A CAPable distributed programming model

Myter, Florian; De Meuter, Wolfgang

Published in:
Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software

DOI:
https://doi.org/10.1145/3276954.3276957

Publication date:
2018

Document Version:
Final published version

A CAPable Distributed Programming Model

Florian Myter
Vrije Universiteit Brussel
Brussel, Belgium
fmyter@vub.be

Christophe Scholliers
Universiteit Gent
Gent, Belgium
Christophe.Scholliers@UGent.be

Wolfgang De Meuter
Vrije Universiteit Brussel
Brussel, Belgium
wdmeuter@vub.be

Abstract

Developers of modern distributed systems continuously face the impossibility result proved by the CAP theorem. In a nutshell, the theorem states that a partition-tolerant system can either guarantee consistency or availability.

Most distributed programming languages implicitly make the choice between consistency or availability in their designs and implementations. Concretely, distributed programming languages can be roughly divided into two categories. A first category of languages provide abstractions to implement the consistent parts of a distributed system. A second category of languages provide abstractions to implement the available parts of a distributed system.

However, real-world distributed systems often require consistency for some parts while requiring availability for others. Programmers are therefore forced to implement the abstractions missing from their chosen distributed programming language themselves or rely on external libraries.

In this paper we present a novel distributed programming model. This model introduces two object-oriented abstractions: consistents and availables. The former guarantees strong consistency by sacrificing availability. The latter guarantees availability, but only provides eventual consistency. Through these constructs programmers are able to implement the entirety of their distributed system within the same language.

We present a prototypical implementation of the model as a TypeScript library called CAPtain.js. To showcase the usefulness of our approach we implement a non-trivial example application. Moreover, we highlight both the functional as well as the performance characteristics of both language abstractions.

CCS Concepts • **Software and its engineering → Distributed programming languages**;

Keywords distributed programming, CAP theorem, data consistency

ACM Reference Format:
Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3276954.3276957

Onward! '18, November 7–8, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6031-9/18/11…$15.00
https://doi.org/10.1145/3276954.3276957

1 Introduction

Modern distributed web applications replicate their state across multiple servers and/or clients to provide features such as offline availability, performance, security, etc. Replication forces the developer to think about *Consistency*, *Availability* and *Partition tolerance* as captured by the CAP theorem [3, 9]. This theorem proves that it is impossible for a distributed system to simultaneously guarantee all three. Since web applications must be partition tolerant (clients disconnect frequently) the programmer is left with a trade-off guaranteeing either availability or consistency.

Most high-level distributed programming languages and libraries implicitly make this trade-off for the programmer. For example, E's [21] *eventual references* are used to implement consistent and partition-tolerant (*CP*) systems. On the other side of the spectrum, Lasp [18] exclusively relies on CRDTs [23] for distribution which makes it suitable to implement available and partition-tolerant (*AP*) systems. Unfortunately, many applications cannot be categorized as fully *AP* or *CP*. Programmers faced with such mixed *AP-CP* applications cannot rely on the high-level abstractions offered by a *single* distributed programming language. Instead they are forced to resort to low level APIs or external libraries to guarantee *either* their system's consistency or availability.

This paper builds forth on the idea presented by Christopher Meiklejohn in [19]: distributed programming languages should provide programmers the tools to explicitly specify the *AP* and *CP* parts of their system. To this end we introduce a novel object-oriented distributed programming model. At the core of this model lie two kinds of objects: *availables* and *consistents*. Our model ensures that invocations on *available* objects always return a value. Moreover, our model also ensures that all instances of an available object are kept eventually consistent across the application. Conversely, invocations on *consistent* objects are not guaranteed to return a value (e.g. in the case of network failures or partitions).

However, instances of a consistent object are guaranteed to be strongly consistent.

Concretely, this paper provides the following contributions:

- A novel distributed programming model for *AP-CP* applications.
- A prototypical implementation of this model in a TypeScript library called CAPtain.js.
- The implementation of collaborative grocery list application which showcases the functional characteristics of CAPtain.js.
- Micro-benchmarks which showcase the performance characteristics of CAPtain.js.

2 CAP and its (im)Practical Implications

The CAP theorem [3, 9] is a widely cited impossibility result in the field of distributed systems and distributed programming. Since the original publication of the CAP theorem a large number of alternative definitions and personal interpretations have emerged [13]. In this paper, we follow the definition of the CAP theorem as given by Seth Gilbert and Nancy Lynch [9]. We summarize this definition using a fully connected distributed system consisting of three nodes (i.e. *A*, *B* and *C*) which conceptually share a piece of readable and writeable memory.

Consistency Assume *B* writes a value *v* to the shared memory at some point in time. A distributed system guarantees consistency if all subsequent read operations by any node in the network return *v* until the memory is overwritten by another value. This property is also known as *linearizability* [11].

Availability A system ensures availability if a node is guaranteed to receive *meaningful* results for all operations performed on the shared memory (e.g. a request-timeout exception is not considered meaningful). In other words, a node is able to read from and write to the shared memory at any point in time.

Partition Tolerance A partition separates the network into disjoint sub-networks. Communication across sub-networks is impossible for the duration of the partition. In our example, a network partition could separate *A* from *B* and *C*. A system is partition tolerant if it is able to maintain its consistency or availability guarantees in the face of these partitions.

The CAP theorem proves that it is impossible for any distributed system to ensure that the operations on shared memory are consistent, available and partition tolerant. Distributed systems can only guarantee two out of these three properties. Because any real-world distributed system faces partitions at some point in time the programmer is left with the choice between offering availability or consistency. While the CAP theorem proofs that sharing data in a partition-tolerant distributed system is either available or consistent this choice can be made for each shared piece of data.

