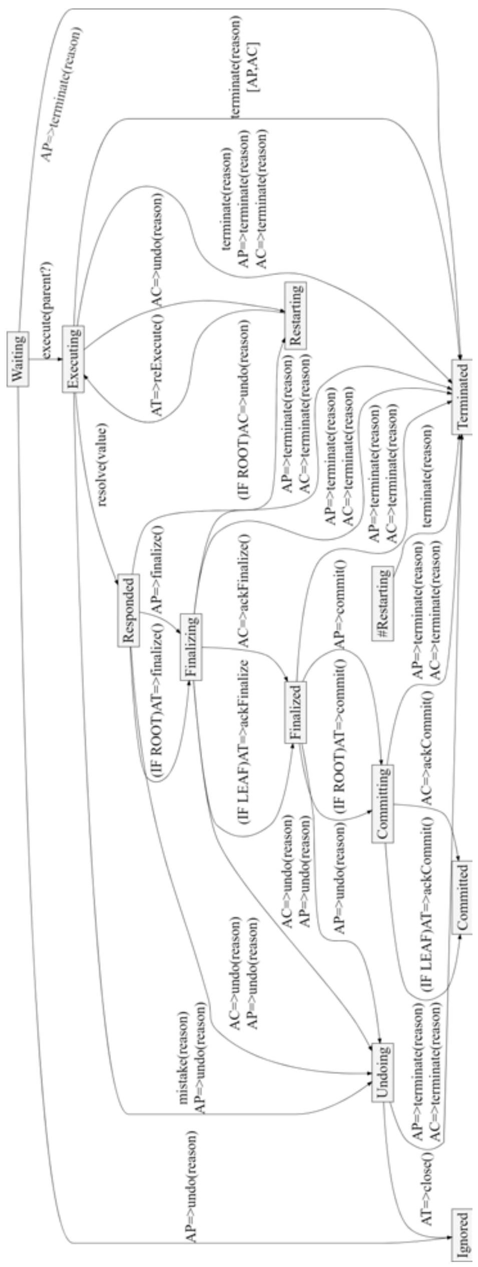


Figure 10: Comprehensive Traquest state diagram

