The Need for Language Support for Fault-tolerant Distributed Systems

Cezara Drăgoi, Thomas Henzinger, Damien Zufferey

The Need for Language Support for Fault-tolerant Distributed Systems

Cezara Drăgoi[1], Thomas A. Henzinger[*2], and Damien Zufferey[†3]

1 **INRIA, ENS, CNRS**
Paris, France
cezara.dragoi@inria.fr
2 **IST Austria**
Klosterneuburg, Austria
tah@ist.ac.at
3 **MIT CSAIL**
Cambridge, USA
zufferey@csail.mit.edu

Abstract

Fault-tolerant distributed algorithms play an important role in many critical/high-availability applications. These algorithms are notoriously difficult to implement correctly, due to asynchronous communication and the occurrence of faults, such as the network dropping messages or computers crashing. Nonetheless there is surprisingly little language and verification support to build distributed systems based on fault-tolerant algorithms. In this paper, we present some of the challenges that a designer has to overcome to implement a fault-tolerant distributed system. Then we review different models that have been proposed to reason about distributed algorithms and sketch how such a model can form the basis for a domain-specific programming language. Adopting a high-level programming model can simplify the programmer's life and make the code amenable to automated verification, while still compiling to efficiently executable code. We conclude by summarizing the current status of an ongoing language design and implementation project that is based on this idea.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Programming language, Fault-tolerant distributed algorithms, Automated verification

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Replication of data across multiple data centers allows applications to be available even in the case of failures, guaranteeing clients access to the data. For instance, state machine replication achieves fault-tolerance by cloning the application on several replicas, and ensuring that the same sequence of commands is executed on each replica using message passing. Building fault-tolerant software is challenging because designers have to face (1) asynchronous concurrency and (2) faults. Asynchrony means that replicas can interleave their actions in arbitrary ways and they communicate via message passing in a network that can delay

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).
† Damien Zufferey was supported by National Science Foundation grant 1138967.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–10

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

messages for an unbounded amount of time. Hardware faults can be transient, e.g., network partition, or permanent, e.g., crashes.

In a distributed system processes only have a limited view of the global state. However, one has to enforce some global property, e.g., ensuring consistency [1] between replicas by making sure that they execute a similar sequence of commands. Ensuring such a property typically requires solving a consensus problem: roughly, each process has an initial value, and all non-faulty processes have to agree on a unique decision value picked from the initial ones, even in the presence of faults and uncertainty in the timing of events. Strong consistency can be implemented by using consensus iteratively. Noteworthy examples from industry are Google Megastore, which uses Paxos [**?**] to replicate primary user data across data centers on every write, the Chubby lock service [**?**], Apache Zookeeper [**?**], which embeds a consensus protocol called Zookeeper Atomic Broadcast [**?**], or Microsoft Autopilot [**?**] that manages a data center.

State machine replication has received a lot of attention in both academia and industry. A fundamental result on distributed algorithms [**?**] shows that it is impossible to reach consensus in asynchronous systems where at least one process might crash. Consequently, a large number of algorithms have been developed [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**], each of them solving consensus under different assumptions on the type of faults, and the "degree of synchrony" of the system. Moreover, weaker versions of consensus that solvable under fewer network assumptions [**?**, **?**] have been proposed.

Consensus algorithms are difficult both from a theoretical and practical perspective. Because of the impossibility of solving consensus in asynchronous networks in the presence of faults, the existing consensus algorithms not only make various assumptions on the network for which they are designed but have also a complex flow of data. For example, the widely use Paxos algorithm [**?**] was considered hard to understand. A few years after its first publication Lamport wrote "Paxos Made Simple" [**?**] to re-explain the algorithm in simpler terms. However, the challenge of understanding it still remained. As an example, 2014 saw the publication of the raft algorithm [**?**], which purpose is to be simpler to understand than Paxos.

From a practical perspective implementing fault-tolerant distributed algorithms is challenging because the programmer has to use clocks, timing constrains, and complex network programming using low-level constructs, e.g., sockets, event-handlers. High-level programming languages like Erlang, or in general library based approaches, offer limited the possibility to reason about faults. Moreover, these algorithms are rarely implemented as theoretically defined, but modified to fit the constraints of the system in which they are incorporated. A group of engineers working at Google wrote "Paxos Made Live" [**?**] to describe some of the problems encountered while building a system based on Paxos and possible solutions for them. They [**?**, Section 9] emphasizes the following:

- "There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol."
- "The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms."