For example, consider a collaborative grocery list application which allows users to add grocery items to a list and mark items on the list as *bought*. Availability of the grocery list ensures that users can concurrently add items to the grocery list even if the user is temporarily disconnected from the application's server. Consistency of the *bought* status of an item ensures that said item can only be bought once. Unfortunately, current distributed programming languages only natively offer the ability to implement the former or the latter requirement of our example. This make it extremely tedious to develop such mixed AP-CP applications.

In this paper we introduce two novel object-oriented language constructs for distributed programming: *availables* and *consistents*. On one hand *availables* allow the programmer to implement the *AP* functionalities of their system. Our runtime ensures that availability is guaranteed even in the face of partitions. Moreover, *availables* are kept eventually consistent across nodes in the network. This unburdens the programmer from manually synchronising diverging state of *availables* after a partition heals. On the other hand *consistents* allow the programmer to implement the *CP* functionalities of their system. The runtime guarantees that operations on *consistents* are consistent even in the face of partitions. This comes at the price of these operations not always being available.

In this paper we mean *eventual consistency* to be the kind of eventual consistency guaranteed by, for example, the global sequence protocol [4]. This contrasts with *strong* eventual consistency, as provided by CRDTs [23], which we do not yet support.

3 CAPtain: a Novel AP-CP Programming Model

This section starts by introducing CAPtain: a novel programming model for AP-CP distributed systems. Furthermore, we provide an overview of CAPtain.js [1]: a prototypical implementation of our distributed programming model in Spiders.js [22] [2].

3.1 The CAPtain Programming Model

Our core model consists of two kinds of distributed objects:

Availables implement the *AP* functionality in a distributed application. Figure 1 (A) gives a conceptual overview of how they work. A node *x* has acquired a replica of an available (marked *A*). Conceptually, a single node in the network (i.e. node *y*) owns the *master* (marked *A'*) replica for all instances of an available. An object

[1] https://github.com/myter/CAPtain

[2] A Typescript library offering actor-based programming for web applications

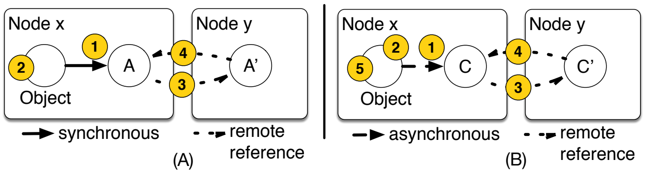

Figure 1. (A) Method invocation on an Available. (B) Method invocation on a Consistent

in x synchronously invokes a method of A (1), the method executes locally on A and returns a value (2). Subsequently, A sends its new local state to its master (3) which merges the state with its own and returns the new global state (4).

Consistents Implement the *CP* functionality in a distributed application. Figure 1 (B) gives a conceptual overview of how they work. A node x has acquired a replica of a consistent (marked *C*). As is the case for availables, a single node in the network (i.e. node y) owns the *master* (marked *C'*) replica for all instances of a consistent. An object in x asynchronously invokes one of *C*'s methods (1) which returns a promise (2). Subsequently, *C* sends the invocation (i.e. the method's name and arguments) to its master (3) which locally performs the invocation and sends back the return value (4). Finally *C* resolves the promise (5) returned in step 2 with the return value received from *C'* in step 4.

Availables and consistents differ in their underlying synchronisation mechanisms. The state of an available can always be changed or read, the synchronisation mechanism eventually ensures that this state change propagates to the other replicas. In contrast, the state of a consistent can only be changed or read if strong consistency amongst all replicas is guaranteed. It is important to note that CAPtain abstract away from the actual implementation of these synchronisation mechanisms.

3.2 Implementing a Counter in Captain.js

```
1  class AvailableCounter extends Available{
2   value
3
4   constructor(){
5    super()
6    this.value = 0
7   }
8
9   @mutating
10  inc(){
11   this.value++
12  }
13 }
```

Listing 1. Defining an available counter

```
class ConsistentCounter extends Consistent{
 value

 constructor(){
  super()
  this.value = 0
 }

 inc(){
  this.value++
 }
}
```

Listing 2. Defining a consistent counter

We introduce CAPtain.js (a prototypical implementation of the CAPtain model) using a counter example. Listings 1 and 2 contain the definitions for two kinds of counters: available and consistent counters. Instantiating a counter from either of these definitions returns a replica which can be replicated amongst clients. *AvailableCounter* replicas can be incremented, and their values can be read, regardless of a node's network connectivity. Consequently, the *value* of these replicas might temporarily diverge on multiple clients. CAPtain.js ensures the eventual consistency of all replicas of an *AvailableCounter* instance. In contrast, the *value* of *ConsistentCounter* replicas never diverges amongst clients. CAPtain.js ensures this by only allowing connected clients to access *ConsistentCounter* replicas.

