Programming Models and Languages for Distributed Computation

Christopher S. Meiklejohn
Université catholique de Louvain
Louvain-la-Neuve, Belgium
christopher.meiklejohn@uclouvain.be

October 18, 2016

Contents

1 Remote Procedure Call

1.1 Relevant Reading

- *A Critique of the Remote Procedure Call Paradigm*, Tanenbaum and van Renesse, 1987 [40].
- *A Note On Distributed Computing*, Kendall, Waldo, Wollrath, Wyant, 1994 [25].
- *It's Just A Mapping Problem*, Vinoski, 2003 [41].
- *Convenience Over Correctness*, Vinoski, 2008 [42].

1.2 Commentary

> "Does developer convenience really trump correctness, scalability, performance, separation of concerns, extensibility, and accidental complexity?" [42]

1.2.1 Timeline

- 1974: RFC 674,
  "Procedure Call Protocol Documents, Version 2"
  RFC 674 attempts to define a general way to share resources across **all 70 nodes** of the Internet. This work is performed at Bolt, Beranek and Newman (BBN Technologies).[1]
- 1975: RFC 684,
  "A Commentary on Procedure Calling as a Network Protocol"
  First outlines the problems of RPC and how they related to fundamental problems in distributed systems.
- 1976: RFC 707,
  "A High-Level Framework for Network-Based Resource Sharing"
  An attempt to "mechanize" the TELNET and FTP protocols through a generalization to functions.
- 1984: "Implementing Remote Procedure Calls" [5]
  In this work, the **Cedar RPC** mechanism is designed at Xerox PARC.
- 1987: Distribution and Abstract Types in Emerald [7]
  One of the first major implementations of distributed objects containing distribution-specific calling conventions, such as *call-by-move*.

[1] Fun fact: BBN's Internet division would later become Genuity and then Level3 out of bankruptcy and has autonomous system number 1 (ASN 1).

- 1987: A Critique of the Remote Procedure Call Paradigm [40]
  Tanenbaum and van Renesse provide a criticism, similar to the one in RFC 684, on why the RPC model is the wrong model for distributed computing.

- 1988: Distributed Programing in Argus [27]
  In Argus, atomic units of computation referred to as "guardians" coordinate application of effects and rollback.

- 1988: RFC 1057,
  "Remote Procedure Call Protocol Specification, Version 2"
  Defines **sunrpc** as a standard along with its supporting infrastructure, such as portmapper, that's used in the creation of NFS.

- 1991: CORBA 1.0
  CORBA 1.0 introduces distributed objects.

- 1994: A Note On Distributed Computing [25]
  Kendall et al. also talk in great length about why the RPC model, extended to objects, is problematic.

- 1996: A Distributed Object Model for the Java System [45]
  Introduces the Java RMI system.

- 1997: CORBA 2.0
  CORBA 2.0 represents the major release of CORBA that most people are familiar with.

- 1999-: EJB, XML-RPC, SOAP, REST, Thrift, Finagle, gRPC, etc.
  Modern day RPC mechanisms.

1.2.2 Overview

Remote Procedure Call (RPC) is a general term for executing a subroutine in a different address space without writing the actual code used to perform the remote execution. To provide an example, we can imagine a user wishing to invoke the random number generator function on another machine, but, the only difference between the local and remote invocation is supplying an additional node identifier where it should occur. While not the first implementation, because it was preceded by Apollo Computer's Network Computing System (NCS), the first major implementation to be widely known and adopted was the SunRPC mechanism, from Sun Microsystems, used to back their Network File System (NFS).

Remote Procedure Call mechanisms you may be more familiar with are Java's Remote Method Invocation (RMI), it's predecessor Modula-3's Network Objects, XML-RPC, SOAP, CORBA, Avro, Facebook's, (now Apache)

Thrift, Google's Protocol Buffers with Stubby, Twitter's Finagle, and Google's gRPC.

1.2.3 RFC 684

> "Rather, we take exception to PCPs underlying premise: that the procedure calling discipline is the starting point for building multi-computer systems."

