A Programming Model for Failure-Prone, Collaborative Robots

Nels Beckman Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213 USA
{nbeckman, jonathan.aldrich}@cs.cmu.edu

***Abstract*— A major problem in programming failure-prone collaborative robots is liveness. How do we ensure that the failure of one robot does not cause other robots to be permanently unavailable if, for example, that robot was a leader of others? In this paper, we propose a general mechanism which could be added to existing RPC libraries that allows applications to detect failure and execute programmer-specified recovery code.**

I. INTRODUCTION

A. *Collaborative Robots*

As robots become more sophisticated, writing applications for them becomes a more daunting task. In this work our goal is to ease the development of software for a particular class of robotic systems; those in which large numbers of robots must work collaboratively to accomplish some goal. (We refer to these collections of robots working collaboratively as robot “ensembles” or “systems.”) Our immediate inspiration comes from the Claytronics project [2], which focuses on the development of robots that work collaboratively to form dynamic three-dimensional shapes. However, previous work suggests [5], [11] that this is a growing domain with a variety of potential applications.

The domain itself is characterized by a few distinctive features. Because these are robots whose purpose is to effect some result in the real world, sensor and actuator activity is important. We focus, in particular, on robot ensembles for which using actuators for purposes like movement or change in form is the primary goal. Also there are soft real-time constraints on the overall actions of the robots. While it may not be important for any one robot to accomplish a task in a specific time-frame, it is important that collaborative tasks be completed in a reasonable amount of time, since applications will be interacting with the real world. In this domain applications will be massively distributed. Because of the physical distribution of the robots and the overhead associated with global communication, applications will have to be designed to work in parallel using primarily local communication. These types of applications have proven quite difficult to develop in the past.

The final defining characteristic of this domain, and the one we focus on in this work, is that the ensembles themselves are quite failure-prone. This is true for the following reasons:

- For such large numbers of robots to be affordable, the per-unit cost will have to be extremely small, removing the feasibility of hardware error handling features.
- For the same reason, the rate of mechanical imperfection of these robots is likely to far exceed any currently manufactured robot.
- Interaction between robot ensembles and the physical world increases the likelihood of unexpected events (e.g., encountering unexpected debris).
- The large number of robots taking part in an ensemble at any given time, in combination with the above factors will all but assure that some will fail over an application’s execution.

Properly accounting for failure scenarios in applications is notoriously hard. A large number of failures might possibly occur, potentially affecting the application in many subtle ways. Developing applications for this domain is already extremely difficult, given the massively distributed nature of the systems and the soft real-time constraints. So far, few application developers have put in the effort to failure-proof their applications. In this work we present a new programming model which provides an abstraction making failures easier to reason about. This abstraction is targeted at failures that might cause otherwise working robots to deadlock. It allows the programmer to easily express the sections of code during which liveness concerns exist, and which actions must be taken in order to preserve liveness.

B. *Approach*

For ease of presentation, we express our new model as a programming language primitive, the `fail_block`, but it could also be implemented as a library. The `fail_block` lexically surrounds a section of code in which transfers of control to other hosts in the ensemble occur. Then, in the event of host failure during that section of code, the underlying system helps the programmer to ensure liveness of the overall application by executing compensating actions. Furthermore, a successful exit from the `fail_block` guarantees to the programmer that all actuator commands have been successfully issued.

The paper proceeds as follows: Section II describes related work and explains why the goals of this paper have not already been trivially met. Additionally, this section describes our assumed failure model, some failure scenarios we would like our system to be able to handle, and ML5, the programming language upon which we build our work.

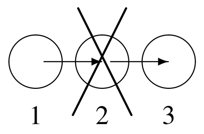

Fig. 1. After the thread of control has been transfered from Host 1 to Host 2 to Host 3, Host 2 fails.