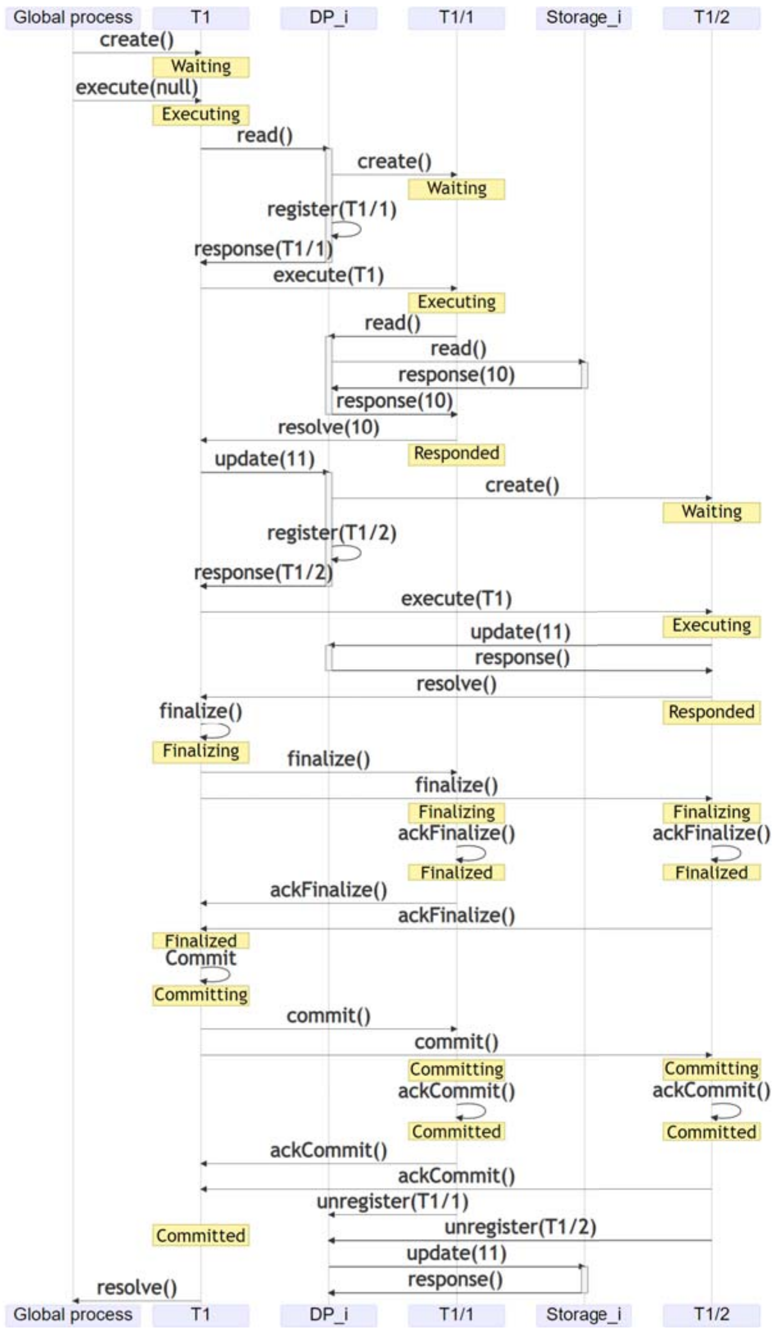
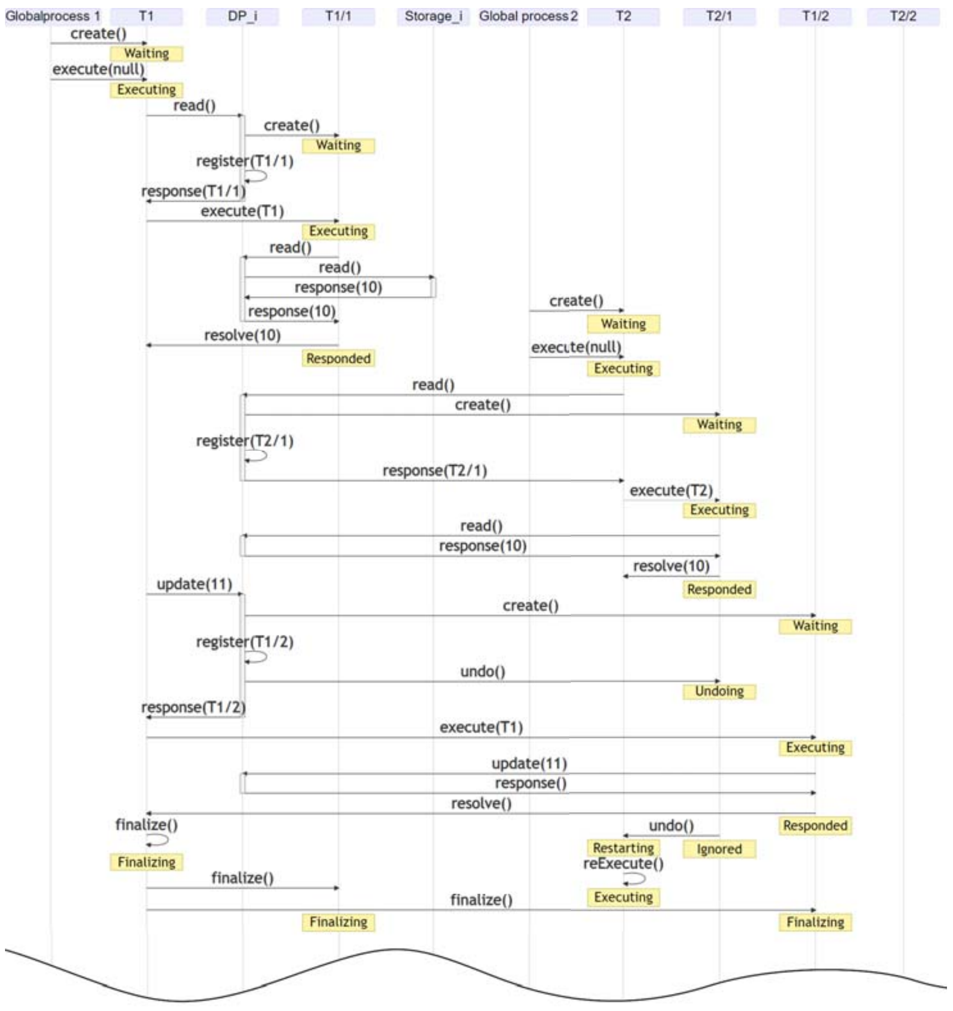


