Continuation Complexity: A Callback Hell for Distributed Systems

Edgar Zamora-Gómez, Pedro García-López, and Rubén Mondéjar

Department of Computer Engineering and Mathematics,
Universitat Rovira i Virgili, Spain
{edgar.zamora, pedro.garcia, ruben.mondejar}@urv.cat

Abstract. Designing and validating large-scale distributed systems is still a complex issue. The asynchronous event-based nature of distributed communications makes these systems complex to implement, debug and test. In this article, we introduce the continuation complexity problem, that arises when synchronous invocations must be converted to asynchronous event code. This problem appears in most Actor libraries where communication is mainly asynchronous, and where a synchronous call to other Actor would block the current Actor, precluding the processing of incoming messages.

We propose here a novel parallel invocation abstraction that considerably simplifies the continuation complexity problem for distributed actor systems requiring non-blocking synchronous invocations. Our parallel abstraction extends the message passing concurrency model to support concurrent interleavings of method executions within a single Actor. We present here two candidate solutions for implementing such parallel calls: one based on threads and locking, and other based on green threads and continuations.

We evaluated the simplicity of our solution implementing a well known distributed algorithm like Chord (ring-based structured overlay). We compare our Actor implementation of Chord with three different simulators (PeerSim, PlanetSim, Macedon). This validation demonstrates that our approach is superior in simplicity (less LoC) and complexity (less McAbe complexity), envisaging its great potential for distributed systems scenarios.

Keywords: Actor Model, Object Actors, Continuation Complexity

1 Introduction

Nowadays, modeling, programming, and validating distributed systems is a complex issue requiring advanced skills. On the one hand, multithread programming, locks and concurrency-control can considerably complicate the development of any middleware library. In general, programming event-based systems using callbacks is a challenging task. Callback coordination is complex because different code fragments must manipulate the same data and the order of execution is unpredictable.

In literature, the problem of callback management is known as Callback Hell [4]. A recent analysis of Adobe desktop applications revealed that event handling logic caused nearly a half of the bugs reported.

Even if this problem is normally associated to user interface code, the Callback Hell is also very relevant in the development of distributed systems.[6] [8] In particular, many distributed systems like Actor libraries, event-based simulators [5] [7], and event-based server libraries like node.js impose an asynchronous style of programming.

But the asynchronous event-based nature of distributed protocols implies tangled code using message handlers and callbacks that is difficult to follow and maintain. These problems get even worse when the distributed algorithms rely on Remote Procedure Call (RPC) semantics that require complex state maintenance between messages.

In these cases, a brief and succinct algorithm written with sequential code and RPCs will have to be broken in a number of separated callbacks. Callbacks then become the old *goto* statement revamped for distributed systems.

In this article we first identify and formalize the so called Continuation Complexity problem, which arises when synchronous RPC code must be converted to asynchronous messages and handlers. We also propose a solution to the Continuation Complexity problem for Actor libraries. To this end, we present a novel parallel invocation abstraction that permits blocking synchronous calls to other actors that do not stall the current Actor thread of control. Finally, to validate our approach, we evaluate the simplicity of our solution implementing a well-known distributed algorithm: a ring-based structured overlay (i.e., Chord).

2 Continuation Complexity Problem

The Continuation Complexity Problem arises in distributed systems when synchronous RPC code must be converted to asynchronous messages and handlers. We call this problem Continuation Complexity because it is directly related to the concept of Continuation [9] in programming languages. In distributed settings, the developer is implicitly implementing its own continuation when he must split a synchronous call in different code fragments using messages and handlers (callbacks). Normally, the programming language uses a call stack for storing the variables its functions use. But in this case, the developer must explicitly maintain this information between different messages.

The complexity of programming event-based systems with callbacks is also very relevant in interactive user interface systems. For example, a recent solution to the so-called Callback Hell is the reactive programming paradigm [3].

Another interesting solution in the .NET platform are the powerful *async* and *await* abstractions [1]. They simplify asynchronous programming and improve the clarity of code, and thus reducing the Callback Hell problem. Nevertheless these abstractions are not transparent to the developer and they are not aimed for single-threaded Actor libraries.