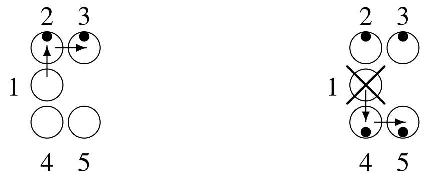

Fig. 2. On the left, the thread of control passes through hosts 2 and 3, setting Host 1 as their leader. Then, on the right, as the thread passes through hosts 5 and 6, Host 1 fails. The dark circle represents Host 1 being set as the leader of other hosts.

Section III describes our new model in detail and shows examples of its use. Finally Section IV concludes.

II. BACKGROUND AND RELATED WORK

A. *Specific Failure Scenarios*

As a motivating example for our new programming model, we will examine two specific failure scenarios. In Figure 1 we have three hosts who are communicating in order to achieve some goal. In this scenario, Host 1 transfers the thread of control to Host 2, which in turns transfers the thread of control to Host 3. (We will frequently use the metaphor of a thread migrating from host to host which is appropriate in synchronous distributed paradigms like RPC. When Host 1 calls a function on Host 2, this can be thought of the thread of control migrating from Host 1 to Host 2.)

Now, the question is, what should happen if Host 2 were to fail at this point in time? We would like for the thread of control to return to Host 1, throw some kind of exception indicating this failure, and allow computation to continue from the point where control was transfered to Host 2. However, using our thread migration metaphor, it is not exactly clear how this transfer of control will take place. In particular, if the system uses point-to-point communication, the only known route from Host 3 back to Host 1 has just failed and it is not clear that a new route can easily be found. Note that traditional RPC systems simulate thread migration by having Host 1 spontaneously resume the thread's execution in the event of a timeout.

In the second scenario, shown in Figure 2, we have a larger group of hosts working together. In this scenario, we are attempting to accomplish a goal which requires one host to be the "leader" in some application-specific sense. Here, that leader is Host 1. Host 1 transfers control to hosts 2 and 3, and assigns its own address to the memory location where a host's leader is stored. After returning control to Host 1, control is passed to other hosts in the group. At this time Host 1, the leader of the group, fails. In this scenario we can not resume the thread of control, since the natural candidate, Host 1, has failed. However, we do not want the failure of Host 1 to prevent the other hosts from performing work. This is possible in applications where hosts do not act until commanded by a leader or distinguished host, as these hosts would now wait indefinitely on a command that will never arrive. Hosts 2 through 5 need to have their leader fields reset.

B. *Failure Model*

In this work, we assume a relatively realistic failure model. (Failure means the inability to perform any computation.) We assume that hosts can fail at any point in time, and that they can fail permanently, or fail and then return to working status at a later point in time. It is also assumed that messages may fail to be delivered or delayed indefinitely. Similarly, any computation may take an unbounded amount of time. We assume that the successful delivery of a command to a hardware actuator can be confirmed or denied, but we realize that the successful performance of the actuator's command may be difficult to verify. We *do* assume that there are no malicious hosts in the system. Additionally, we assume that there is no duplication of messages, or at least that a lower-level protocol ensures duplicate messages are not delivered to the application level.

C. *Existing RPC Systems*

We must point out that the goals of our new programming model are not met trivially by existing work.

Remote Procedure Call (RPC) mechanisms are in wide use, and many come with limited forms of failure detection. However, currently available RPC libraries and more modern distributed object libraries, such as Java RMI [13] and CORBA [14], have weaker failure detection and weaker failure recovery than the model that we propose. Most RPC systems use a failure detection and recovery mechanism consisting of timeouts and exceptions. When transferring control across host boundaries, the application developer typically specifies a timeout period, a length of time after which the client will give up on the server by throwing a failure exception. Timeout mechanisms are typically the only form of failure detection. Of course, in a system where computation can take arbitrarily long amounts of time, selecting a good timeout period is difficult. Our model instead builds upon the more advanced concept of a failure detector [7] which actively attempts to detect host failure by periodically contacting remote hosts.

