On the Design of Distributed Programming Models

Christopher S. Meiklejohn*
Université catholique de Louvain
Louvain-la-Neuve, Belgium
INESC-ID & Instituto Superior Técnico, Universidade de Lisboa
Lisboa, Portugal
christopher.meiklejohn@uclouvain.be

ABSTRACT

Programming large-scale distributed applications requires new abstractions and models to be done well. We demonstrate that these models are possible.

Following from both the FLP result and CAP theorem, we show that concurrent programming models are necessary, but not sufficient, in the construction of large-scale distributed systems because of the problem of failure and network partitions: languages need to be able to capture and encode the tradeoffs between consistency and availability.

We present two programming models, Lasp and Austere, each of which makes a strong tradeoff with respects to the CAP theorem. These two models outline the bounds of distributed model design: strictly AP or strictly CP. We argue that all possible distributed programming models must come from this design space, and present one practical design that allows declarative specification of consistency tradeoffs, called Spry.

ACM Reference format:
Christopher S. Meiklejohn. 2017. On the Design of Distributed Programming Models. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Languages for building large-scale distributed applications experienced a Golden Age in the 1980s with innovations in networking and the invention of the Internet. These languages tried to ease the development of distributed applications, influenced by the growing requirements of increased computing resources, larger storage capacity, and the desired high-availability and fault-tolerance of applications.

Two of the most widely known from the era, Argus [23] and Emerald [4] each took a different approach to solving the problem, and the abstractions that each of these languages provided aimed to simplify the creation of distributed applications, to reduce uncertainty when dealing with unreliable networks, and to alleviate the burden of dealing with low-level details related to a dynamic network topology; Emerald focused heavily on object mobility and Argus focused on atomic transactions across multiple objects each of which may reside on different machines.

As it stands, these languages never saw much adoption[1] and most of the large-scale distributed applications that exist today have been built with sequential or concurrent programming languages such as Go, Rust, C/C++, and Java. These languages have taken a library approach to distribution, adopting many ideas from languages such as Emerald and Argus. We can highlight two examples: first, the concept of *promises* from Argus, is now a standard mechanism relied upon when issuing an asynchronous request [13, 24, 30]; second, from Emerald, the concept of a directory service that maintains the current location of mobile processes [6].

Distributed programming today has become "the new normal." Nowadays, whether you are building a mobile or a rich web application, developers are increasingly trying to provide a near-native experience [8], where users of these applications feel as if the application is running directly on their machine. To achieve this, shared state is usually replicated on user's devices, locally mutated and then these changes are periodically propagated back to a server. In these scenarios, maintaining and reasoning about consistency can become increasingly challenging if the application is to stay available when the server cannot be reached. **Therefore, it is now paramount that we have tools to ease the creation of distributed applications.**

We argue that the reason that these previous attempts at building languages for large-scale distributed systems have failed to see adoption is that they fail to capture the requirements of today's applications and meet the requirements of today's application developers. For instance, if an application must operate offline, a language should have primitives for managing consistency and conflict resolution; similarly, if a language is to adhere to a strict latency bound for each distributed operation, the language should have primitives for expressing these constraints.

In this paper, we relate these real-world application requirements to the CAP theorem [5, 15], showing that a distributed application must sacrifice consistency if it wishes to remain available under network partitions. We demonstrate that there are several design points for programming models that could, and do, exist within the bounds of the CAP theorem, with two examples that declaratively specify application-level distribution requirements.

*Partially funded by the SyncFree Project in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement nº 609551 and by the Erasmus Mundus Doctorate Programme under grant agreement nº 2012-0030.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] One notable exception here is the Distributed Erlang [31] extension for the Erlang programming model, later adopted by Haskell [12].

2 SEQUENTIAL AND CONCURRENT PROGRAMMING

We first explore the challenges when moving from sequential programming to concurrent programming to highlight the challenges of concurrency before looking at the challenges of partial failure.

2.1 Sequential Programming

Most of the programming models that have widespread adoption today are designed in von Neumann style[2]: computation revolves around mutable storage locations whose values vary with time, aptly called variables, and control statements are used to order assignment statements that mutate these storage locations. Programs are seen to progress in a particular order, with a single thread of execution, and terminate when they reach the end of the ordered list of statements.

2.2 Concurrent Programming

Concurrent programming extends sequential programming with multiple sequential threads of execution: this is done to leverage all available performance of the computer by allowing multiple tasks to execute at the same time. Each of these threads of execution could be executing in parallel, if multiple processors happened to be available, or a single processor could be alternating control between each of the threads of execution, giving a certain amount of execution time to each of the threads.