RFC 684 is commentary on RFC 674 that introduced the Procedure Call Paradigm, Version 2 (PCP). This commentary highlights, what boils down to, three major points from a critical analysis of the Procedure Call Paradigm.

- Procedure calling is usually a primitive operation; by primitive, it should be an extremely fast context switch operation performed by the underlying abstraction.
- Local and remote calls each have different cost profiles; remote calls can be delayed, and in the event of failure, may **never return**.
- Asynchronous message passing, or sending a message and waiting for a response when the response is needed, is a much better model because it makes the passing of messages **explicit**.

Following from these three points, we see a series of concerns develop about this programming paradigm, all of which become a common theme across the 40+ years in RPC's history. These are:

- Difficulty in recovery after malfunction or error. For instance, do we rollback or throw exceptions? How do we handle these errors? Can we just try again?[2]
- Difficulty in sequencing operations. If all calls are synchronous and some of these calls can fail, it can require a significant amount of code to ensure correct re-execution to preserve order moving forward.
- Remote Procedure Call forces **synchronous programming**: a method is invoked and the invoking process waits for a response.
- Backpressure, or blocking on previous actions completing, load-shedding, or dropping messages on the floor when the system is overloaded, and priority servicing become more difficult with the call-and-response model of Remote Procedure Call.

[2] Some systems that attempt to address this include Liskov's promises [30] and Lee's [26] work on a reusable framework for linearizability.

1.2.4 RFC 707

> "Because of this cost differential, the applications programmer must exercise discretion in his use of remote resources, even though the mechanics of their use will have been greatly simplified by the RTE. Like virtual memory, the procedure call model offers great convenience, and therefore power, in exchange for reasonable alertness to the possibilities of abuse."

RFC 707 generalizes the ideas from RFC 684 and discusses the problem of resources sharing for services such as TELNET and FTP: each of these services presents a *different* interface for interacting with it, which requires the operator to know the specific protocol for interacting with that service. In realizing that both services like TELNET and FTP follow the call-and-response model, the authors propose an alternative idea: rather than needing to know all of the available commands and protocols on the remote machine, can we define a generic interface for executing a remote procedure that takes an argument list and follows the call-and-response model?

While we can, the problems of control flow and priority servicing outlined in RFC 684 remain; however, not enough that it prevents this model from being adopted by many systems in the future.

1.2.5 "A Critique of the Remote Procedure Call Paradigm"

> "We propose the following test for a general-purpose RPC system. Imagine that two programmers are working on a project. Programmer 1 is writing the main program. Programmer 2 is writing a collection of procedures to be called by the main program. The subject of RPC has never been mentioned and both programmers assume that all their code will be compiled and linked together into a single executable binary program and run on a free-standing computer, not connected to any networks."
>
> "At the very last minute, after all the code has been thoroughly tested, debugged, and documented and both programmers have quit their jobs and left the country, the project management is forced by unexpected, external circumstances to run the program on a distributed system. The main program must run on one computer, and each procedure must run on a different computer. We also assume that all the stub procedures are produced mechanically by a stub generating program."
>
> "It is our contention that a large number of things may now go wrong due to the fact that RPC tries to make remote procedure calls look exactly like local ones, but is unable to do it perfectly. Many of the problems can be solved by modifying the code is

> various ways, but then the transparency is lost. Once we admit that true transparency is impossible, and that programmers must know which calls are remote and which ones are local, we are faced with the question of whether a partially transparent mechanism is really better than one that was designed specifically for remote access and makes no attempt to make remote computations look local at all."

Tanenbaum and van Renesse directly attack the Remote Procedure Call paradigm stating that it is fundamentally wrong to treat local and remote calls the same. They state that the transparency that RPC tries to achieve is impossible and posit that a protocol that is designed specifically for remote access is better.

The authors go on to describe an alternative paradigm: a "virtual circuit". This alternative comes off sounding very similar to Distributed Erlang running with TCP: non-blocking send and receive operations across a sliding-window network protocol.