While one could imagine adding this functionality to RPC systems, the failure recovery mechanisms of RPC systems leave much to be desired. As mentioned, the thread of control is logically returned to the client via an exception. This allows the client to perform failure recovery actions, but does not give the server the chance to do so. This may seem like a strange thing to say. Why should the server need to recover from failures when it is the host that supposedly has failed? In fact, there are three situations where we might like to be able to run failure-recovery code on the server; i.) when the client *thinks* that the server has failed but in reality it hasn't, ii.) when the client itself has failed, and iii.) in a system where communication is routed through the robots themselves, when a different host on the communication route has failed. Figure 1 is an example of this type of scenario. Current RPC systems do not give the programmer a mechanism by which to specify failure-recovery code on the server end of a communication.

Finally, problems that reduce liveness often may be the result of a series of interactions among a group of hosts. In other words, the failure that deadlocks a certain host may occur after that host's interaction with the failed host, rather than during the middle of it. The scenario depicted in Figure 2 is a perfect example of this. Existing RPC systems do not account for this possibility.

D. Failure Detection

An important piece of our model is detecting when some host in a set of hosts have failed. We rely on existing work to provide implementations of these so-called "failure detectors" [1]. For a given group of hosts dynamically formed when using our programming model, the hosts would like to be able to reach consensus on whether or not some host has failed. However, in "time-free" systems such as the ones we describe (systems where there is no bound on the time it takes to perform a computation step or send a message from one host to another) it has been shown that this type of consensus cannot be formed [9]. Our approach takes advantage of certain features of the domain, discussed in Section III-E, so that the inability of a group of hosts to reach perfect consensus does not matter. Failure detectors that provide accurate information during times of system stability have been implemented [7]. Their suspicions of failure are good enough for our needs.

E. Transactional Systems

The semantics of the compensating actions in our model are in some ways quite similar to the roll-back behavior typical of transactional systems. The idea of using transactions to ensure consistency is a long-studied one, and many systems make use of Two-Phase Commit [6] for this purpose. One possibility we considered was treating the `fail_block` as an operation that commits effects on each host when `end` is reached if no hosts have failed, but aborts otherwise. However there are a few reasons why transactions were inappropriate. First, there is a basic problem of higher overhead in transaction-based systems. Commit protocols generally require one or two extra rounds of communication during which the application cannot move forward. All of the additional messages that our model requires can be done in parallel with forward progress of the system.

Another problem with standard Two-Phase Commit protocols is that hosts can block indefinitely if the coordinator fails [3], which is certainly bad for preserving liveness. Non-blocking commit protocols without this problem do exist, but they are typically more heavy-weight than Two-Phase Commit protocols and they bound the number of hosts that can fail [3]. Our model has no such limit.

F. The ML5 Programming Language

The examples given in this paper show our new programming language primitives as extensions to the ML5 programming language (which we have simplified in this paper for presentation purposes). ML5 [8] is a programming language based on the functional programming language ML that allows distributed algorithms to be programmed from a local perspective. This is possible through the use of a new primitive, `get`, which transfers control from one host to another. This primitive, appears as follows:

$$\texttt{get}[h]e$$

This means that expression e will be executed on host h. The result of the expression will be returned across the network and will be the result of the entire `get` expression. Since `get` is itself an expression, host h can in turn transfer control to another host. ML5 has an advanced type system that determines statically which data needs to be sent across the network in order to evaluate the expression, as well as ensuring that expressions do not attempt to use values such as memory references on hosts where they do not make sense.

We believe that ML5 makes our example algorithms easier to understand because distributed algorithms can be programed from the perspective of a single thread. This in turn makes the `fail_block` easier to understand since it can lexically enclose computations at several hosts. However, the programming model that we have developed can be adapted to work in any programming language with standard RPC features.

III. THE MODEL

A. Two Pieces

Failure detectors, and the actions that are taken upon failure, are essential for keeping the members of a distributed system live in the event of host failure. We propose a new model that detects host failures and executes compensating actions to preserve liveness when necessary.

This model consists of two pieces. The first piece, here exemplified as the programming language primitive `fail_block`, is a mechanism for signifying the logical time period and the hosts over which failure should be detected. Essentially we are specifying the members of the group that should be attempting to determine if one of the group has failed, and the period of time during which these group members should actively attempt to detect these failures. The `fail_block` also determines the point to which control flow should return in the event of a host failure.