The main difference between both definitions is the *@mutating* annotation for the *inc* method on line 9 in Listing 1. This annotation informs the CAPtain.js runtime that invocations of this method mutate the state of a replica. CAPtain.js synchronises the state of all instances of a replica as soon as it detects that *inc* was invoked on one of them.

```
setupPubSubServer(serverAddress, serverPort)
```

Listing 3. Initialising the counter server.

```
let conTopic = new Topic("Consistent")
let avTopic = new Topic("Available")
setupPubSubClient(serverAddress, serverPort)
sub(conTopic).each((replica)=>{
 replica.inc()
})
sub(avTopic).each((replica)=>{
 replica.onLocal(()=>{
  //Update UI
 })
 replica.onGlobal(()=>{
  //Update UI
 })
 replica.inc()
})
function newConCounter(){
 pub(new ConsistentCounter(),conTopic)
}
function newAvCounter(){
 pub(new AvailableCounter(),avTopic)
}
```

Listing 4. Publishing and subscribing to counters

Replicas are disseminated across clients through a topic-based publish-subscribe mechanism. Clients can publish a replica under a certain topic, after which other clients can obtain a copy of this replica by subscribing to said topic.

Listings 3 and 4 show how this mechanism is used for the counter application. The former initialises the application's server, which provides inter-client communication. We assume that *serverAddress* and *serverPort* are provided to the server on start-up. The latter defines the clients' behaviour.

Each client creates two topics, *conTopic* and *avTopic* (see line 1 and 2 in Listing 4), to respectively share consistent and available counter replicas. Replicas are created and published by the *newConCounter* and *newAvCounter* functions (see line 16 and 19), which are invoked by the clients' UI. Clients obtain replicas of these counters through the *sub* function. The function accepts a callback as argument which is invoked each time a counter replica is published under a given topic. In our example each client subscribes to the topics for consistent and available counters. Each subscription callback is therefore invoked with a different kind of counter. The first callback (see line 4) is invoked with *ConsistentCounter* replicas. Invoking *inc* on such a replica will only be performed if the client is connected to the network. CAPtain.js ensures that all reads following the increment return the updated counter value.

The second callback (see line 7) is invoked with *AvailableCounter* replicas. Invoking *inc* on such a replica immediately increments the value of the counter. CAPtain.js ensures that eventually all clients which obtained a replica of this counter witness this increment. Programmers are able to install two types of state-change listeners on availables. A replica's *onLocal* listeners are triggered whenever the node which holds a reference to the replica changes its state. For example, *onLocal* listeners are triggered regardless of a node's connectivity to the network. A replica's *onGlobal* listeners are triggered whenever its master sends the new global state.

3.3 Interactions Between Available and Consistent Objects

In order to guarantee their properties, available and consistent objects are completely separated from each other. To allow a limited form of interaction, CAPtain provides operators which convert available objects into consistent objects and the other way around. On one hand *freeze* accepts an available as argument and creates a new consistent which represents a snapshot of the available's state at freeze time. On the other hand, *thaw* accepts a consistent as argument and returns a new available which represents a snapshot of the consistent's state at thaw time.

Fields of an available cannot be assigned to a consistent value nor can a method of an available be invoked with a consistent as argument. If CAPtain would not enforce this restriction an available could have a field containing a consistent. This would jeopardise the availability of the object because reading the field's value could return a promise which might never resolve.

The restrictions which apply to consistents are similar to those on availables. Consistents can only hold references to other consistents or primitive values. Accessing a consistent's field or invoking one of its methods is strongly consistent across all nodes in the network. This property cannot be guaranteed if consistents can hold references to availables which only provide eventual consistency. CAPtain.js enforces these restrictions through run-time exceptions.

Finally, CAPtain also imposes a number of restrictions on the lexical scope of availables and consistents. Both availables and consistents only have (restricted) access to their lexical scopes at creation time. While this might look very restricting at first sight it closely resembles Scala's *spores* [20]. In a nutshell, spores allow programmers to create closures which can be safely distributed (e.g. by enforcing that spores and the variables they capture are serialisable). Both approaches rely on the programmer to specify which variables in the available's/consistent's or spore's lexical scope are to be captured. However, the spores approach is more substantial as it includes a type system which can enforce safety properties at compile time.

4 Ensuring Consistency in Captain

Availables and consistents differ in the consistency guarantees they provide. The former provide some form of eventual consistency (i.e. eventual or strong eventual consistency) while the latter provide some form of strong consistency. Our model abstracts away the mechanisms used to provide these consistency guarantees. Implementations of our model are free to choose how they provide these consistency guarantees. For example, availables in CAPtain.js use the global sequence protocol (GSP) [4] to provide eventual consistency. Consistents in CAPtain.js use *far references* [6] to provide sequential consistency.

4.1 Eventual Consistency through GSP

In a nutshell, GSP allows for concurrent and offline operations to be performed on replicated pieces of data, called *data models*. Each data model defines an operation which can be applied over it (i.e. *Update*). Updates over a data model are aggregated in a log of operations. Furthermore, each data model is associated with a function which returns the value of an instance of the model given the update log and an initial value (i.e. *Read*). In our counter example the *Read* function returns the length of the update log, which contains increment operations.

Distributed clients each have an instance of the data model. Clients can perform updates on their local instances of this data model. GSP ensures that all clients eventually read the same value for their instance of the data model. To do so it assumes that clients communicate through a *reliable total order broadcast* (RTOB) [7] communication medium (i.e. all messages are reliably received by all clients in the same order). CAPtain.js ensures RTOB through a client-server (i.e.