Concurrent programming is difficult. In the von Neumann model where shared memory locations are mutated by the sequential threads of execution, care must be taken to prevent uncontrolled access to these memory locations, as this may sacrifice correctness of the concurrent program. For instance, consider the case where two sequential threads of execution, executing in parallel, happen to read the same location and write back the location incremented by 1. It is trivial to observe that without controlled access to the shared memory location, both threads could increment the location to 2; effectively "losing" one of the valid updates to the counter.

Originally described by Dijkstra [11], this "mutual exclusion" problem is the fundamental problem of concurrent computation with shared memory. How can we provide a mechanism that allows correct programming where multiple sequential threads of execution can mutate memory that is visible by all nodes of the system. Dijkstra innovated many techniques in this area, but the most famous technique he introduced was that of the "mutex" or "mutual exclusion."

Mutual exclusion is the process of acquiring an exclusive lock to a shared memory location to ensure safe access. Returning to our previous example, if each sequential thread of execution was to acquire an exclusive lock to the memory location before reading and updating the value, we no longer have to worry about the correctness of the application. However, mutual exclusion can be difficult to get right when multiple locks are required, if they are not handled correctly to avoid deadlock.

But, how is the programmer to reason about whether their concurrent application has been programmed correctly? Given multiple threads of execution, and a nondeterministic scheduler, there exists an exponential number of possible schedules, or executions, the program may take, where programmers desire that each of these executions result in the same value, regardless of schedule.

Therefore, most programmers desire a correspondence commonly referred to as confluence. Confluence states simply, that the evaluation order of a program, does not impact the outcome of the program. In terms of the correspondence, programmers ideally can write code in a sequential manner, that can be executed concurrently, and possibly in parallel, and any of the possible schedules that it may take when executed, results in the same outcome of the sequential execution.

From a formal perspective, we have several correctness criteria for expressing whether a concurrent execution was correct. For instance, sequential consistency [16, 19] is a correctness criteria for a concurrent program that states that the execution and mutation of shared memory reflects the program order in the textual specification of the program; linearizability [17] states that a concurrent execution followed the real time order of shared memory accesses and strengthens the guarantees of sequential consistency.

3 DISTRIBUTED PROGRAMMING

Distributed programming, while superficially believed to be an extension of concurrent programming, has its own fundamental challenges that must be overcome. One of the major challenges in distributed programming is partial failure.

3.1 The Reasons For Distribution

Distributed programming extends concurrent programming even further. Given a program that's already concurrent in it's execution, programmers distribute the sequential threads of execution across multiple machines[3] that are communicating on a network. Programmers do this for several reasons:

- **Working set.** The working data set or problem programmers are trying to solve will take too long to execute on a single machine or fit on a single machine in memory, and therefore programmers need to distribute the work across multiple machines.
- **Data replication.** Programmers need data replication, to ensure that a failure of one machine does not cause the failure of our entire application.
- **Inherent distribution.** Programmers application's are inherently distributed; for example, a client application living on a mobile device being serviced by a server application living at a data center.

What programmers have learned from concurrent programming is that accesses to shared memory should be controlled: programmers may be tempted to use the techniques of concurrent programming, such as mutexes, monitors [18], and semaphores, to control access to shared memory and perform principled, safe, mutation.

3.2 The Challenges of Distribution

On the surface, it appears that distribution is just an extension of concurrent programming: we have taken applications that relied

[2]Functional and logic programming remain notable exceptions here, although their influence is minimal in comparison.

[3]We acknowledge that newer designs of machines are beginning to appear as distributed systems, but in this example we specifically focus on the case where one or more asynchronous networks sit between the each of the processors.

on multiple threads of execution to work in concert to achieve a goal and only relocated the threads of execution to gain more performance and more computational resources. However, this point of view is fatally flawed.

As previously mentioned, the challenges of concurrent programming are the challenges of nondeterminism. The techniques pioneered by both Dijkstra and Hoare were mainly developed to ensure that nondeterminism in scheduling did not result in nondeterminism in program output. Normally, we do not want the same application, with fixed inputs, to return different output values across multiple executions because the scheduler happened to schedule the threads in different orders.[4]

Distribution is fundamentally different from concurrent programming: machines that communicate on a network may be, at times, unreachable, completely failed, unable to answer a request, or, in the worst case, permanently destroyed. Therefore, it should follow that our existing tools are insufficient to solve the problems of distributed programming. We refer to these classes of failures in distributed programming as "partial failure": in an effort to make machines appear as a single, logical unit of computation, individual machines that make up the whole may independently fail.