> "An alternative model that does not attempt any transparency in the first place is the virtual circuit model (e.g., the ISO OSI reference model [Zimmermann, 1980]). In this model, a full-duplex virtual circuit using a sliding window protocol is set up between the client and server. If nonblocking SEND and RECEIVE primitives are used, incoming messages can be signalled by interrupts to allow the maximum amount of parallelism between communication and computation."

Tanenbaum and van Renesse outline many of the same criticisms as RFC 684: latency, lack of parallelism, lack of streaming, exception handling and failure detection. In addition, they raise a few additional criticisms that we will highlight below.

Unexpected Messages With a synchronous call-and-response protocol between client and server, how is one supposed to send a message to the client that is unexpected if that client is not waiting for a message?

Single Threaded Servers How does one handle the situation where the server does not have a response ready for a client immediately – for instance, if it needs to wait on input from another server? Not only does this block the server, but it also blocks the client from proceeding further with local computation.[3]

[3] Our section on promises talks about this problem in depth [30].

> "There is, in fact, no protocol that guarantees that both sides definitely and unambiguously know that the RPC is over in the face of a lossy network." [40]

The Two Army Problem How do we handle requesting irreplaceable data? (or, more generally have two servers come to agreement that some RPC was successfully executed and the response received.) Well, we can acknowledge the receipt of that information, but then we would also have to acknowledge the acknowledgements to be sure the acknowledgement was delivered, right? This topic, central to the agreement problem, has been discussed extensively in distributed systems literature [18].

Parameters Tanenbaum and van Renesse discuss the problem of parameter passing and parameter marshalling. This issue becomes exacerbated with references in object systems such as CORBA, where specific distributed references have to be used to ensure they remain accessible and valid over time.

> "However, if there are reference parameters or pointers, things are more complicated. While it is obviously possible to copy pointers into the message, when the server tries to use them, it will not work correctly because the object pointed to will not be present."
>
> "Two possible solutions suggest themselves, each with major drawbacks. The first solution is to have the client stub not only put the pointer itself in the message, but also the thing pointed to. However, if the thing pointed to is the middle of a complex list structure containing pointers in both directions, sublists, etc., copying the entire structure into the message will be expensive. Furthermore, when it arrives, the structure will have to be reassembled at the same memory addresses that it had on the client side, because the server code will just perform indirection operations on the pointers as though it were working on local variables."
>
> ...
>
> "The other solution is just to pass the pointer itself. Every time the pointer is used, a message is sent back to the client to read or write the relevant word. The problem here is that we violate one of the basic rules: the compiler should not have to know that it is dealing with RPC. Normally the code produced for reading from a pointer is just to indirect from it. If remote pointers work differently from local pointers, the transparency of the RPC is lost."

Idempotence Finally, the authors highlight the problem of providing exactly-once semantics across the network and the power of idempotence. Tanenbaum and van Renesse say it very concisely.

> "Suppose a client does a nonidempotent RPC and the server crashes one machine instruction after finishing the operation, but before the server stub has had a chance to reply. The client stub times out and sends the request again. If the server has rebooted by then, there is a chance that the operation will be performed two or more times and thus fail."

1.2.6 CORBA

The Common Object Request Broker Architecture (CORBA) is an abstraction for object-oriented languages, popularized by C++, that allows you to communicate between different languages and different address spaces running on different machines. CORBA relied on the use of an Interface Definition Language (IDL) for specifying the interfaces of remote classes of objects; this IDL was used to generate stubs of what the remote systems object interfaces appeared as on the local machine. These IDL's would be used to generate mappings between the abstract interfaces provided by the IDL's and the actual implementations in languages such as C++ and Java.

CORBA attempted to provide several benefits to the application developer: language independence, OS-independence, architecture-independence, static typing through a mapping of abstract types in the IDL to machine and language specific implementations of those types, and object transfer, where objects can be migrated over the wire between different machines. CORBA's promise was that through the use of mapping that remote calls could appear as local calls, and that distributed systems related exceptions could be mapped into local exceptions and handled by local exception handling mechanisms.