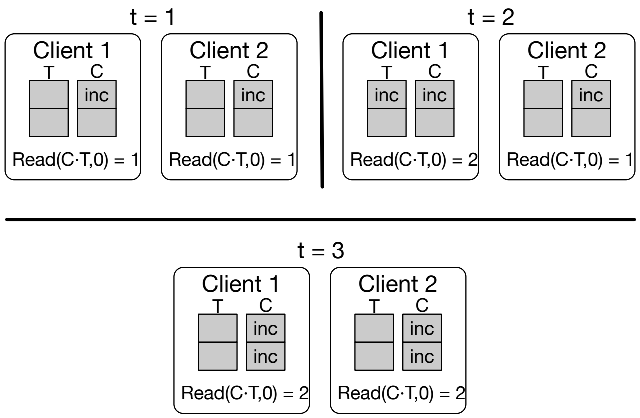

Figure 2. Example run of the GSP algorithm on the counter example.t=1, consistent starting state.t=2, *Client 1* performs an increment.t=3, consistent final state.

slave and master replicas) architecture, where the server acts as a broadcaster.

Offline operations are supported by letting each instance of a data model maintain two logs of updates: *committed* and *tentative* updates. The former represents the last known global log of updates. The latter contains a log of update operations which are yet to be broadcasted. For instance, operations performed while the client lost connection. Applying *Read* to the committed and tentative logs returns the current value for an instance of a data model.

Whenever a client performs an update on its instance of a data model the update is added to the tentative log. Furthermore, the update is broadcasted to all clients. Upon receiving an update each client adds the update to the committed log. If a client receives its own update it removes said update from the tentative log. Figure 2 depicts how GSP ensures eventual consistency for our counter example. At *t=1 Client 1* and *Client 2* are in a consistent state. Both have a single update operation (i.e. an *inc*) in their committed log. Performing the *Read* operation with the committed log, tentative log and the initial value 0 therefore returns 1 for both clients. At *t=2 Client 1* performs an update which is added to *Client 1*'s tentative log. This update is broadcasted by *Client 1* but not yet received by *Client 2*. The clients are therefore in a temporarily inconsistent state given that applying the *Read* function returns a different value for both clients. At *t=3 Client 2* and *Client 1* have received the broadcast. Both clients add the operation to their committed log and *Client 1* removes the update from its tentative log. Applying the *Read* function returns a consistent value (i.e. 2) for both clients.

Explaining the complete workings of the algorithm (i.e. how it deals with message loss and efficiency optimisations) would take us out of the scope of this paper. However, the implementation used in CAPtain.js adheres to an optimised and fault tolerant version of the algorithm. We refer the reader to [4] for an in-depth explanation of optimisations for the algorithm as well as how it deals with message loss and disconnections.

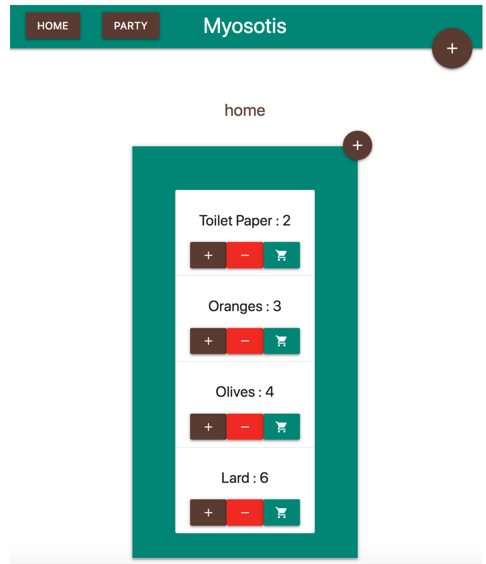

Figure 3. Screenshot of Myosotis

4.2 Sequential Consistency through Far References

CAPtain.js implements consistents using *far references* [6]. The node in the network which instantiates a new consistent is said to be the *server* for said instance. All other nodes have *proxies* to this instance. Each method invocation or field access on such a proxy results in an asynchronous message sent to the server and returns a promise. The server performs the invocation or field access and resolves the promise with the return value of the invocation or access. In case the connection between a proxy and the server is lost all messages are buffered by the proxy until the connection is re-established. This mechanism guarantees sequential consistency: the operations issued by a node in the network follow that node's program execution. Moreover, the result of all operations on a consistent is the same as if these operations were executed in some sequential order.

5 Availability versus Consistency: A Functional Choice

To showcase how availables and consistents are used to build real-world web applications we detail the implementation of Myosotis [3]: a collaborative grocery list application. Figure 3 shows a screenshot of the application.

The navigation bar at the top of the screen shows the grocery lists linked to the user's account. Moreover, it allows to create new grocery lists. The lower part of the screen shows all the items contained within one of these grocery

[3] https://github.com/myter/Myosotis

lists. The quantity needed of each item can be incremented or decremented. Moreover, a user is able to mark an item as *bought* using the *"shopping cart"* button. Myosotis requires both available and consistent functionality:

Available Functionality Once a user logs in it receives a replica of all the grocery lists created for its account. A user is always able to create a new list, add an item to a list or change an item's quantity (i.e. increment or decrement it). Two users logged into the same account might therefore temporarily witness different values for the collection of lists, items in a list or the quantity of an item.

Consistent Functionality Marking an item as *bought* happens strongly consistent. In other words, an item can only be marked as *bought* once by a single user. Consequently, this functionality is only available to users which are connected to the Myosotis server.

We discuss the implementation of both kinds of functionality using Captain.js.