In distributed systems, we want to outline two major programming models that can help to cope with asynchronous event programming : distributed state machines and Actor libraries.

Distributed state machines are the classical formal model for the implementation of distributed systems. Macedon [10] is a concise Domain Specific Language (DSL) that aims to cover the entire cycle of distributed systems including design, implementation, experimentation, and evaluation. In any case, we compare our implementation of Chord with Macedon one in Section 5.

The *Actor model* [2] has inspired many middleware solutions since it provides an elegant model of concurrent communication, which treats Actors as the universal primitives of concurrent computation.

A recent proposal for distributed Actors is Akka. Akka has an advanced programming model where Futures are integrated with Actor-based receive operations. They provide TypedActors enabling synchronous method invocation over Actors, but they clearly recommend to use asynchronous communications and Futures to avoid blocking the current Actor execution.

This is not an exclusive problem of Actor models, but of the development of event-based system. In this line, event-based simulators [5] [7] require that the algorithms must be converted to asynchronous messages and message handlers.

The Continuation Complexity is a recurrent problem in distributed systems based on RPC semantics like Peer-to-Peer (P2P) algorithms and overlays (e.g., Chord or Kademlia), consensus algorithms (e.g., Paxos or 2PC) and distributed applications (e.g., key-value stores or replication algorithms).

The Continuation Complexity of a function depends directly on the number of RPCs inside this function plus the number of iterators containing RPCs in that function. Every RPC will produce three new code fragments: two for the continuation and one for the timeout code. With the continuation we refer to the code before the RPC that sends the request message, and the response handler containing the code that continues from that point onward. Furthermore, every RPC must contain a timeout handler that will also continue after the RPC if no response was received in the timeout time.

On the other hand, iterators containing RPCs must also be considered in the cost since the complexity increases as the response handler must control the state of the iteration in every response. This means that the handler must jump back and also repeat the sending of requests if the iteration has not finished.

The continuation complexity is directly related to the Cyclomatic complexity of the method including RPCs. When the control flow of the method is very complex (high Cyclomatic Complexity) the continuation complexity is also very high. In this case, the original control flow must be rebuilt using messages and message handlers.

2.1 A simple example: Chord

Many distributed systems make extensive use of RPCs. Describing their algorithms using object oriented notation is straightforward, since an invocation of a method in a node implies a RPC to that node. In this line, the Chord [11] distributed routing algorithm is a suitable example of this approach.

Chord organizes nodes in a structured ring overlay where every node contains a local routing table with references to other nodes. Chord is also defined as a Distributed Hash Table (DHT) or Key Based Routing (KBR) layer, since the space of identifiers is uniformly distributed among nodes thanks to consistent

hashing. In Chord, the routing algorithm is clockwise and every node is responsible of the identifiers between itself and its predecessor in the ring overlay.

Algorithm 1 Chord Routing Mechanism

```
function FIND_PREDECESSOR(id)
    n1 ← this_node
    while id is not between (n1,n1.successor] do
        n1 ← n1.closest_preceding_finger(id)
    return n1
function CLOSEST_PRECEDING_FINGER(id)
    for i ← N down to 1 do
        if finger[i].node is between (n, id) then
            return finger[i].node
    return this_node
```

As we can see in the Algorithm 1, looking for a key (*id*) implies accessing the routing tables of nodes to find the closest node to that key. So if we invoke the *find_predecessor(id)* operation in a specific node, it will look for a node n where the identifier is in the range (n, n.successor). For example, in a Chord overlay with nodes N1,N7,N18, and N30, *find_predecessor(9)* will return the node N7.

This is a very interesting example, since closest_preceding_finger is a RPC invocation to other nodes. Indeed, it is a clear mechanism of iterative routing where the node performing the search must ask sequentially nodes that are closer to the destination.

Note that *closest_preceding_finger* is returning a reference to another Node. In a remote setting, this would imply sending a reference to the remote object.

Nevertheless, the problem arises when this code must be converted to purely asynchronous messages. This is happening in most networks and distributed simulators, but also in most Actor libraries. In this case, the developer must renounce to RPCs completely, and convert each RPC to two messages and message handlers: Message Request and Message Response.