However, as Vinoski points out in 2003, the evaluation of programming languages and abstractions based on transparency alone is flawed:

> "The goal is to merge middleware abstractions directly into the realm of the programming language, minimizing the impedance mismatch between the programming language world and the middleware world. For example, mappings make request invocations on distributed objects and services appear as normal programming-language function calls, and they map distributed system exceptions into native programming language exception-handling mechanisms." [41]

1.2.7 "A Note On Distributed Computing"

> "It is the thesis of this note that this unified view of objects is mistaken." [25]

In this pinnacle Waldo paper, they argue that "it is perilous to ignore the differences" between local and distributed computing and that the unified view of objects is flawed. [25] They cite two independent groups of work, the systems of Emerald and Argus, and the modern equivalents of those systems, Microsoft's DCOM and OMG's CORBA: all systems that extended the RPC mechanism to objects and method invocation.

We can summarize the "promise" of the unified view of objects, as Waldo does in the paper.

- Applications are designed using interfaces on the local machine.
- Objects are relocated, because of the transparency of location, to gain the desired application performance.
- The application is then tested with "real bullets."

This strategy for the design of a distributed application has two fundamental flaws. First, that the design of an application can be done with interfaces alone, and that this design will be discovered during the development of the application. Second, that application correctness does not depend on object location, but only the interfaces to each object.

Waldo swats down this design with the "three false principles" [25]:

> "there is a single natural object-oriented design for a given application, regardless of the context in which that application will be deployed"
>
> "failure and performance issues are tied to the implementation of the components of an application, and consideration of these issues should be left out of an initial design"
>
> "'the interface of an object is independent of the context in which that object is used"

1.2.8 "Every 10 years..."

> "The hard problems in distributed computing are not the problems of getting things on and off the wire." [25]

Waldo argues that every ten years we approach the problem of attempting to unify the view of local and remote computing and run into the same problems, again and again: **local and remote computing are fundamentally different.**

Latency Waldo argues that the most obvious difference should be the issues of latency: if you ignore latency, you will end up directly impacting software performance. He states that is it wrong to “rely on steadily increasing speed of the underlying hardware” and that it is not always possible to test with “real bullets”. Performance analysis and relocation is non-trivial and a design that is optimal at one point will not necessarily stay optimal.

Memory Access His criticisms of memory access are very specific to CORBA and its predecessors in the object space: objects can retain pointers to objects in the same address space, but once moved these pointers will no longer be valid. He states that one approach to solving the problem is distributed shared memory, but more practically techniques such as marshalling or replacement by CORBA references, which are marshalled for distributed access, are used.

Partial Failure Finally, the most fundamental problem: partial failure. In local computing, he argues, failures are detectable, total, and result in a return of control. This is not true with distributed computing: independent components may fail, failures are partial and a failure of a link is indistinguishable from a failure of a remote processor.

As always, Waldo says it best:

> “The question is not ‘can you make remote method invocation look like local method invocation?’ but rather ‘what is the price of making remote method invocation identical to local method invocation?”’ [25]

Waldo argues that there is only two paths forward if we want to achieve the goal of the unified object model.

- Treat all objects as local.
- Treat all objects as remote.

However, he states that if the real goal is to “make distributed computing as simple as local computing”, that the only real path forward is the first. This approach, he believes, is flawed, and that distribution is fundamentally different, and must be treated so.

> “This approach would also defeat the overall purpose of unifying the object models. The real reason for attempting such a unification is to make distributed computing more like local computing and thus make distributed computing easier. This second approach to unifying the models makes local computing as complex as distributed computing.” [25]

The paper provides two examples of where this paradigm is problematic, but we will highlight one case, that builds upon RPC.

1.2.9 Network File System

Sun Microsystem's **Network File System (NFS)**, built upon RPC, is one of the first distributed file systems to gain popularity. Network File System adhered to the existing filesystem API, but introduced an entire new class of failures resulting from network partitions, partial failure, and high latency. Network File System is a stateless protocol implemented in UDP; the decision to implement it this way is motivated by crash recovery avoidance and protocol simplification [35].