The second piece, here exemplified by the programming language primitive `push_comp`, is a mechanism that allows programmers to specify code that should be executed by a host in the event that a failure is detected. These two mechanisms form the basis for our new model.

B. The Fail Block

The `fail_block` appears as follows:

$$\texttt{fail\_block}\ e\ \texttt{end},$$

This expression returns the value of an arbitrary sub-expression e upon successful completion. At the point in the program execution where we encounter this expression, a globally unique identifier i is created. This is the identifier of the operation that is about to occur. During the execution of e, whenever the thread of control is transfered to another host, that host is told that it is executing within operation i. The members of a group are all the hosts that have been told they are executing in operation i. Note that the members of a group are determined dynamically, and may differ based on which paths are taken in a given section of code.

When expression e has been completely evaluated, a message is sent to the members of the group notifying them that operation i has ended. This signifies to each member of the group that they no longer have to prepare for the possibility of failure; The code that executes in between the `fail_block` and `end` is the logical time period during which liveness is important. Consider the following code excerpt:

```
1:  fail_block
2:    (* Begin on host 1 *)
3:    get[w_2,a_2](
4:      (* do work on host 2 *)
5:      ...
6:      get[w_3,a_3](
7:        (* do work on host 3 *)
8:        ...
9:      )
10:   );
11:
12:   get[w_4,a_4](
13:     (* do work on host 4 *)
14:      ...
15:   )
16: end
```

Immediately before line 16, the group consists of four members, hosts one through four.

The group that we have formed is the basis for our failure detection. Our model does not specify one particular algorithm for detecting host failure within a group of hosts. We leave the details of this problem to existing work [7]. However, we imagine a simple active scheme where hosts periodically "ping" the members of the group with whom they have directly interacted and declare a failure if the host does not respond to these pings. This simple scheme would cut down on the number of total messages necessary when compared to a system where every host in the group attempted to contact every other member.

Finally, in the event that a failure of some kind is detected, the result of the entire `fail_block` is that an exception is thrown. This will resume the thread of control on Host 1, assuming that Host 1 still exists.

C. Specifying Compensations and Error Handling

In order to ensure the liveness of a still-living host in the event of a failure, that host must be allowed to execute code that will return it to a live state. For this purpose, we borrow the concept of compensating actions. Compensating actions were originally proposed as a programming language tool that would automatically free resources such as file handlers even in the event of an exceptional exit from a code block [12]. In our system, however, compensating actions are only executed in the event that a failure is detected.

Compensating actions are given to a host using the `push_comp` programming language primitive, which appears as follows:

`push_comp` e

Here e is an arbitrary expression. If this expression is called outside of a `fail_block`, nothing happens. Otherwise, `push_comp` takes the expression e, does not evaluate it but rather pushes it on top of a stack on the host where it was called. Each host may in fact contain multiple stacks, one for each operation (identified by a unique ID) that it is taking part in. If a host fails in a group and this failure is detected, then all of the compensations in the stack identified by that operation's unique ID will be executed in last-in, first-out order. Revisiting our earlier example, this gives us the following:

```
5:  get[a_3](
5a:   myLeader := a_1;
5b:   push_comp
5c:     (if   myLeader = a_1
5d:      then myLeader :=NONE
5e:      else ()
5f:     )
5g:   );
6:    (* do work on host 3 *)
7:  )
```

In this modified example, Host 3 assigns the address of the first host to a memory cell, `myLeader`. In this hypothetical application, having a host that no longer exists as your leader will lead to deadlock, as hosts only accept commands from their leader. Therefore, Host 3 adds as a compensating action an expression that clears its `myLeader` reference cell. Note that this solves the problem presented in Figure 2. In order to avoid a race-condition between the action and the addition of its compensating action, we assure a failure can only interrupt the thread of control at certain designated points in the code, for instance during a call to or return from the `get` primitive.