Figure 11: Sequence diagram of an incrementation



...

Figure 12a: Sequence diagram of an incrementation with a conflict (Part 1/2)

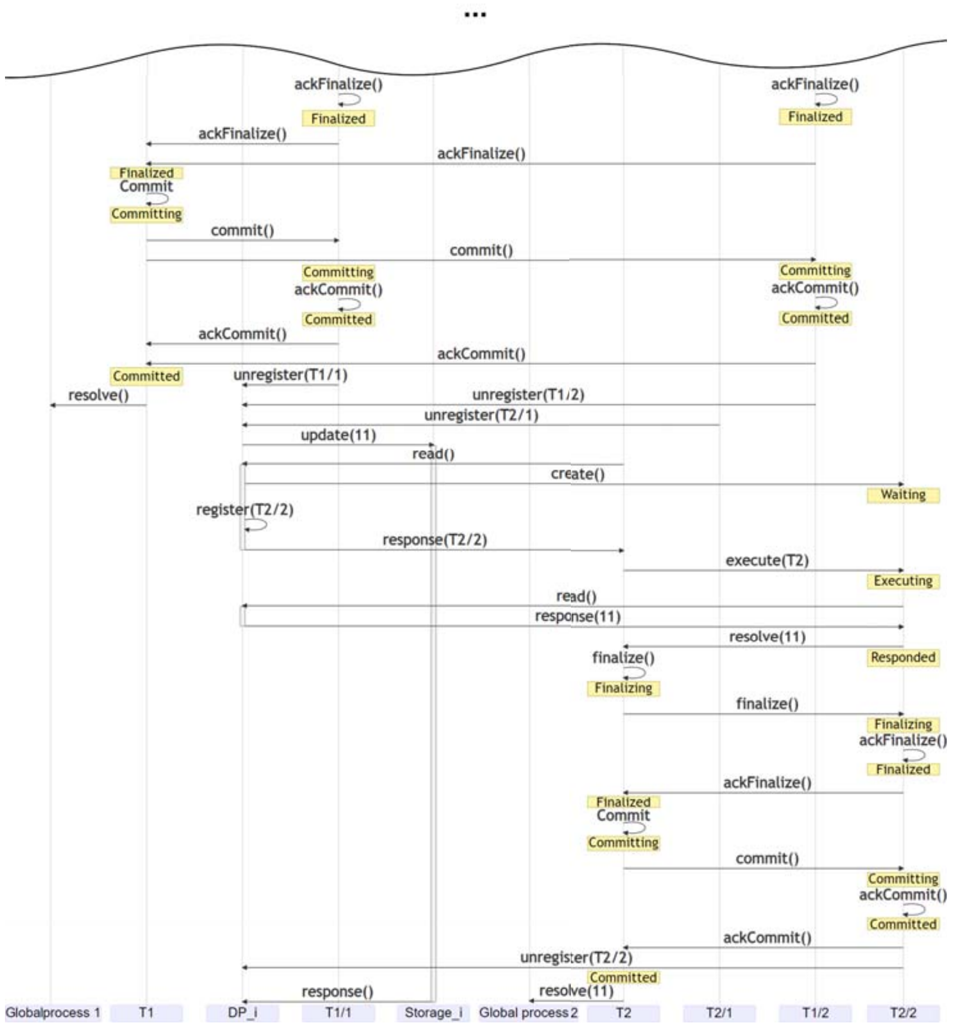


Figure 12b: Sequence diagram of an incrementation with a conflict (Part 2/2)