[1] Both strong or weak consistency are global properties.

- "The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems."

Despite the importance of fault-tolerant distributed algorithms, there are no automated verification techniques that can deal with the complexity of an implementation of an algorithm like Paxos. A few algorithms have been mechanically proved correct using interactive theorem provers, like Isabelle [**?**]. However these proofs required substantial effort and machine-checked proof of correctness for asynchronous implementations would demand even more expert time. For automated verification methods, the main difficulties are the unbounded number of replicas, the complex control structure using event-handlers, sockets, etc., and the complex data structures, e.g. unbounded buffers.

We think that the difficulty does not only come from the algorithms but from the way we think about distributed systems. Therefore, we are interested in finding an appropriate programming model for fault-tolerant distributed algorithms, that increases the confidence we have in applications based on state machine replication.

Our goal is to develop a domain-specific language for fault-tolerant distributed systems. The language must

1. be high-level to help the programmer focus on the algorithmic questions rather than spending time fiddling with low-level network and timer code;
2. compile into working, efficient systems that preserve the important properties of high-level algorithms;
3. use a programming model amenable to automated testing and verification.

Even though some of the difficulty of implementing fault-tolerant algorithms were addressed in the distributed algorithms community, there isn't a widely accepted programming language for fault-tolerant algorithms. Therefore, we think that this class of systems is relevant also for the programming language community. If we look at the past few years of programming language design and implementation, as well as software verification, i.e., POPL, PLDI, OOPSLA, and CAV, we find that only a handful of papers deal with systems where faults can occur. We believe that one should explore the advances in formal methods to increase the confidence in fault-tolerant applications. The standard formal verification approach considers the asynchronous code implemented in general programming language like C or Java, which is known for being notoriously hard. Instead we propose use alternative view over replicated systems, leading to a new programming model that eases the programmers' task and is amenable to automated verification.

In Section 2, we give an example of a fault-tolerant algorithm and explain some of the challenges in implementing and verifying it. In Section 3, we review existing work in the domain of programming languages for message-passing concurrency, models of computation for distributed systems, and the automated verification of distributed algorithms. In Section 4, we highlight several promising research directions that combine ideas across different fields to achieve the goals stated above. This section also summarizes the status of our ongoing project on this topic.

2 Example

Figure 1 shows a fault-tolerant algorithm [**?**] that solves the consensus. The algorithm is given in the formulation of [**?**], which is a round-based model: all processes execute the same code in lock-step manner. Each process knows initially a (potentially) different value for the variable `x`. The algorithm first establishes a majority of processes agreeing on the same value

```
Repeat:
    Send:     send x to all processes
    Update:   if received more than 2n/3 messages then
                  x := the smallest most often received value
                  if more than 2n/3 received values are equal to x then
                      decide(x)
```

Figure 1 A round based algorithm that solves Consensus.

for `x`, then witnesses that majority, and finally decides that the value of `x` is the one the majority agreed on. This particular algorithm needs four replicas to tolerate one crash, i.e., $n > 3f$, where n is the total number of replicas and f is the number of faulty processes.

Distributed algorithms are typically published in pseudo-code, or, in the best case, a formal but not executable specification [**?**]. For the implementation, the programmer has to make many decisions that influence the final system in a way that is hard to predict a priori. The first decision is to fix the fault model. The algorithm in Figure 1 tolerates only *benign* faults, i.e., messages can be lost but not modified. Allowing a *byzantine* attacker, which corrupts messages, requires using a modified algorithm [**?**]. After deciding on a benign faults, one still needs to decide whether to consider *crash-stop* or *crash-recovery*. Crash-stop means that a fault causes a replica to stop taking any steps. On the other hand, in a crash-recovery model faults are transient and a replica can rejoin system after a crash. Implementing an algorithm is a crash-recovery fault model requires a *stable memory*, i.e., writing to a permanent storage every change of the local state, or using a dedicated *recovery procedure* to be executed by a replica before rejoining the system after a crash. Next there is the question if the communication is reliable or if the network can *drop messages*. This will influence how long the processes wait to receive messages and whether a missing message is interpreted as a crash. All these choices influence the implementation: what kind of *failure-detector* to implement, how much additional information has to be added to messages, etc. A failure-detector is an application which keeps track of the processes in the system to know whether they have failed. In its simplest incarnation, a failure detector periodically sends heartbeat messages to the others processes in the system. Then, for performance reasons, these messages can be piggybacked on top of the protocol's messages to avoid the overhead of sending additional messages. Such small optimizations quickly add up, perturb the normal flow of the protocol, and lead to a complex, hard-to-maintain codebase. Finally, when the algorithm is implemented, there is the question of how to test or verify it. Asynchronous systems are notoriously known hard to test, debug, and prove correct.