Distributed systems researchers have realized this, and have identified the core problem of distributed system as the following: the agreement problem. The agreement problem takes two forms, duals of each other, mainly:

- **Leader election.** The process of selecting an active leader amongst a group of nodes to act as a sequencer or coordinator of operations for those nodes; and
- **Failure detection.** The process of detecting a node that has failed and can no longer act as a leader.

These problems, and the problems of locking, are only exacerbated by two fundamental impossibility results in distributed computing: the CAP theorem [15] and the FLP result [14].

The FLP result demonstrates that on a truly asynchronous system, agreement, when one process in the agreement process has failed, is impossible. To make this a bit more clear, when we can not determine how long a process will take to perform a step of computation, and we can not determine how long a message will take to arrive from a remote party on the network, there is no way to tell if the process is just delayed in responding or failed: we may have to wait an arbitrarily long amount of time.

FLP is solved in practice via randomized timeouts that introduce nondeterminism into the leader election process to prevent infinite elections. Algorithms that solve the agreement protocol, like the Raft consensus protocol [28] and the Paxos leader election algorithm [7, 20] take these timeouts into account and take measures to prevent a seemingly faulty leader from sacrificing the correctness of a distributed application.

The CAP theorem states another fundamental result in distributed programming. CAP states simply, that if we wish to maintain linearizability when working with replicated data, we must sacrifice the ability for our applications to service requests to that shared memory if we wish to remain operational when some of the processes in the system can not communicate with each other. Therefore, for distributed applications to be able to continue to operate when not all of the processes in the system are able to communicate – such as when developing large-scale mobile applications or even simple applications where state is cached in a web browser – we have to sacrifice safe access to shared memory.

Both CAP and FLP incentivize developers to avoid using replicated, shared state, if that state needs to be synchronized to ensure consistent access to it. Applications that rely on shared state are bound to have reduced availability, because they either need to wait for timeouts related to failure detection or for the cost of coordinating changes across multiple replicas of the shared state.

4 TWO EXTREMES IN THE DESIGN SPACE

While development of large-scale distributed applications with both sequential and concurrent programming models has been widely successful in industry, most of these successes have been supported by systems that close the gap between a language that has distribution as a first-class citizen, and a concurrent language where tools that solve both failure detection and the agreement problem are used to augment the language.

For programming models to be successful for large-scale distributed programming, they need to embrace the tradeoffs of both the FLP result and the CAP theorem. We believe that there exists a space, in what we refer to as the boundaries of the CAP theorem, where a set of programming models that take into account the tradeoffs of the CAP theorem, can exist and flourish as systems for building distributed applications.

We now demonstrate two extremes in the design space. First, Lasp, a programming model that sacrifices consistency[5] for availability. Second, Austere, a programming model that sacrifices availability for consistency. Both of these models sit at extreme sides of the spectrum proposed by the CAP theorem.

Both of these languages, and in fact all possible languages that can exist in this design space, share a common logical component: a data store (or, database of cells) that track the state of "variables." To ensure recency of these replicas, a propagation mechanism is used: where strong consistency is required, this protocol may be driven by a consensus protocol such as Raft [28] or Paxos [20]; where weaker consistency is required, either a naive or causal anti-entropy protocol [10] may suffice. Where the models differ is in there evaluation semantics. Application written in these models may choose to synchronize replicas before using a value in a computation or not, depending on whether the model prefers availability or consistency.

In terms of language semantics, each of these models can be seen as a concurrent λ-calculus with futures λ(fut) (like, Alice ML. [27]) The aforementioned data store is realized in this model as reference cells and futures are used to provide on data-driven synchronization: when strong consistency is required, synchronization occurs with all replicas and an atomic exchange is performed on the cell; when weaker consistency is allowed, a previous value can be used and the exchange performed asynchronously.

[4] That is, unless the program is specifically written to determine its output on scheduling, such as a multithreaded debugger.

[5] Big "C" consistency here, referring to the CAP result and linearizability.

4.1 Lasp

Lasp [25, 26] is a programming model designed as part of the SyncFree and LightKone EU projects [1, 2] focusing on synchronization-free programming of large-scale distributed applications. Lasp sits at one extreme of the CAP theorem: Lasp will sacrifice consistency in order to remain available.