5.1 Implementing Availability in Myosotis

```
class AccountLists extends Available{
    owner : string
    lists : Array<GroceryList>

    @mutating
    newList(list : GroceryList){
        this.lists.push(list)
    }
}
```

Listing 5. Defining the lists per account.

```
1  class GroceryList extends Available{
2      listName    : string
3      items       : Map<string,number>
4
5      @mutating
6      addItem(itemName){
7          if(this.items.has(itemName)){
8              this.incQuantity(itemName)
9          }
10         else{
11             this.items.set(itemName,1)
12         }
13     }
14     @mutating
15     remItem(itemName){
16         this.items.delete(itemName)
17     }
18     @mutating
19     incQuantity(itemName){
20         this.items.get(itemName)+=1
21     }
22
23     @mutating
24     decQuantity(itemName){
25         let curr = this.items.get(itemName)
26         if(curr -1 <= 0){
27             this.remItem(itemName)
28         }
29         else{
30             this.items.get(itemName)-=1
31         }
32     }
33 }
```

Listing 6. Defining individual grocery lists

We implement Myosotis' available functionality using two availables: one which maintains all lists of an account and one which maintains the state of individual lists. The definition of the former is given by Listing 5. The *AccountLists* available maintains an array of all lists created for a particular account. It defines a single mutating method *newList* which adds a newly created *GroceryList* instance for the account. The definition of *GroceryList* is given by Listing 6. It maintains a hashmap which pairs the name of a specific item to the desired quantity. Moreover, it provides a number of mutating methods to add or remove items from a list and increment or decrement the quantity of a specific item.

The functionality provided by *AccountLists* and *GroceryList* instances is available by design. Users are able to create new lists or update the state of a specific list on local replicas, CAPtain.js' runtime ensures that the state of these replicas is kept eventually consistent. CAPtain.js programmers are freed from manually maintaining replicated state. However, programmers cannot be completely oblivious to the inherent concurrency of availables. For example, the *addItem* method (see line 6 in Listing 6) needs to check whether an item is already present in the list. If this is the case the quantity of the desired item is incremented. This is needed due to the possibility of two users concurrently adding the same item to the list. Similarly, before retrieving the quantity of an item in the *items* map one needs to ensure that the item is present in the map (i.e. it might have been deleted by another user). We omit this sanity check from Listing 6 for the sake of brevity.

5.2 Implementing Consistency in Myosotis

```
class Bought extends Consistent{
    bought : Map<string,Array<string>>

    buyItem(listName,itemName){
      let lst = this.bought.get(listName)
      if(lst.includes(itemName)){
          return false
      }
      else{
          lst.push(itemName)
          return true
      }
    }
}
```

Listing 7. Defining the bought markers for items

Listing 7 shows the consistent responsible for Myosotis' marking functionality. A single instance of *Bought* is created per account by the server. Upon connection of a client the server returns a replica of this instance together with an *AccountLists* replica. *Bought*'s single method *buyItem* allows a user to mark a specific item as bought. The method returns whether marking the item happened successfully. In other words, the method returns false if the item had previously been marked as bought. *Bought* guarantees sequential consistency. Only the server is able to execute the *buyItem* method, which ensures that an item can only be bought once.

Moreover, clients which have lost connection with the server are unable to execute the method.

5.3 Disseminating the Data and Reacting to Change

```
1 setupPubSubServer(serverAddress, serverPort)
2 var loginTopic = new Topic("LoginReq")
3 var loginResTopic = new Topic("LoginResp")
4 let reps = new Map()
5 sub(loginTopic).each((accountName)=>{
6  let lists
7  let bought
8  if(reps.has(accountName)){
9   [lists, bought] = reps.get(accountName)
10  }
11  else{
12   lists = new AccountLists()
13   bought = new Bought()
14   reps.set(accountName,[lists, bought])
15  }
16  pub([lists, bought], loginResTopic)
17 })
```

Listing 8. The Myosotis server

Listings 8 and 9 show simplified implementations of the Myosotis server and clients respectively. The server maintains a hashmap (i.e. *reps*) which contains the replicas (i.e. an instance of *AccountLists* and *Bought*) associated to each account. Whenever a client logs into the Myosotis server (i.e. by publishing a login request) the server either creates a new *AccountLists* and *Bought* replica (see line 12) or it fetches the previously stored ones (see line 9). Subsequently the server returns the client's replicas (i.e. by publishing a login response, see line 16).

```
1 setupPubSubClient(serverAddress, serverPort)
2 pub(accountName, loginTopic)
3 sub(loginRepTopic).each(([lists, bought])=>{
4  lists.onGlobal(()=>{
5   //update UI
6  })
7  lists.onLocal(()=>{
8   //update UI
9  })
10 })
```

Listing 9. The Myosotis client

Clients start by publishing a login request (see line 2 in Listing 9) using their *accountName* (which we assume is provided on start-up). Whenever a client receives a login response it installs *onGlobal* (see line 4) and *onLocal* (see line 7) listeners on the *AccountLists* replica. Whenever a new grocery list is created or an item is mutated these listeners will ensure that the client's UI remains up to date.

6 Availability versus Consistency: A Performance Choice

Programmers implement different parts of their distributed systems using availables or consistents based on *functional* requirements (e.g. offline availability). However, availables and consistents also differ in their *performance* characteristics. Assume that an operation *o* changes the state of a given available or consistent replica *r*. We define the following two performance characteristics:

Time to Consistency (TC) is the time required for the state change induced by *o* to be visible on all other replicas.

Time to Local Change (TLC) is the time required for the state change induced by *o* to be visible on *r* itself (i.e. within the node in which *o* is applied).