When defining the semantics of a programming language, it is simple and natural to assume an asynchronous model, but most programmers and algorithm designers have a notion of time when they write code. The round structure of the algorithm in Fig. 1 can be seen as an abstract notion of time. From a more practical perspective, in the context of the programming language P [**?**], we had discussions about adding constructs to deal with time. P is a domain specific language for communication state machine which is used to program device drivers. A common coding pattern for drivers is to send a request, initialize a timer, and wait for either the response or the timer to fire. If the response comes first, then the timer needs to be canceled. This pattern introduces boilerplate that obscures the code. The proposal was to add a new type of transition, `Timeout(duration)`, that would automatically initialize/cancel the timer. However, in the end we decided against adding timeouts to keep the number of primitives in the language small.

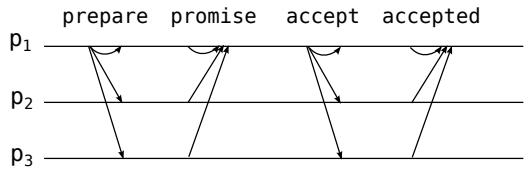

Figure 2 Paxos decomposed in four communication-closed rounds. Process p_1 has the role of proposer, acceptor, and learner. Processces p_2 and p_3 are acceptors.

3 Prior Work

Programming abstractions for message-passing systems. Formalisms such as the π-calculus [**?**, **?**] and I/O-automata [**?**] have mostly been used to give a formal semantics to message-passing systems and to analyze them, but not as programming model. Some of these models have been extended with time, but they are all natively asynchronous. Furthermore, faults are not part of these models. To deal with faults one has to modeled them on top of the provided primitives, increasing dramatically the complexity of the final system.

The Actor model [**?**] is probably the most successful high-level abstraction for message-passing systems. CSP [**?**] has also seen some success with programming languages like Occam, Ada, etc., and CSP-style rendezvous concurrency is present in many other languages. Programming languages like Erlang [**?**] are based on the actor model and many programming languages have libraries implementing the actor model. Erlang also has dedicated library support to help the programmers write fault-tolerant applications. Finally, at the lowest level, we find the POSIX sockets. Disturbingly enough, they are probably still the most common programming abstraction for distributed systems.

In this paper, we focus on fault-tolerance though replication, considering algorithms that make uniform and non-uniform assumptions on the replicas. The programming models designed to implement replicated computation and storage [**?**, **?**], typically take a centralized approach, based on a trusted client that interacts with all replicas. For example, the Erlang/OTP framework [**?**] provides fault tolerance using supervision. An application is hierarchically structured in a tree where the leaf nodes are workers and the inner nodes are supervisors. The supervisors watch over workers and other supervisors. They are responsible for starting and shutting them down in the normal course of action and, in presence of failure, responding to a fault in their children, usually by restarting the process. However, there are implementations of Paxos in Erlang designed to integrate with the OTP library and systems like Zookeeper, that combine both approaches, a uniformly replicated core that manages a large collection of workers. We are mostly interested in replicated systems that do not rely on supervision, and make uniform assumptions on the replicas, or non-uniform assumptions that are weaker than supervision.