The problem gets even worse when RPCs are invoked inside control blocks like iterators or conditionals. In these cases, the developer must also maintain the state of variables (context) between different messages. As we can see, this is happening in the *find_predecessor* method that is calling a RPC inside a *while* iterator. Even worse, in the control structure the condition can also contain RPCs (*n.successor*).

The Fig. 1 shows the control flow graph of the previous algorithm 1 when divided in different code fragments. In this case, Actor 1 is the control flow of *find_predecessor* and Actor n is the control flow of *closest_preceding_finger*. The RPC call inside Actor 1 must be converted in messages to Actor n, and the return results will then resume the state of the control flow in Actor 1.

As we increase the number of RPCs, the complexity of breaking the code is even higher. Imagine now that we want to implement parallel queries to different nodes to increase the robustness of the iterative routing. If we must invoke now

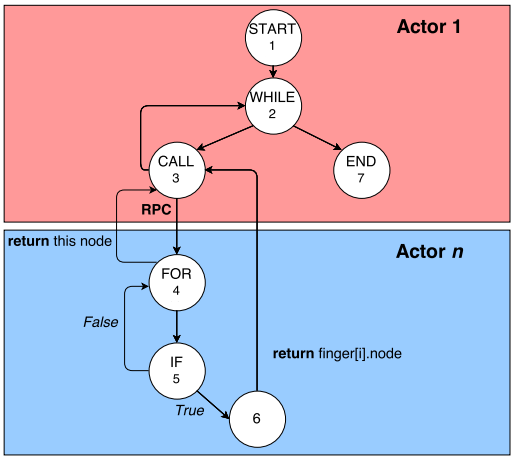


Fig. 1: Control flow Graph

closest_preceding_finger in three nodes, the code must then handle the different messages and resume the state when necessary.

As we can see, a simple code written with RPCs must be scattered in different code fragments making more difficult its legibility and maintenance. This is a clear example of the Callback Hell in distributed systems.

3 Overcoming the continuation complexity problem in Actor models

The Continuation Complexity problem leads us to a paradox in Actor models: synchronous blocking calls are needed to correctly implement RPCs in order to avoid breaking the code into asynchronous messages and handlers. But Actor models are purely asynchronous, and using synchronous calls would block the calling Actor unique thread of control until the response arrives.

The aforementioned paradox also represents a burden for Object Oriented Actor libraries. If only asynchronous calls are allowed, the resulting code breaks with the traditional object oriented paradigm and complicates the resulting code.

To solve this problem, we present a novel method invocation abstraction called **parallel**, enabling synchronous invocations to other Actors, without stalling the current Actor's thread of control. Our parallel calls enable concurrent interleavings of method executions within a single Actor.

3.1 Concurrency Control

The major challenge is to enable concurrent interleavings of method executions within a single Actor. We mainly want to allow the main Actor thread to continue processing incoming method requests while the parallel thread is blocked waiting for a response in a remote Actor due to a synchronous invocation.

We present two solutions: one based on threads and locks and another based on green threads and continuations,

Solution 1: Threads: The Actor implementation will spawn normal threads in the Scheduler and Parallel Methods threads. To achieve consistency between Scheduler and Parallel threads, we use traditional Thread Lock mechanisms that prevent multiple threads to access the same object at the same time. Similar to the Monitor Object pattern, we use a Lock in the Actor object that is only activated when parallel threads are spawned. We aim to provide here a simple