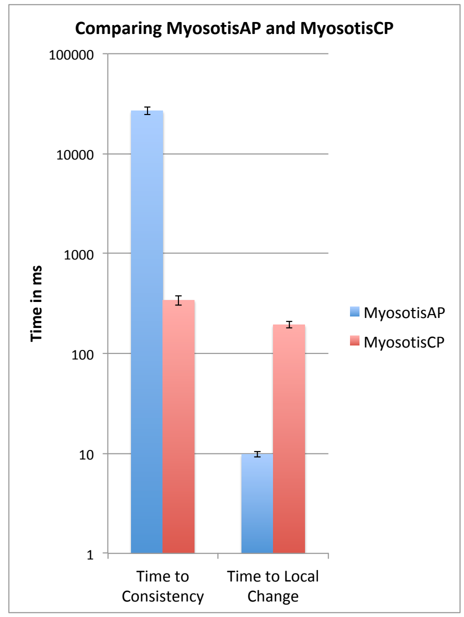

Figure 4. Comparing MyosotisAP and MyosotisCP. Error bars indicate the 95% confidence interval.

6.1 Comparing Availables and Consistents

In order to showcase how availables and consistents differ with regards to these characteristics we perform micro-benchmarks using two versions of Myosotis. The first version (i.e. *MyosotisAP*) is discussed in detail in Section 5. In a nutshell, all list functionality (i.e. creating lists, adding items to a list, etc.) is implemented by two availables. The only consistent in *MyosotisAP* is responsible for marking items as bought. The second version (i.e. *MyosotisCP*) solely uses consistents. In other words, *MyosotisCP* clients are unable to use any functionality while being offline. However, all changes to lists are always ensured to be strongly consistent across all clients.

The micro-benchmarks are conducted on an Ubuntu 14.04 server with two dual core Intel Xeon 2637 processors (2 physical threads per core) at 3.5 GHz with 265 GB of RAM using CAPtain.js version 0.5.0. For each version of Myosotis we simulate 50 clients concurrently adding 10 items to a shared grocery list.

Figure 4 shows the results of these benchmarks that highlight a fundamental difference between availables and consistents. In *MyosotisAP* the TLC is roughly a factor 1000 faster than the TC. In comparison, this difference is substantially more nuanced for *MyosotisCP* where the TLC is roughly twice as fast as the TC. There are two reasons for this difference. First, availables are able to perform operations immediately which results in low TLC. However, maintaining all available replicas eventually consistent produces a significant performance overhead: Each operation must be sent to the server which conceptually replays the entire log of operations. Subsequently this log is sent to *all* replicas which in turn replay all operations as well. Second, operations are only performed by a consistent replica if it can guarantee strong consistency. In other words, such operations only require a single round trip message to the server. Although this negatively impacts TLC, this reduces the synchronisation overhead needed to maintain the global state strongly consistent. The difference between TLC and TC is therefore much smaller for consistents.

6.2 Availability on Demand

MyosotisCP models the application's entire state using consistents. In other words, *AccountLists* and *GroceryList* (see Listings 5 and 6 in Section 5.1) extend *consistent* rather than *available*. As shown in Section 6.1, implementing MyosotisCP entirely using consistents reduces the *time to consistency*. This comes at the cost of MyosotisCP not providing any offline-available functionality.

Through CAPtain.js' built-in *freeze* and *thaw* methods we are able to dynamically make a trade-off between performance and offline availability. To do so, the server needs to switch from a consistent to an available implementation of all list-related functionality at run time. Concretely, by pressing a button clients inform the server that all lists linked to their account should be made available. Conversely, clients can inform the server that their lists are no longer required to be available. It is important to note that this run-time switching can only happen if all logged-in clients linked to a specific account are connected to the server.

```
1  function goOffline(){
2   pub(accountName,offTopic)
3  }
4  function goOnline(){
5   pub(accountName,onTopic)
6  }
7  sub(offResTopic).each((listsAV)=>{
8   //continue using available AccountState
9  })
10 sub(onResTopic).each((listsC)=>{
11  //continue using consistent AccountState
12 })
```

Listing 10. The MyosotisCP client

Listings 11 and 10 show a simplified version of the MyosotisCP server and client. The client implements two functions (i.e. *goOffline* and *goOnline*) which are invoked whenever the user presses a button. These functions notify the server that the lists linked to the specified account should be made available or consistent. This is achieved for both functions by publishing the account name under the correct topic.

```
1  let offTopic = new Topic("OffReq")
2  let offResTopic = new Topic("OffResp")
3  let onTopic = new Topic("OnReq")
4  let onResTopic = new Topic("OnResp")
5  let reps = new Map()
6  //Login code omitted
7  sub(offTopic).each((accountName)=>{
8   let [lists,bought] = reps.get(accountName)
9   let listsAV = thaw(lists)
10  reps.set(accountName,[listsAV,bought])
11  pub(listsAV,offResTopic)
12 })
13 sub(onTopic).each((acountName)=>{
14  let [lists,bought] = reps.get(accountName)
15  let listsC = freeze(lists)
16  reps.set(accountName,[listsC,bought])
17  pub(listsC,onResTopic)
18 })
```

Listing 11. The MyosotisCP server

The server reacts to these client requests as follows. If a user requests their lists to be available (see line 7 in Listing 11) the server thaws the consistent (i.e. an instance of the consistent version of *AccountLists*) representing the lists linked to the specified account. The result of *thaw* is an available copy (i.e. both state and methods) of the *AccountLists* consistent. The server publishes this available copy to all clients logged in to the specified account, which use the available (see line 7 in Listing 10) from that point on.