Network File System operated in two modes: soft mounting and hard mounting. Soft mounting introduced a set of new error codes related to the additional ways file operations could fail: these error codes were not known by existing UNIX applications and led to smaller adoption of this approach. Hard mounting introduced the opposite behavior for failures related to the network: **operations would block until they could be completed successfully**.

It's just a mapping problem, right? [41]

1.2.10 "Convenience Over Correctness"

> "We have a general-purpose imperative programming-language hammer, so we treat distributed computing as just another nail to bend to fit the programming models." [42]

Vinoski highlights three very important points in "Convenience Over Correctness" criticism of RPC many years later.

- **Interface Definition Languages (IDL) "impedance mismatch"**: base types may be easy to map, but more complex types may be less so.

- **Scalability:** the RPC paradigm does not have any first class support for caching, or mechanisms for mitigating high latency, and is remains a rather primitive operation to build distributed applications with.

- **Representational State Transfer (REST)**: REST is good: it specifically addresses the problem of managing distributed resources; but most frameworks built on top of REST alter the abstraction and present something that repeats the problem [4].

[4] For instance, if one was to build an object model on top of REST.

1.3 Impact and Implementations

Remote Procedure Call (RPC) has been around for a very long time and while many opponents of RPC have been extremely critical of it, it still remains one of the most widely used way of writing distributed applications. There's an unprecedented amount of use of frameworks for RPC such as Google's Protocol Buffers and Apache's Thrift used in production applications.

Frameworks such as Google's gRPC for HTTP/2.0 and Twitter's Finagle continue to reduce the amount of complexity in building applications with them, attempting to bring RPC to an even wider audience. For instance, Twitter's Finagle is protocol-independent and attempts to deal with the problems of distribution directly. Finagle does this through the use of futures, which allow composition and explicit sequencing; Google's gRPC does similar. These frameworks claim that since they do not attempt to hide the fact that the calls are remote, that it provides a better abstraction. However, now we have returned to the aforementioned problem of soft mounting in NFS: explicitly handling the flow control from the array of possible network exceptions that could occur, though mitigated through the use of promises/futures [5].

But, the question we have to ask ourselves is whether the abstraction of an individual method invocation or function call is the correct paradigm for building distributed applications. Is the idea of treating all remote objects as local, and making distribution as transparent as possible, the correct decision moving forward? Does it mask failure modes that will allow developers to build applications that will not operate correctly under partial failure?

When we talk about distributed programming languages today, many developers equate this to **programming languages** that can be, and have been used, to build **distributed systems.** For example, any language with concurrency primitives and the ability to open a network socket would suffice to build these systems; this does not imply that these languages are distributed programming languages.

But, a **distributed programming language** is where the distribution is **first class**. Languages like Go are more closely related to **concurrent** languages, where concurrency is first class; and, while concurrency is a requirement for distribution, these are different topics. CORBA is an example of trying to make distribution first class in languages such as C++.

Erlang [12, 38, 39] is one language where distribution is first class. Erlang has a RPC mechanism, but prefers the use of asynchronous message passing between processes. In fact, the RPC mechanism in Erlang is implemented using Erlang's native asynchronous message passing. While you can peek under the covers and see **where** processes are running, Erlang tries to make

[5] Our related post on promises and futures challenges whether that abstraction is right either.

the programmer assume that each process could be executing on a different node. Motivated by the expressiveness of the design, both Distributed Process, from the Cloud Haskell group, and Akka, in Scala, are examples that attempt to bring Erlang-style semantics to Haskell and Scala, respectively.

One approach taken in the Scala community for distributed programming is serializable closures [33], or what's known as the function shipping paradigm. In this model, entire functions are moved across the network, where the type system is used to ensure that all of values in scope can be properly serialized or marshalled as these closures move across the network. While this solves some of the problematic points in systems like CORBA and DCOM, it does not have a solution for the problem of how to ensure exactly-once execution of functions, or to handle partial failure where you can not distinguish the failure of the remote node from the network.