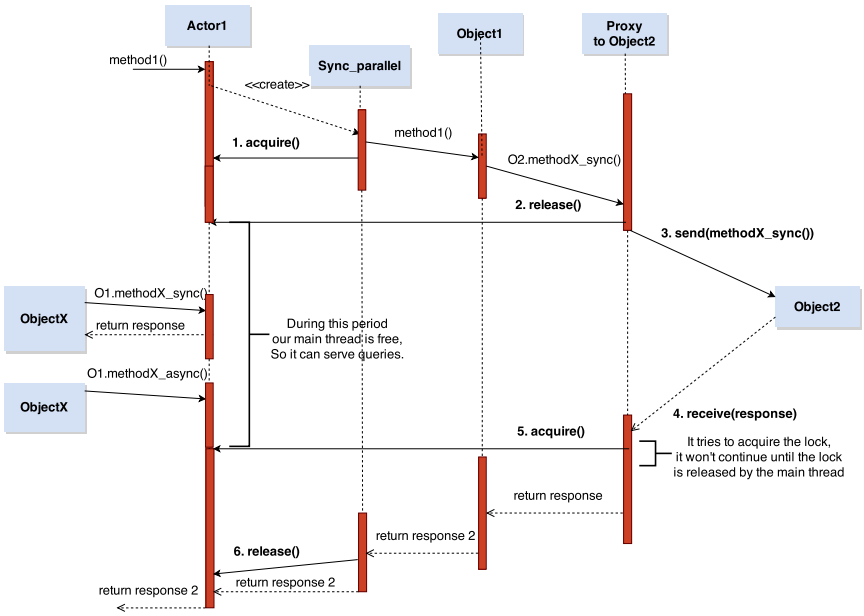


Fig. 2: Concurrency Flow Diagram

solution to demonstrate that parallel threads can coexist with the Actor main thread without conflicting with the servant object shared state. Our lock mechanism is completely transparent to the developer, so that the simplicity of the message passing concurrency model is not affected.

In Figure 2 we can see how our solution handles the concurrency problem. In this flow diagram we can observe where it uses the Lock acquire and Lock release primitives in a call flow. In addition, it is necessary to know that a single Lock is shared between Actor and Parallel Threads wrappers for each Actor. The life cycle is the following:

1. **Acquire:** When a call is incoming, if it exists some parallel call in the object, we need to acquire the lock to be sure that only one thread at this time is using the object. In Figure 2, we show an example with a parallel synchronous call. At this moment, a parallel thread takes the control of the object, and any other thread can not access to the object.

2. **Release:** The parallel thread sends the synchronous invocation to a remote actor. In this moment, the parallel actor releases the lock so that the main thread or other parallel threads can continue working. This is the key point where the Actor is not blocked until the response arrives, and it can process incoming messages.
3. **Acquire:** While the parallel thread is waiting the response, our main thread and other parallel threads can serve other petitions. When the response arrives to the Proxy, the Parallel thread will be able to continue executing the method. Nonetheless, it must try to acquire the Lock, because it is possible that the Lock is now in possession of another Thread.
4. **Release:** Finally, when the multicalls return object method ends, the parallel synchronous wrapper will release the Lock, and it will end its process.

As we can see, our solution ensures that only one thread access the shared state, but also that waiting for a response in a parallel thread will not preclude the main thread to process incoming messages. Obviously, we control whether the response will not return using a timeout. Using this system we guarantee that the main thread will not be blocked even if the response does not arrive.

Our solution permits to increase the Actor service time because it avoids blocking the Actor during synchronous invocations. We guarantee the correct interleavings of Actor parallel and main threads. Furthermore, we maintain the simplicity of message passing concurrency since the developer is still unaware of this concurrency control mechanisms. He must only tag the appropriate methods as *parallel* when necessary.

Solution 2: Micro threads and continuations: In this case, we can use Fig. 2 but removing the acquire and release invocations. In that case, we don't need to use a lock system. This is because in a single-threaded environment, two microthreads cannot modify the same state at the same time because their execution is sequential.

We assume here that the *send* and *receive* primitives in microthreads will execute a context-switch to other green thread processing the communication to other Actor. Following, we will try to explain better this process, step by step:

1. **send:** When a proxy sends a synchronous message to another Actor, it automatically releases its control to other microthread.
2. **receive:** Parallel microthread will be inactive until it will receive a message. At moment that it will receive a message it will wake up and wait for its turn to continue the method execution. The implicit continuation implies that the code continues from that point.

Since *send* and *receive* policies are executed inside the Proxy, the developer is completely unaware of context-switches and continuations. Furthermore, in this case there is no need to use locks to prevent concurrent access.

4 PyActive Abstractions

We present an Object Oriented implementation of the Actor model that is designed to support synchronous, asynchronous and parallel calls. We have implemented a prototype of this Actor library in Python that supports different

remote transport protocols, and threading models: Python system threads and Stackless cooperative threads.

Our programming model for Object Actors provides explicit mechanisms for Actor creation and location, method invocation abstractions in actors, and pass by reference of Actor entities. Let us review these mechanisms in our Actor programming model.