Lasp's only data abstraction is the Conflict-Free Replicated Data Type (CRDT) [29]. A CRDT is a replicated abstract data type that has a well defined merge operation for deterministically combining the state of all replicas, and Lasp builds upon one specific variant of CRDTs: state-based CRDTs. CRDTs guarantee that once all messages are delivered to all replicas, all replicas will converge to the same result.

With state-based CRDTs[6], each data structure forms a bounded join semilattice, where the join operation computes the least-upper-bound for any two elements. While CRDTs come in a variety of flavors, like sets, counters, and flags (booleans), two main things must be keep in mind when specifying new CRDTs:

- CRDTs are replicated, and by that fact inherently concurrent. Therefore, when building a CRDT version of a set, the inventor of the CRDT must define semantics for all possible pairs of concurrent operations that do commute: for instance, concurrent additions of different elements commute, but concurrent additions and removals of the same element do not commute.[7]
- To ensure replica convergence with minimal coordination, it follows from the join-semilattice that the join operation compute a least-upper-bound: therefore, all operations on CRDTs must be associative, commutative, and idempotent.

Lasp is a programming model that allows developers to do basic functional programming with CRDTs, without requiring application developers to work directly with the bounded join-semilattice structures themselves: in Lasp, a developer sees a CRDT set as a sequential set. Given all of the data structures in Lasp are CRDTs themselves, the output of Lasp applications are also CRDTs that can be joined to combine their results.

Lasp never sacrifices availability: updates are always performed against a local replica and state is eventually merged with other replicas. Consistency is sacrificed: while replica convergence is ensured, it may take an arbitrarily long amount of time for convergence to be reached and updates may arrive in any order.

However, Lasp has several strong restrictions given the CRDT foundation that provides its availability: all operations must commute and all data structures must be able to be expressed as a bounded join-semilattice. This obviously rules out several common data structures, such one very important one: the list.

4.2 Austere

Austere is a programming model where all replicated, shared state is synchronized for every operation in the system. Austere sits at another extreme of the CAP theorem: Austere will sacrifice availability in order to preserve consistency.

[6]Herein referred to as just CRDTs.

[7]This is precisely why CRDT sets come in two flavors: versions that bias towards the removal winning under concurrent addition and removal, and versions that bias towards the addition winning under concurrent addition and removal.

Before any access or modification to replicated, shared state, Austere will contact all replicas using two-phase locking (2PL) [22] to ensure values are read without interleaving writes with communication, and two-phase commit (2PC) [3] to commit all modifications. In the event that a replica can not be contacted, the system will fail to make progress, ensuring a single system image: reducing distributed programming to a single sequential execution to ensure a consistent view across all replicas.

Compared to Lasp, Austere shares the common λ(fut)-calculus core; however, the gain of semantics in Austere is paid for by reduced availability.

5 THE "NEXT 700" DISTRIBUTED PROGRAMMING MODELS

The "next 700" [21] distributed programming models will set between the bounds presented by Austere and Lasp: extreme availability where consistency is sacrificed vs. extreme consistency where availability is sacrificed. (see Figure 1)

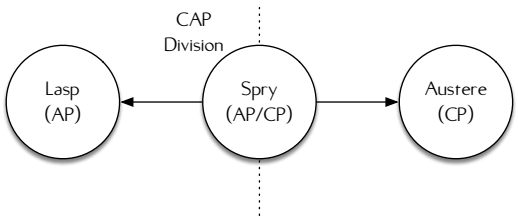

Figure 1: The design space of the "next 700" distributed programming models.

More practically, we believe that the most useful languages in this space will allow the developer to specifically trade-off between availability and consistency at the application-level. This is because the unreliability of the network, dynamicity of real networks, and production workloads require applications to remain flexible. These trade-offs should be specified declaratively, close to the application code. Application developers should not have to reason about transaction isolation levels or consistency models when writing application code.

We provide an example of one such language, Spry, that makes a trade-off between availability and consistency based on application-level semantics. We target this language for one use case in particular: Content Delivery Networks (CDNs), where application-level semantics can be used to declaratively specify Service Level Agreements (SLAs.)

5.1 Spry

Spry [9] is a programming model for building applications that want to tradeoff availability and consistency at varying points in application code to support application requirements.

We demonstrate two different types of tradeoffs that application developers might make in the same application. Consider the case of a Content Delivery Network (CDN), an extremely large-scale distributed application.