If a user requests their lists to be consistent again (see line 13) the server freezes the previously thawed *AccountLists* instance. The result of *freeze* is a consistent copy of the provided available. The server publishes the consistent copy which allows clients to continue running the MyosotisCP application as it was before requesting availability.

7 Related Work

This work is heavily inspired by *Repliqs* [5]. A *repliq* is a first-class replicated object which is kept eventually consistent across clients through the *global sequence protocol* [4]. In this regard *repliqs* and *availables* are essentially the same. *Repliqs* allow programmers to implement the *AP* parts of distributed systems. We extend the work presented in [5] with constructs to implement the *CP* parts of distributed systems. Moreover, we introduce constructs to convert the *AP* parts of a distributed system into *CP* parts and vice versa.

Correctables [10] are a language construct which allows programmers to perform operations on replicated objects using different levels of consistency. In a nutshell, invoking an operation on a *correctable* will initially return a weakly consistent result after which it will progressively be refined with more consistent results (e.g. strongly consistent results). Additionally, correctables allow programmers to explicitly specify the level of consistency desired for a particular operation. Correctables differ from our approach with regards to the level of granularity on which programmers specify the desired level of consistency. Programmers using correctables specify the desired level of consistency *per operation*. Both approaches also differ from a programmer's perspective. The work presented in [10] provides an API consisting of three methods: *invokeStrong*, *invokeWeak* and *invoke*. These methods allow the programmer to specify the desired level of consistency given an operation to be performed on a replicated object. Moreover, the correctables API allows programmers to implement their own consistency guarantees. In contrast, CAPtain provides a full-fledged distribution model: it enables programmers to define the consistency levels for data types as well as how instances of these data types should be replicated amongst nodes in the network.

Lasp [18] is a distributed programming language whose sole data abstractions are CRDTs [23]. Lasp is therefore able to model the *AP* parts of a distributed system while guaranteeing strong eventual consistency. However, Lasp lacks the programming constructs to implement the *CP* parts of a distributed system.

Dexter [26] is a Java framework which allows programmers to implement various distributed parameter passing semantics. Two of these semantics are of particular interest compared to the work presented in this paper. Pass by *remote reference* is essentially the same as AmbientTalk's *far* references or *E*'s *eventual* references. In other words, using pass by *remote reference* one is able to implement the *CP* parts of a distributed system. Pass by *copy-restore* allows an object to be passed by copy between a server and a client. Changes made to the copy by the server are later restored on the client. To some extend this enables the implementation of the *AP* parts of a distributed system in Dexter. To the best of our knowledge *copy-restore* does not provide any consistency guarantees. In other words, conflicts arising from concurrent modifications are not resolved. In contrast, availables allow concurrent modifications while ensuring eventual consistency.

In the tuple space model [8] processes conceptually access a globally shared memory comprised of data structures called *tuples*. Processes can write, read and remove tuples from this global memory. In the traditional tuple space model as defined by [8] a centralised server maintains the state shared by clients. This model therefore only allows to implement the *CP* parts of a distributed system. Other tuple space models [1, 16] replicate tuples across clients, allowing them to read or write tuples while being offline. However, these models do not account for conflicting updates to the conceptually shared tuple space. They are therefore unable to provide the eventual consistency guarantees required by availables.

E [21] and AmbientTalk [6] both provide language constructs to implement the *CP* parts of a distributed systems (i.e. *eventual* and *far references* respectively). Moreover, AmbientTalk provides *isolates* which are a kind of object that adhere to pass-by-copy semantics. Although *isolates* can therefore be used to implement the *AP* parts of a distributed system they provide no consistency guarantees whatsoever. In other words, the states of two instances of the same *isolate* are never synchronised.

A number of approaches have been proposed which allow programmers to perform operations (e.g. queries) on replicated datastores with various levels of consistency. In contrast, CAPtain provides a general purpose programming model which introduces replicated data as first-class language abstractions (e.g. availables can be provided as arguments to remote method invocations or published/subscribed to).

Using Sieve [15] programmers specify application invariants to help static and dynamic analyses to determine optimal consistency levels for operations on the datastore. Operations which can run under weak consistency are translated to *commutative shadow operations* (i.e. operations on CRDTs). In Quelea [24] programmers write contracts which specify the application-level consistency requirements of operations. The Quelea runtime statically verifies these contracts while a theorem prover maps these contracts to consistency properties which adhere to the contract's semantics. DCCT [27] allows programmers to separate a datastore's objects into *regions*. These regions are annotated with varying degrees of consistency which influences the semantics of the read and write operations one can perform on objects within a region. IPA [12] programmers specify consistency policies by using an extensive annotation system (e.g. one can dynamically specify consistency policies based on the system's latency). Furthermore, IPA's type system allows it to enforce a number of properties at compile time (e.g. weakly consistent values never flow into strongly consistent operations). ConSysT [17] provides consistency specifications at the type level (i.e. values are typed with the desired consistency level). ConSysT's type system guards the programmer from erroneously combining values with different consistency levels (e.g. low consistency values flowing into high consistency computations).

8 Vision and Future Work

CAPtain.js serves as a prototypical implementation of our CAPtain model. As such, it only provides two levels of consistency: sequential consistency and eventual consistency (i.e. not strong eventual consistency). Section 4 discusses how we implement the former using *far references* and the latter using the *global sequence protocol*. Both approaches pose a number of disadvantages: On one hand, far references provide a replication factor of 1. For example, in Myosotis the server maintains the actual instance of the *Bought* consistent while clients only have a proxy (i.e. a far reference) to this instance. On the other hand, the global sequence protocol only guarantees eventual consistency. In contrast to *strong eventual consistency* (e.g. as provided by CRDTs [23]) this means that two concurrent operations on available replicas might conflict.