The compensating actions for a given operation ID will be executed when the host declares that there has been a failure. This will occur when that host's failure detector determines that one member of the operation group has failed. At this point, the host will share this failure information with any other members of the group it had been in contact with previously. These members will accordingly execute their compensating actions. Furthermore, any time that a member of the group that is unaware of the declared failure attempts to communicate with a member of the group that is aware of the failure, that member will also be informed of the failure.

This prevents a host that has temporarily failed from coming back to life and continuing with an operation that has already been declared a failure.

Finally, when or if the host where the `fail_block` was begun finds out about the failure, this host will continue execution of the thread by throwing an exception from the location of the `fail_block`. This gives the application programmer the ability to catch such an exception and attempt a different course of execution.

D. Guarantees Provided

Out new programming model provides certain guarantees to the programmer that we will now describe.

For a given `fail_block` enclosing transfers of control to other hosts, if the `fail_block` returns successfully then

- No host failure was encountered by the thread as it passed through the various hosts, and
- All of the actuator commands issued by the thread as it passed through those hosts were successfully issued.

If on the other hand, a host failure was encountered by the thread as it moved from host to host or if a failure is detected on a previously visited host before the end of the `fail_block` is reached, then for every host n visited within the block,

- Either n failed permanently,
- Or,
  - If n is the host on which the `fail_block` was begun, then an exception signaling the error will be thrown.
  - Otherwise, if n is another host, the stack of compensations will be run.

The above guarantees do imply a low-level liveness guarantee. Unlike in protocols such as Two-Phase Commit where the entire system can be prevented from making progress due to the failure of certain hosts [3], in our system, threads are always able to move forward, whether it be executing application code or failure handling code. This is largely due to the fact that our model does not have to block waiting for agreement of any kind.

Finally, there is some implication of liveness at the application level, but it is dependent on the application code. Our system does guarantee that in the event of a host failure within the `fail_block`, compensating actions will be run. However, we have left it up to the programmer to determine which actions need to be taken as compensation to ensure overall application liveness. Therefore, it is possible that even with compensating actions applications may become “stuck” in some application-specific sense. We leave this problem for future work.

E. A Lack of Consistency

The guarantees provided in the previous section, when combined with features of the application domain, provide us with a major benefit; our model does not require consistency across the hosts in a group. Specifically, there is no need to verify that upon successful exit of the `fail_block` none of the other hosts in the group have executed their compensating actions. This is important because it has been shown in previous work that in systems where computation and communication can take an unbounded amount of time it is not always possible for the members of a group to reach consensus about whether one of the members has failed [9].

The reasoning behind why this consistency is not required is as follows: In the domain that we describe, the ultimate goal of any application is to effect actuator actions, and in order to issue a command to an actuator, the thread of control must transfer to the host on which that actuator is located. In terms of consistency, what we really are talking about is the thread of control proceeding past the `end` of an `fail_block`, signaling successful completion, when in fact a host in the group detected failure and executed its compensating actions. At this point we know something important about the host(s) that detected failure and the host(s) (if there is one) that actually failed; in between the time when the failure was detected and when the `fail_block` successfully completed no actuator actions were necessary on those hosts, because if they had been the thread of control would have had to migrate to those hosts, and it would have detected the inconsistency. Therefore, in the sense that actuator movements are the ultimate goal of any application in this domain, those hosts have already done their important work (or never had any to do).

We believe that this programming feature supports a “best effort” programming style, where robots in the ensemble do their best to perform a given action but can try other approaches if it is unsuccessful. We believe that this style of programming will be key to reducing application complexity when ensembles grow to thousands and even tens of thousands of robots.

All this raises the question, “what if we will not be making an actuator movement on a host, but we still need to know that it has not given up prematurely?” This tests the assumption that actuator movements are the ultimate goal of applications in this domain. For example, what if a robot performs an important structural function in a robot ensemble? What this question actually comes down to is whether the liveness of that structural robot, in the event that the robots that interact with it fail, is critical. For certain applications, the developer may decide that the stability of the structure is more important than liveness in the event of a failure. This is exactly the time when the `fail_block` should *not* be used, since this feature is intended to delineate those parts of the computation where liveness *does* matter. If the liveness of a structural host was important, then you would at least have to have some sort of timeout mechanism. Our system provides a natural way to express these timeouts without having to explicitly encode them each time.