- **Availability for consistency.** In a CDN, the system tries to ensure that content that is older than a particular number of seconds will never be returned to the user. This is usually specified by the application developer explicitly, by checking the object's staleness and fetching the data from the origin before returning the response to the user.
- **Consistency for availability.** CDN's usually maintain partitioned inverted indexes that can be queried to search for a piece of content. Because nodes may become unavailable, or respond slowly because of high load, application developers may want to specify that a query returns results from a local cache if fetching the index over the network takes more than a particular amount of time. This is usually specified by the application developer explicitly, by setting a timer, attempting to retrieve the object over the network, reusing cached results if the latency bound can not be met, and returning the response to the user.

Application developers specify these constraints declaratively in Spry. If a replicated value should not be older than a particular number of milliseconds, developers can annotate these values with the bounded staleness requirements. If a replicated value should always be as fresh as it can be within a bound of a number of milliseconds, this can be specified as well. Similarly, these values can be tweaked while the application is running, allowing developers to adjust the system while it is operating, responding to failures or increased load.

6 CONCLUSION

We have seen that the move from sequential programming to concurrent programming was fairly straightforward: all that was required was a principled approach to state mutation through the use of techniques like locking to prevent values from being corrupted, which can lead to unsafe programs. However, the move to distributed programming is much more difficult because of the uncertainty that is inherent in distributed programming. For example: will this machine respond in time? Has this machine failed and is it able to respond?

Distributed programming is a different beast, and we need programming models for adapting accordingly. Can we come up with new abstractions and programming models that aid in expressing the application developers intent **declaratively?**

We believe that it is possible. We have shown you three different programming model designs that all make different tradeoffs when nodes become unavailable. Two of these, Lasp and Austere, provide the boundaries in model design that are aligned with the constraints of the CAP theorem. One of these, Spry, takes a declarative approach that puts the tradeoffs in the hands of the application developer. All three of these can cohabit the same underlying concurrent language.

ACKNOWLEDGEMENTS

We want to thank Zeeshan Lakhani, Justin Sheehy, Andrew J. Stone, and Zach Tellman for their feedback.

REFERENCES

[1] LightKone: Lightweight computation for networks at the edge. (????). European H2020 project to start in January 2017.
[2] SyncFree: Large-scale computation without synchronisation. https://syncfree.lip6.fr. (????). European FP7 project 2013–2016.
[3] Yousef J Al-Houmaily and George Samaras. 2009. Two-phase commit. In *Encyclopedia of Database Systems*. Springer, 3204–3209.
[4] Andrew P Black, Norman C Hutchinson, Eric Jul, and Henry M Levy. 2007. The development of the Emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 11–1.
[5] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7.
[6] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 16.
[7] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 398–407.
[8] Andre Charland and Brian Leroux. 2011. Mobile application development: web vs. native. *Commun. ACM* 54, 5 (2011), 49–53.
[9] Christopher Meiklejohn. Spry, prototype implementation in Erlang. https://github.com/cmeiklejohn/spry. (????).
[10] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM, 1–12.
[11] Edsger W Dijkstra. 2001. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*. Springer, 289–294.
[12] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *ACM SIGPLAN Notices*, Vol. 46.
[13] Marius Eriksen. 2013. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*. ACM, 5.
[14] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.
[15] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News* 33, 2 (2002), 51–59.
[16] James R Goodman. 1991. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department.
[17] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
[18] Charles Antony Richard Hoare. 1974. Monitors: An operating system structuring concept. In *The origin of concurrent programming*. Springer, 272–294.
[19] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers* 100, 9 (1979), 690–691.
[20] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
[21] Peter J Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166.
[22] Georg Lausen. 2009. Two-Phase Locking. In *Encyclopedia of Database Systems*. Springer, 3214–3218.
[23] Barbara Liskov. 1988. Distributed programming in Argus. *Commun. ACM* 31, 3 (1988), 300–312.
[24] Barbara Liskov and Liuba Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*. Vol. 23.
[25] 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*. ACM, 184–195.
[26] Christopher Meiklejohn and Peter Van Roy. 2015. Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation. In *Reliable Distributed Systems Workshop (SRDSW), 2015 IEEE 34th Symposium on*. IEEE, 62–67.
[27] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. 2006. A concurrent lambda calculus with futures. *Theoretical Computer Science* 364, 3 (2006), 338–356.
[28] Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm.. In *USENIX Annual Technical Conference*. 305–319.
[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.
[30] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 175–189.
[31] Claes Wikström. 1994. Distributed programming in Erlang. In *In PASCO'94-First International Symposium on Parallel Symbolic Computation*. Citeseer.