Models of computation for distributed algorithms. Distributed algorithms and systems are mature research fields. This is by no means a complete overview of these fields, but we highlight a few ideas that have not been adopted by the language and verification communities and that we believe are good candidates to form the basis of new programming abstractions. An important idea to simplify the formulation of distributed algorithms is scoping the communication. *Communication-closed rounds* [**?**] encapsulate all communication within rounds. Conceptually, processes operate in lock-step: in each round they send messages, receive messages, and update their local state depending on the local state at

the beginning of the round and the received messages. Many distributed algorithms can be formulated naturally in a round-based model [?, ?, ?, ?, ?, ?, ?]. Figure 2 shows how this can be done for the Paxos algorithm. Organizing the computation in rounds looks restrictive and originally was limited to synchronous systems. However, rounds can be generalized to partial synchrony [?]. Moreover, failure detectors [?] have been proposed to structure the detection and handling of failures. The combination of these ideas gives rise to round-based computational models that can *uniformly model synchronous and asynchronous systems* [?, ?, ?], e.g., Charron-Bost and Schiper [?, Table 1] showed how to do this for common fault models. In this view, an asynchronous, or faulty, system is a synchronous system with an adversarial environment that can delay, drop, or modify messages.

Verification of distributed algorithms. From the verification perspective, distributed algorithms are a very challenging class of systems because of several sources of unboundedness: messages come from unbounded domains, the number of processes is a parameter, and channels may also be unbounded. Indeed, most of the verification problems are undecidable for parameterized systems [?]. As alternative, model checking is done by instantiating the algorithms using a finite, usually small, number of processes [?, ?, ?, ?]. This approach works for finding bugs, but cannot prove an algorithm correct. Furthermore, the reduction to finite-state systems cannot model faithfully all types of faults, even for a finite number of processes. For instance, if we consider byzantine systems, a faulty process can send arbitrary messages that may not even be defined in the protocol.

The channels also make the verification problem hard. Unbounded FIFO channels causes undecidability even for two processes [?]. Making the channels lossy and fixing the number of processes makes the problem decidable [?], with a non-primitive recursive complexity [?]. Weaker channel models are usually at least EXPSPACE-hard for verification. A pervasive pattern in fault-tolerant systems is counting messages up to a threshold that depends on the number of processes in the system, e.g., $2n/3$ in Figure 1. Unfortunately, this means that these systems do not fall in the class of well-structured transition systems [?], because they do not satisfy the required monotonicity conditions. Well-structured transition systems are arguably the most general class of infinite-state systems for which verification questions are decidable [?, ?].

As a result, full formal proofs of correctness of fault-tolerant algorithms are often based on interactive proof assistants [?, ?, ?]. However, they require a substantial effort. For instance, a proof of correctness of the Paxos algorithm is about 1500 lines long [?]. Recently, we have seen the use of interactive proof assistants to program real systems [?].

4 Project Outline

We started to look at fault-tolerant distributed systems with the goal of verifying them [?]. We quickly realized that the situation is very grim if we look at them through classical, asynchronous language and verification models. Using the round-based models from the distributed algorithms community we can achieve a much higher degree of automation in the verification. After all, these models were developed as simplifications for manually reasoning about asynchronous systems, e.g., for expressing the FLP [?] proof "on the back of a napkin"[2]. Structuring the computation in communication-closed rounds is an intuitive programming

[2] From the talk "Better late (40 years late!) than never: Monday morning quarterbacking the coordinated-attack problem" by Eli Gafni at Yale University on 2014-10-09.

model as for each round, the programmer only has to give two methods, one to send messages, and the other to update the local state upon reception of messages. To cover asynchronous and faulty executions, one only needs to accept that not every message gets delivered. From the verification perspective, this model (1) removes the problem of interleavings, because computations look synchronous: all processes proceed in lock-step, and (2) allows removing all channels from the state of the system by looking at the system's invariant at the boundary between rounds.

We believe that like the verification community, the programming language community has neglected to address the question of faulty systems and has been unproductively conservative in modeling them. The opportunity for the language community is to develop better abstractions for programming and reasoning about faulty distributed systems. As in the case of verification, a good place to get inspiration from are the models of the distributed algorithms community and our goal is to adapt their ideas to the needs of programming languages. However, this is not trivial: models such as [**?**] have been developed to be theoretically well-behaved and to simplify reasoning about faulty systems, but little has been done to compile and execute them. Also, if we adopt such a seemingly synchronous abstraction to reason about asynchronous systems, we must tell the programmer which properties of the synchronous abstraction transfer to the actual asynchronous system. Indeed, not every property is preserved. Yet we postulate that a round-based programming abstraction is still superior to the current state of affairs, where implementations need to handle faults in the asynchronous model. In particular, control-state reachability, a classical safety property for concurrent systems, is preserved by asynchrony due to its locality. Also most properties that can be checked using well-structured transition systems are preserved. In fact, it seems that most "reasonable" properties transfer from a synchronous abstraction to asynchronous executions [**?**, **?**], and that this can be formalized.