Our vision is for CAPtain.js to support a multitude of consistency levels. For example, availables would come in multiple flavours (e.g. eventually and strongly eventually consistent). Similarly, programmers would be able to choose a consistent's replication factor (the CAPtain.js runtime would ensure consistency of these replicas through two-phase locking or paxos for example).

CAPtain.js already provides programmers the necessary language constructs to implement this vision. Explaining the intricate details of these constructs would take us beyond the scope of this paper. In a nutshell, CAPtain.js provides a mirror-based metaprogramming API [2] akin to mirrors in AmbientTalk [25]. This allows expert programmers to implement different kinds of consistency by extending the mirrors provided by availables and consistents. For example, strongly eventually consistent availables can be implemented by extending the default available mirror with the functionality provided by general purpose CRDTs [14].

9 Conclusion

Most modern distributed systems provide some form of data replication (e.g. for security, redundancy, offline availability, etc.). As a result, programmers have to face the impossibility result posed by the CAP theorem. In a nutshell, the CAP theorem states that a partition-tolerant system can either guarantee the consistency (CP) or availability (AP) of a replicated piece of data. Most distributed programming languages implicitly make the choice between *CP* and *AP* for the programmer. As a result, programmers are unable to express both the *CP* and *AP* parts of their distributed systems in a single language.

In this paper we present a novel distributed programming model. In this model programmers implement the *CP* and *AP* parts of their systems using dedicated object-oriented language constructs: *consistents* and *availables*. All instances of a consistent object are guaranteed to be strongly consistent. However, method invocations and field accesses are not guaranteed to return a value. Conversely, instances of an available object are only guaranteed to be eventually consistent. However, method invocations and field accesses always return a value.

We provide a prototypical implementation of this model in a TypeScript library called CAPtain.js. We showcase the power of our model and its functional characteristics by implementing a collaborative grocery list application. Through micro-benchmarks we showcase the performance characteristics of availables and consistents.

Acknowledgments

This work is supported by Innoviris (the Brussels Institute for Research and Innovation) through the Doctiris program (grant number 15-doct-07)

References

[1] Elisa Gonzalez Boix, Christophe Scholliers, Wolfgang De Meuter, and Theo D'Hondt. 2014. Programming mobile context-aware applications with TOTAM. *Journal of Systems and Software* 92 (2014), 3 – 19.

[2] Gilad Bracha and David Ungar. 2004. Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 331–344.

[3] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00)*. ACM, New York, NY, USA, 7–.

[4] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 568–590.

[5] Tim Coppieters, Wolfgang De Meuter, and Sebastian Burckhardt. 2016. Serializable Eventual Consistency: Consistency Through Object Method Replay. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '16)*. ACM, New York, NY, USA, Article 3, 3 pages.

[6] Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. 2014. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures* 40, 3–4 (2014), 112 – 136.

[7] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.* 36, 4 (Dec. 2004), 372–421.

[8] David Gelernter. 1985. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (Jan. 1985), 80–112.

[9] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59.

[10] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 169–184.

[11] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[12] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NY, USA, 279–293.

[13] Martin Kleppmann. 2015. A Critique of the CAP Theorem. *arXiv preprint arXiv:1509.05393* (2015).

[14] M. Kleppmann and A. R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 2733–2746.

[15] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical*

Conference (USENIX ATC 14). USENIX Association, Philadelphia, PA, 281–292.

[16] Marco Mamei and Franco Zambonelli. 2004. Programming Pervasive and Mobile Computing Applications with the TOTA Middleware. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04) (PERCOM '04)*. IEEE Computer Society, Washington, DC, USA, 263–.

[17] Alessandro Margara and Guido Salvaneschi. 2017. Consistency Types for Safe and Efficient Distributed Programming. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs (FTFJP'17)*. ACM, New York, NY, USA, Article 8, 2 pages.

[18] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. ACM, New York, NY, USA, 184–195.

[19] Christopher S. Meiklejohn. 2017. On the Design of Distributed Programming Models. In *Proceedings of the Programming Models and Languages for Distributed Computing (PMLDC '17)*. ACM, New York, NY, USA, Article 1, 6 pages.

[20] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. Springer-Verlag New York, Inc., New York, NY, USA, 308–333.

[21] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers: Programming in E As Plan Coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing (TGC'05)*. Springer-Verlag, Berlin, Heidelberg, 195–229.

[22] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. Parallel and Distributed Web Programming with Actors. In *Programming with Actors*. Springer, 3–31.

[23] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400.

[24] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 413–424.

[25] Mostinckx Stijn, Van Cutsem Tom, Timbermont Stijn, Gonzalez Boix Elisa, Tanter ÃĽric, and De Meuter Wolfgang. [n. d.]. MirrorâĂŘbased reflection in AmbientTalk. *Software: Practice and Experience* 39, 7 ([n. d.]), 661–699.

[26] Eli Tilevich and Sriram Gopal. 2011. Expressive and Extensible Parameter Passing for Distributed Object Systems. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 3 (Dec. 2011), 26 pages.

[27] Nosheen Zaza and Nathaniel Nystrom. 2016. Data-centric Consistency Policies: A Programming Model for Distributed Applications with Tunable Consistency. In *First Workshop on Programming Models and Languages for Distributed Computing (PMLDC '16)*. ACM, New York, NY, USA, Article 3, 4 pages.