We provide several **call abstractions** in Actors: asynchronous calls, timed synchronous calls, parallel calls and pass by reference.

An important difference in our model is that we introduce different types of method requests. The developer must explicitly establish the types of methods using annotations or meta-information that can be processed by the Actor library. As we can see in Figure 3, we include this meta-information as properties of the class. These properties explicitly tell the Actor which methods are exposed and their different types.

Asynchronous calls are one-way invocations that do not require a response from the Actor. These are the usual calls implemented in Actor libraries since they naturally adapt to the asynchronous message passing concurrency model.

Timed Synchronous calls are two-way blocking invocations where the client waits for the response from the server during a certain time (timeout). If the timeout is reached an exception is triggered. This kind of calls are not usually recommended nor even implemented in Actor libraries since the entire calling Actor is blocked until the response arrives. This can be appropriate in state-machine protocols but it can also prevent the Actor for continuing its normal operation. But we claim that they are completely necessary to avoid the aforementioned continuation complexity.

Parallel calls imply the processing of the method invocation in a parallel thread in the Actor. As explained before they must be used to avoid stalling the current Actor when a method performs a synchronous blocking request to other Actor.

Pass by reference calls are invocations where either the parameters or the result contain a reference to another Actor. References to Actors (Proxies) must be explicitly tagged by the developer for performance but also for correctness. It is important not to confuse Objects and Actors references: where the former are restricted to a single address space, the latter can span different locations.

The best way to understand the simplicity of our approach is to implement well-known distributed algorithms.

4.1 A complete example : Chord

As stated before, Chord [11] is an excellent example of a distributed algorithm that makes extensive use of RPCs. We implemented a Python object oriented version of Chord (following the original pseudo-code) in 217 lines of code.

We can see in Figure 3 a fragment of the `Chord implementation` in our library. The original Chord implementation using plain Python objects only required some modifications that we detail now.

To begin with, the class must contain meta-information in four variables (*_sync*, *_async*, *_ref*, *_parallel*) to specify the type of methods that will be exposed remotely by the Actor.

```
class Node(object):
    _sync = {'closest_preceding_finger':'2'
             , ...}
    _async = ['set_successor','fix_finger'
              ,...]
    _ref = ['closest_preceding_finger', ...]
    _parallel = [ 'fix_finger', 'stabilize']

     def find_predecessor(self, id):
         try:
              if id == int(self.id):
                   return self.predecessor
              n1 = self.proxy
              while not betweenE(id,
                   int(n1.get_id()),
                  int(n1.successor().get_id())):
                   n1 = n1.
                     closest_preceding_finger(id)
              return n1
         except(succ_err):
              raise succ_err()
         except(TimeoutError):
              raise TimeoutError()
```

```
def closest_preceding_finger(self, id):
    try:
        for i in range(k - 1, -1, -1):
            if between(int(self.finger[i]
                .get_id()), int(self.id), id):
                return self.finger[i]
        return self.proxy
    except(TimeoutError):
        raise succ_err()

def fix_finger(self):
    if(self.currentFinger <= 0 or
        self.currentFinger >= k):
        self.currentFinger = 1
    try:
        self.finger[self.currentFinger] =
        self.find_successor(
        self.start[self.currentFinger])
    except:
        None
    finally:
        self.currentFinger += 1
```

Fig. 3: Chord Implementation

Moreover, it is important that developers clearly distinguish when they are using object references, and when they are using Actor references. A special case is the reference to the current object (self in Python). If one Actor uses a Proxy to itself to invoke a method, it could create a deadlock. To avoid this, our Actor library sets the *self.proxy* variable to a special Proxy that avoid conflicts, and that can be passed by reference to other Actors. We can see in find_predecessor and closest_preceding_finger how they use *self.proxy* to refer to the current object and avoid conflicts. Note that in Chord, the routing table (finger table) may contain references (Proxies) to the current Actor.

Apart from these important changes, the developer just must be aware of catching the exceptions that may be produced by invoking other Actor (Timeouts). The resulting code is very simple and it can now be executed in multiple machines in a transparent way.