The heard-of model [?]. The base primitive of our language is communication-closed rounds [**?**] and we use the heard-of (HO) formalization. Conceptually, processes operate in lock-step, and a distributed algorithms consist of rules that determine the new state of a process depending on the state at the beginning of the round and the messages received by the process in the current round. During a round, between the send and reception of the message, an adversarial environment can choose to drop messages. The impact of the environment is captured by the heard-of sets: for a process p its heard-of set, denoted $HO(p)$, contains the processes from which p may receive messages from in a given round. To model different kinds of network assumptions, the power of the environment is restricted by constraints on the HO set, given in linear temporal logic [**?**]. For instance, in a synchronous system without faults the environment respects $\Box(\forall p, q.\, p \in HO(q))$, which means q hears from every p, i.e., that every message is delivered. On the other hand, an asynchronous system with arbitrary faults does not have impose any restriction on the HO set. The environment can drop an arbitrary number of messages. The HO model can express intermediate failure models between these two extremes [**?**, Table 1]. For instance, a partially synchronous system with eventual reliable links and at most f crashes is $\Diamond(\exists S.\, |S| > n - f \wedge \forall p.\, |HO(p)| = S)$.

The language. We are currently working on building a complete infrastructure for implementing fault-tolerant distributed systems. The front-end consists of a domain-specific language and the corresponding specification logic. The DSL is based on the "heard-of model" [**?**], in which algorithms are structured in communication-closed rounds and have a synchronous semantics. The specification logic is an extension of [**?**] which is suitable

for expressing many agreement properties such as consensus. At the back-end, we have an automated verifier and compiler+runtime. The verifier checks the algorithms against the specification in the synchronous semantics and the compiler+runtime generates code for asynchronous systems such that a process running the algorithms cannot distinguish between the synchronous semantics and an asynchronous execution. In our system, the programmer writes the core of the application, e.g., the fault-tolerant distributed algorithm, in the high-level DSL. As the round abstraction is suitable for designing distributed algorithms, but not necessarily for the rest of the system, the algorithm interfaces with other software components like a service, e.g., a consensus service. For the verification, we require the programmer to provide an inductive invariant that captures the correctness idea behind the algorithm. The verifier generates Hoare-style verification conditions and tries to discharge them using an SMT solver. Finally, the compiler+runtime fully automatically generate asynchronous distributed code that is guaranteed to preserve the local fault-handling properties that were established for the algorithm.

Limitations. Round models also have their downsides. First, they impose a regular control flow. This complicates the encoding different computation paths, e.g., paths the try to speculate a potentially reliable system to reach a decision faster, or recovery paths to be taken after a crash. Protocols, with complex recovery paths should be encoded as two different algorithms: one for the normal scenario, and one for the recovery. We have started to investigating distributed systems build using only the synchronous parallel composition of the participants. Richer forms of composition could support modular composition of different algorithms, e.g., making a new algorithm by running many copies of another algorithm [**?**]. Finally, the runtime is dependent on the considered fault-model. A round model is a weak form of clock synchronization which requires some environment assumptions. One can guarantee the progress of an implementation of a round-based model with benign faults, by assuming partial synchrony. However, implementing a round model in the byzantine setting is roughly possible with at most $n/3$ byzantine processes, where n is the number of replicas. Therefore, using this model does not make sense to implement an algorithm whose assumptions are weaker than those imposed by the implementation of the round model.

Prototype implementation. We are currently working on a prototype implementation of DSL in the SCALA programming language. Figure 3 shows the algorithm from Figure 1 in our DSL. The language is designed as a shallow embedding. The parts that require code generation, e.g. resource and state encapsulation, are handled using macros [**?**] which manipulate the SCALA syntax tree, and the serialization is made transparent using the pickling library [**?**].

To write an algorithm in this language, a programmer needs to write the program in a sequence of `Round`. Each round has a `send` and a `update` function. Intuitively, all the processes in the system, when executing a given round execute `send` followed by `update`. A runtime is responsible for delivering messages and detecting faults. Therefore, not every message might be received. Furthermore, the runtime guarantees that messages are only visible within the round in which they are sent, even if the system is asynchronous and processes progress at different speed. Late messages are discarded and late processes will try to catch up after receiving message for a "future" rounds.