Languages like Bloom, and Bloom$_{\mathrm{L}}$, and Lasp [1, 13, 32] take an alternative approach: can we build abstractions that rely on asynchronous programming, very weak ordering and structuring our applications in such a way where they are tolerant to network anomalies such as message duplication and message re-ordering. While this approach is more expensive in terms of state transmission, and more restrictive in what types of computations can be expressed, this style of programming supports the creation of *correct-by-construction* distributed applications. These applications are highly tolerant to anomalies resulting from network failures by assuming all actors in the system are distributed. The restrictions, however, might prohibit wide adoption of these techniques.

So, we ask again:

> "Does developer convenience really trump correctness, scalability, performance, separation of concerns, extensibility, and accidental complexity?" [42]

2 PLITS

2.1 Relevant Reading

- *High Level Programming for Distributed Computing*, Feldman, Jerome A, CACM 1979 [14].

2.2 Commentary

> "A significant conclusion was that parallelism and data sharing are inherently difficult to combine effectively."

The Programming Language in the Sky (PLITS) project was an effort by the University of Rochester Computer Science department started in 1974 to take a serious look at how to make programming languages more declarative, and to see what benefits could be gained from using state-of-the-art compiler technology. This paper, specifically focuses on the problem of addressing the need for programming language constructs for distributed computing, that normally do not arise in conventional programming[6].

At a high level, PASCAL-PLITS views distributed computing as a group of computers communicating over low bandwidth, unreliable communication paths, applications would consist of communication between modules via an asynchronous message protocol instead of through the use of subroutines to avoid having to wait for a response to a request. Modules only communicate through message passing, and each message is composed of a set of name-value pairs which are called slots: names are uninterpreted string and values are an element of some primitive type (for instance, integers.)

To avoid discussion of many of the PASCAL specific details outlined in the formal specification section of the paper, the intuition on how message passing between two modules operates is rather straightforward:

- Messages can be sent between two modules where slots are compatible: for instance, both a server and client must define the types of messages they can receive and send, and they must specify the slots of the message they are going to read. It's perfectly fine in this system to define a message that underspecifies the message slots – readers can only access what they know about through their local specification.

- Messages are sent to recipients by specifying the recipient's module identifier and can be selectively received at the recipient by sender's module identifier.

- Messages in PASCAL-PLITS arrive in FIFO order between modules and waiting on a message that never arrives after a particular time interval will throw an application-level exception.

[6] It is easy to argue today that conventional programming, **is** distributed computing, but I digress.

- Messages support a grouping behavior that allow for fixed size repetitions of values by type, nested one level deep, as values in slots.

An interesting idea that the paper presents is the idea of forwarding the message to another module for handling processing. For instance, a server might need to accept a message and pass that message to another client to handle specific processing of that request. In that case, the path from original sender, to server, to client handling the request must be retraced when the request is complete and the response is being returned to the original caller. PASCAL-PLITS offers an alternative method: a unique transaction identifier is transmitted with the original message, so the client handling the request can send it directly back to the original sender, by passing the coordinating server module. In PASCAL-PLITS, this is referred to as a transaction key, and receive messages can specifically request they want to selectively receive messages **about** a key, instead of **from** a specific recipient; because of this proxy-like behavior, modules need to be able to read only particular slots of the message, and ignore, but forward, the remainder[7].

The authors argue for message passing over subroutines:

> “The message paradigm has several advantages over subroutine calls. If the modules were in different languages, the subroutine call mechanisms would have to be made compatible. Any sophisticated lockout procedure would require the internal coding of queues equivalent to what the message switcher provides. In the subroutine discipline, a module which tries to execute a locked subroutine is unable to proceed with other computation. The total picture on the relative value of messages and calls is much more complex...”