IV. CONCLUSIONS AND FUTURE WORK

A. Contributions

Our work takes the existence of some distributed control-flow mechanism for granted, whether that mechanism be an

RPC-like language, or a a programming language like ML5. Our work expands the failure-handling capabilities of all these systems for use in the domain of collaborative robotics. Our work can be viewed as an improvement to these systems, and as such provides two specific additions:

- **Compensating Actions:** We have taken the concept of compensating actions [12], used in the past as a means of ensuring cleanup from exceptional paths, and extended it to work as a means of ensuring liveness in a distributed system. As such, our compensations only need to be executed in the case of failure, rather than in the general case.
- **Previously Contacted Hosts:** Failure mechanisms provided by existing distributed programming systems only apply to the server host that is currently being contacted by the client host. However, we have shown how the failure of a host can lead to the unavailability of a whole group of hosts, including hosts that were previously contacted. Our system allows these groups of hosts to be formed dynamically so that the appropriate compensating action can be taken on necessary hosts. The implicit building up of the members of the group is one of the key new contributions of this work.

By combining the above contributions with existing work on the detection of failures within a group, we have developed a programming abstraction that makes it simpler for developers to ensure liveness of their applications in the face of large-scale host failure.

B. Future Work

Currently we are developing an implementation of the model described in this paper so that we can study its practicality and expressiveness in real-world applications. Additionally, our model currently dictates that all hosts visited within the `fail_block` be implicitly added to the group of hosts, but we understand that this may not always be a desirable feature. We want to find a way to allow certain visited hosts to be excluded from the group. In our model, nothing special is done to prevent improperly synchronized interaction between multiple threads in compensating action. While this is a concern, it is currently outside the scope of this paper, and we leave the issue for future work. Finally, at this point in time we force the burden of tracking which effects have occurred on the programmer. However, it is our belief that this system could be enhanced with ideas from software transactional memory [10], [4] so that the underlying runtime system would track and automatically roll back modified memory.

C. Conclusions

In this paper we presented a new programmer model that makes it easier for developers of software for distributed, collaborative robots to deal with the failure of hosts. We observe that in order for the overall application to make progress, certain effects of computation may need to be "undone" in the event of host failure. The model we have described, here expressed as a programming language feature, allows developers to specify sections of code over which failure could lead to liveness problems, and implicitly which hosts it might affect. Developers also specify the compensating actions that should be taken on these hosts in the event that a failure is detected. While hosts cannot always come to agreement upon whether or not a failure has occurred, in this domain this agreement is not essential.

V. Acknowledgments

Thanks to Kevin Bierhoff, Ciera Christopher, Michael Ashley-Rollman and Padmanabhan Pillai for their valuable feedback.

References

[1] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[2] Claytronics project website. `http://www.cs.cmu.edu/~claytronics/`.

[3] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[4] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[5] M.W. Jorgensen, E.H. Ostergaard, and H.H. Lund. Modular atron: Modules for a self-reconfigurable robot. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings*, pages 2068–2073, 2004.

[6] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, 1993.

[7] Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *2003 International Conference on Dependable Systems and Networks (DSN'03)*, page 351, 2003.

[8] Tom Murphy, VII, Karl Crary, and Robert Harper. Distributed control flow with classical modal logic. In Luke Ong, editor, *Computer Science Logic, 19th International Workshop (CSL 2005)*, Lecture Notes in Computer Science. Springer, August 2005.

[9] Michel Reynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70, 2005.

[10] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[11] J.W. Suh, S.B. Homans, and M. Yim. Telecubes: mechanical design of a module for self-reconfigurable robotics. In *Robotics and Automation, 2002. (ICRA 2002) Proceedings.*, pages 4095–4101, 2002.

[12] Westley Weimer and George C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press.

[13] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[14] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing. *SIGOPS Oper. Syst. Rev.*, 30(2):4–31, 1996.