To verify the algorithm, we assume that the user gives us and the specification and inductive invariants that describe the global state of the system. For our running example,

```
class OTR extends Algorithm[ConsensusIO] {

  val x = new LocalVariable[Int](0)
  val decision = new LocalVariable[Int](-1)    //used by the verification
  val decided = new LocalVariable[Boolean](false) //used by the verification
  val callback = new LocalVariable[ConsensusIO](null)

  def process = new Process[ConsensusIO]{

    def init(io: ConsensusIO) {
      callback <~ io
      x <~ io.initialValue
      decided <~ false
    }

    val rounds = Array[Round](
      new Round{
        type A = Int //guarantees that the types of send and receive match

        //minimal most often received value
        def mmor(mailbox: Set[(Int, ProcessID)]): Int = ...

        def send(): Set[(Int, ProcessID)] = broadcast(x)

        def update(mailbox: Set[(Int, ProcessID)]) {
          if (mailbox.size > 2*n/3) {
            x <~ mmor(mailbox)
            if (mailbox.filter(msg => msg._1 == x).size > 2*n/3) {
              callback.decide(x)
              decided <~ true
              decision <~ x
              terminate
          } }
        }
} }  }
```

Figure 3 The on-third-rule algorithm implemented in our DSL

the agreement property is

$$\forall i, j.\ decided(i) \wedge decided(j) \Rightarrow decision(i) = decision(j)$$

and this property is enforced by the following invariant:

$$\forall i.\ \neg decided(i) \quad \vee \quad \exists v.\ |\{i.\, x(i) = v\}| > 2n/3 \wedge \forall i.\ decided(i) \Rightarrow decision(i) = v.$$

The invariant states that either no value has been decided, or there exists a majority of replicas formed of more than $2n/3$ replicas, that agree on the value of `x`. Notice that the invariant is still simple due to the fact that we use a state invariant that holds only at the boundary between rounds. Therefore, the invariant does not need to refer to messages or the state of communication channels. Communication-closed rounds are key in enabling us to apply automated verification on this class of algorithms. An additional benefit of this model is that it enables us to reason about liveness. Any computation terminates, if there are two rounds in which all processes receive the messages send by a majority P of cardinality greater than $2n/3$. More precisely, in each of these two rounds the values of the HO-sets assigned by the environment satisfy the property

$$\exists P.\ \forall p.\ HO(p) = P \wedge |P| > 2n/3,$$

where P is a set of processed and p denotes any process in the system.

The verification procedure [**?**] is roughly based on computing the Venn regions for the sets occurring in the formula, e.g., $\{i.\, x(i) = v\}, HO(p), mailbox(p)$, reasoning about the cardinality constraints, e.g., $|mailbox(p)| > 2n/3$, and propagating the global conditions appearing in the definition of set comprehensions to constraints on the values of replica's local variables, e.g., $x(p) = v$ is inferred from the predicate defining the set comprehension in the invariant.

To summaries, the project aims to find a programming model that strikes a balance between program design and program analysis. More precisely, we are interested in a programming model based on high-level concepts that offer programmers freedom in reasoning about faults and asynchrony. Its formal semantics should enable the development of automated verification tools that establish the correctness of the implementations. We intend to start with one of the most studied classes of algorithms in the literature: consensus algorithms [**?**, **?**] and their weaker forms, e.g., k-set agreement [**?**] or lattice agreement [**?**].

Acknowledgements. We thank Josef Widder for the fruitful discussions on this subject, Heather Miller and Philipp Haller for their help with the pickling library, and the anonymous reviewers for their comments.

5 Conclusion

Building correct software in the absence of faults is already a challenging task. Now, doing so when faults are integral part of the system is an even greater challenge. Addressing this will requires adopting new programming models that natively include faults and/or give ways to detect and handle them. Ideally, we should aim for failure friendly programming abstractions where faults are naturally integrate without obscuring the program's logic. More realistically, we can start by identifying the features that makes those system unnecessarily complex and find better alternatives. We briefly sketched how models developed in the distributed algorithms community also solves issues that, in the PL community, make those system hard to implement and severely limit the scope of automated verification for distributed systems.