The authors also argue for the encapsulation provided by message passing as the only interface to data:

> “There are other interesting features that arise when messages are combined with the idea of modules. The most obvious feature of PLITS programming is the high degree of locality and protection it provides. Each PLITS module is totally self-contained and communicates solely through messages. This means that no local variables can even be examined from the outside, no procedures invoked, etc. A module can be asked to return or update a value, execute a function, etc. It now becomes quite natural to

[7] The dilligent reader will notice that the PASCAL-PLITS system argues that they provide stronger guarantees than “very strong typing” through this slot system, but it’s unclear how slot compatibility and forwarding can be verified without either dataflow analysis or type checking at compile time: the slot system is supposed to be more expressive because it describes application level behavior.

> screen requests for validity (much more than type checking), to guard against conflicting demands on a data structure, etc. This does not solve all the problems attacked by structured programming strictures, but does make it clear what has to be done and where.“

The authors use this encapsulation to solve the exclusion problem: if module B is storing data that A wants to compare and swap, B simply ignores messages that do not contain the transaction key generated by A until the swap is complete (or, does not handle messages related to the keys that are beings swapped until the operation is complete.) However, this ultimately brings additional complexities, as now the module itself must deal with the queue management of messages that are waiting to be processed while resources are locked.

Perhaps the most interesting part of the PLITS system is the distribution system for running distributed jobs. Distribution is tackled through a layered approach:

- Networks are made up of a set of machines.
- Machines are made up of multiple sites, each with a kernel that keeps track of scheduling and which modules are waiting to receive messages: within a site, messages must have the same format and same primitive data representation.
- Kernels are responsible for distribution of messages within a site, forwarding messages between sites and resource allocation.
- Co-located with a kernel is a Host Control Program (HCP) that is responsible for forwarding within sites on the same machine, and forwarding messages between machines.
- Finally, the HCP is divided into two components: a distributed job manager and communication manager: one for job-lifecycle related operations and one for managing communication between machines[8].

Messages to the recipients that are overloaded are buffered by the sender: this procedure is done on the sender side through process suspension. If nodes are located together at the same site, the kernel is responsible for this process, if not, the distributed communication manager is responsible for control flow management.

Identification of where a module are located is done through the module name, an incarnation number, site number and local module number:

[8] Not discussed in the literature is how it’s required that the HCP guarantees reliable transmission and handles flow control and error handling.

this means that the identifier of a process can locate where its running on the network alone, without the need for a global registry. However, given this is done on a module instance basis, modules only ever live on the same machine. To get around this, the authors suggest using equivalent modules across machines and having the caller make the decision where to send the message. As the authors state it quite succinctly "contrary to current fantasies about distributed computing".

2.3 Impact and Implementations

So, what's the contribution of this paper?

Well, as the authors describe it, it's the "module-message" paradigm: something that you've probably seen before and the reason this paper has sounded so familiar: **Erlang**, or more specifically **Distributed Erlang**.

PLITS, only briefly mentioned in Joe Armstrong's thesis [2], bears a striking resemblance to Erlang and brings some of the features that are used on a daily basis by Erlang developers. We quickly summarize those features now.

- Selective receive through the use of a transaction identifier has a strong resemblance to Erlang's ability to generate a globally unique reference and use it as criteria to select from the process mailbox. This supports the forwarding behavior of processes that serve mainly for the routing of a request.

- Repetition in slot values, where each value is tagged in the initial position by type bear a strong resemblance of how tagged tuples are used in records to specify a typed value purely as tuples.

- The idea of encapsulation and locking or queueing messages for exclusion is similar to how processes control access to resources using the "generic abstractions" provided by Erlang: specifically, the "generic server" or "gen_server". The generic server and generic finite state machine are both to control access to some state and has the ability to either select specific messages for processing related to a message identifier, or interleave requests if possible based on selective receive and asynchronous messaging.

- The design of both the PLITS kernel, as a process scheduler responsible for waking and suspending processes waiting on messages, as well as the Host Control Program work very similar to the implementation of Distributed Erlang: processes running on remote machines are uniquely identified by a process identifier that encodes the machine name and messages are transparently delivered to them under what aims to achieve reliable transmission, but sometimes falls short.