5 Evaluating the expressiveness and simplicity of our approach

In this experiment we are comparing the complexity of the Chord algorithm implementation in four platforms: Macedon, PlanetSim, PeerSim, and PyActive.

PlanetSim and PeerSim are P2P event simulators implemented in the Java language. They are very popular and widely used in the P2P community and they provide a simple framework for developing decentralized scalable algorithms.

Macedon is a Domain Specific Language (DSL) that generates code for the Ns-2 simulator and C++ code with sockets for experimentation. The DSL is based on event-driven finite state machines and it claims a reduction in lines of code and simplicity of the implemented distributed algorithms.

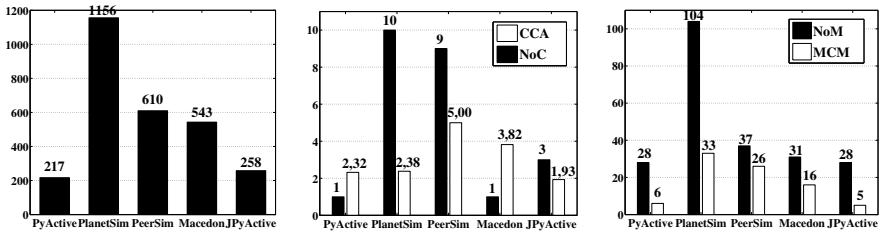
Even if we are comparing different programming languages (Java in PeerSim and Planetsim, DSL in Macedon, Python in our Actor library) the comparison is useful to understand different approaches of implementing the same Chord algorithm. PlanetSim and PeerSim are good examples of the Callback Hell since they require complex callback handlers and message programming. They had to break the elegant Chord RPCs into different code fragments clearly showing our Continuation Complexity Problem.

Macedon is a different approach that uses a DSL and state machines to simplify message handling. But again, it resulted in a complex code that is far from the simplicity of Chord sequential code using RPCs.

Before comparing the code, it is important to outline that the different versions are not implementing the Chord algorithm exactly as stated in the original article. For example, Macedon only implements a successor list of size 2, and their *fix_finger* protocol is using the simpler *update_others* variant that is not recommended for real settings.

Regarding PeerSim, it is important to outline that their implementation is not completely event-based because they also use object invocation shortcuts to simplify the code. In this line, Nodes access the *getProtocol()* method that provides a clone of the desired node.

Finally, PlanetSim provides a fully event-oriented implementation of Chord, but the implementation of the successor list does not consider all possible cases and errors in the protocol.



(a) Lines of Code

(b) Ciclomatic Complexity Average and Number of Classes

(c) Most Complex Method and Number of Methods

Fig. 4: Chord Evaluation

As we can see in the Figure 4 our approach (PyActive, JPyActive) is providing the simplest solution. Note that we also implemented a Java version (JPyactive) to be fair in the comparison with other Java-based solutions like PlanetSim or Peersim.

Our implementation has less LoC. Furthermore, our implementation is simpler and easier to understand that any of the presented alternatives. Our object oriented model is straightforward and it does not need additional understanding of messages, handlers and states. Macedon, PeerSim, and PlanetSim will require an understanding of messages and transitions that is more intricate that the simplicity of sequence diagrams in an object oriented design.

It is worth comparing the different approaches in code complexity (i.e., Cyclomatic Complexity). Again, our implementation is beating the other proposals in

the overall complexity and in the most complex method. One important reason is that our model avoids large message handling conditionals because messages are cleanly mapped to methods. Furthermore, synchronous calls that return results are naturally mapped to method invocations, whereas event-based approaches must split these invocations in requests and responses.

As we can see, even accepting that the Python language produces less lines of code compared to Java (e.g., PeerSim or PlanetSim), we have demonstrated that our additional reduction in terms of complexity and LoC is very meaningful. There are two main reasons: continuation complexity and message handling complexity. The continuation complexity is reduced in our model thanks to synchronous calls masked by proxies. The message handling complexity is reduced by the transparent mapping of messages to method calls in the Active Object pattern.

In particular, four methods in the Chord original algorithm present Continuation Complexity: *find_predecessor*, *fix_finger*, *stabilize*, and *join*. All of them should be changed in an asynchronous programming model, and thus requiring different code fragments (request, response, and timeout) for RPCs. In our case, we implemented Chord as a slight modification of Chord's original OOP code. Just by annotating the remote abstractions in the class, we can run Chord in an Actor library that also permits remote Actors in a transparent way.

On the contrary, the rest of the implementations (Macedon, PlanetSim, PeerSim) suffer from the continuation complexity problem in different degrees. Each takes a different strategy but all of them need to break every RPC in two code fragments (request and response). Regarding Timeouts, some declare a timeout function for every method, or they can reuse the timeout handler for all methods. So for example, Macedon is breaking *find_predecessor* in three code fragments (request, reply, timeout), but they are duplicated for different states of the protocol (joined, joining). Macedon does not add another function for the iteration but the additional code is found inside the response handler. PlanetSim and PeerSim also include handlers for requests, responses and timeouts inside their code implementing implicit continuations.

6 Conclusions

We identified the Continuation Complexity Problem that occurs when synchronous invocations (RPCs) must be converted to asynchronous events, handlers and futures. The produced code is difficult to read and maintain, and it breaks with the object oriented paradigm.

To cope with the Continuation Complexity Problem, we introduced a novel invocation abstraction for Object Actors (i.e., parallel). Our parallel abstraction supports concurrent interleavings of method executions within a single Actor. We mainly allow the main Actor thread to continue processing incoming method requests while the parallel thread is blocked waiting for a response in a remote Actor due to a synchronous invocation. This approach considerably simplifies the continuation complexity problem for distributed systems requiring non-blocking synchronous invocations.

Finally, we demonstrated with well-known algorithm (e.g, Chord) that our resulting code has less lines of code, less Cyclomatic Complexity, and that it is

more expressive than other event-based alternatives (i.e., PlanetSim, PeerSim, and Macedon). We believe that distributed systems can be considerably simplified using object oriented methodologies like the ones proposed in this article.

Our prototype implementation of PyActive can be downloaded from `https://github.com/cloudspaces/pyactive`, under a LGPL license. This implementation includes clarifying code examples and tutorials.

7 Acknowledgments

Special thanks to Douglas C. Schmidt for his helpful comments about the Active Object pattern.

This work has been partially funded by the EU in the context of the project *CloudSpaces* (FP7-317555) and by the Spanish Ministerio de Ciencia e Innovación in the project *Cloud Services and Community Clouds* (TIN2013-47245-C2-2-R)

References

1. .NET platform Async and Await abstractions, `http://msdn.microsoft.com/en-us/library/hh191443.aspx`
2. Agha, G.: Actors: A model of concurrent computation in distributed systems. In: The MIT Press series in artificial intelligence. USA (1986)
3. Bainomugisha, E., Carreton, A.L., Cutsem, T.v., Mostinckx, S., Meuter, W.d.: A survey on reactive programming. ACM Comput. Surv. 45(4), 52:1–52:34 (Aug 2013), `http://doi.acm.org/10.1145/2501654.2501666`
4. Edwards, J.: Coherent reaction. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. pp. 925–932. ACM (2009)
5. García-López, P., Pairot, C., Mondéjar, R., Pujol, J., Tejedor, H., Rallo, R.: Planetsim: A new overlay network simulation framework. In: Software engineering and middleware, pp. 123–136. Springer (2005)
6. Lin, Y., Radoi, C., Dig, D.: Retrofitting concurrency for android applications through refactoring. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 341–352. FSE 2014, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2635868.2635903`
7. Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09). pp. 99–100. Seattle, WA (Sep 2009)
8. Okur, S., Hartveld, D.L., Dig, D., Deursen, A.v.: A study and toolkit for asynchronous programming in c#. In: Proceedings of the 36th International Conference on Software Engineering. pp. 1117–1127. ICSE 2014, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2568225.2568309`
9. Reynolds, J.: The discoveries of continuations. In: Lisp and Symbolic Computation - Special issue on continuations- Volume 6 Issue 3-4, 6 (3/4): 233248. Hingham, MA, USA (1993)
10. Rodriguez, A., Killian, C., Bhat, S., Kostic, D., Vahdat, A.: Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. vol. 1, pp. 267–280 (2004)
11. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM Computer Communication Review. vol. 31, pp. 149–160. ACM (2001)