We should be clear, however, that PLITS is **not** an actor-based language such as Erlang. While PLITS shares many of the same design patterns and abstractions, the language is fundamentally a module-message system: modules are identified by name, which includes a location and incarnation number and are the target of messages, not processes.

3 ARGUS

3.1 Relevant Reading

- *Abstraction Mechanisms in CLU*, Liskov, Barbara and Snyder, Alan and Atkinson, Russell and Schaffert, Craig, CACM 1977 [31].

- *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, Liskov, Barbara and Scheifler, Robert, TOPLAS 1982 [29].

- *Orphan Detection in the Argus System*, Walker, Edward Franklin, DTIC 1984 [43].

- *Implementation of Argus*, Liskov, Barbara and Curtis, Dorothy and Johnson, Paul and Scheifer, Robert, SIGOPS 1987 [28].

- *Distributed Programming in Argus*, Liskov, Barbara CACM 1988 [27].

3.2 Commentary

> "However, regardless of advances in hardware, we believe atomic actions are necessary and are a natural model for a large class of applications. If the language does not provide actions, the user will be compelled to implement them, perhaps unwittingly reimplementing with each new application, and may implement them incorrectly."

3.2.1 Overview

The focus of these papers is the ARGUS system (and it's roots with the programming language CLU), developed in the Laboratory for Computer Science at the Massachusetts Institute of Technology. ARGUS was designed to solve the problem of programming language support for the construction and maintenance of distributed programs constructed from modules executing at, and communicating from, geographically distinct nodes. ARGUS was designed with the following goals in mind:

- **Service:** Programs should have localized failures, and be geographically distributed with replicated data for both fault-tolerance and availability.

- **Reconfiguration:** Software should be able to be reconfigured while the system is running: this allows for the addition of additional capacity to increase processing power, decrease response times, or increase the availability of data.

- **Autonomy:** Nodes in the system may be owned by individuals or organizations that need to control what data is replicated at the node for political or sociological reasons.
- **Distribution:** Programs should be able to control explicit placement of data to ensure responsiveness and cost-effectiveness of hardware in the system.
- **Concurrency:** Distribution should be able to exploit the available concurrency in the system.
- **Consistency:** Consistency must be maintained, specifically for invariant preservation: for instance, conservation of funds during a transfer between two accounts.

Of all of the aforementioned concerns, the authors posit that the most difficult of these to provide is **consistency**: consistency becomes much more difficult to preserve in a system where coordination is minimized to avoid interference and m ask failures of the network. Therefore, the authors present a technique for integrating **atomicity** as fundamental concept in a programming language.

3.2.2 Atomicity and Actions

Since data resiliency only guarantees consistency under a quiescent environment, it is necessary to make some operations in the system **atomic**. When the authors state **atomic**, they mean two things:

- **Indivisibility:** the execution of an activity in the system never appears to overlap with another activity in the system.
- **Recoverability:** the overall effect of an activity is all or nothing: in the event of a failure, the activity must be completed after recovery or all objects must be restored to their initial state.

ARGUS introduces **actions**: atomic transactions that need to either be committed or aborted and provide both the indivisibility and recoverability properties. If an action happens to abort, it is as if the the effects had never happened; if an action happens to commit, all modified objects in the system take on their new state. However, since ARGUS is a concurrent system, shared objects that will be manipulated by concurrent atomic actions must be done via synchronization, with read/and write locks.

To achieve this, ARGUS introduces a type of atomic object, or as they call it, **atomic abstract data types**: sets of objects and primitive operations for interacting with those objects. These objects are similar to normal data types, but are extended with operations that ensure indivisibility and recoverability. Access to objects within an atomic abstract data type are done through locking: read/write locks with versioning is used to facilitate concurrency access in the system. Write operations operate against a copy of the current version, and when the action completes, the new version of the object is written, replacing the old.

3.2.3 Nested Actions

The authors argue for indivisibility as a desired property, even if it reduces the amount of potential concurrency in the system:

> "It has been argued that indivisibility is too strong a property for certain applications because it limits the amount of potential concurrency. We believe that indivisibility is the desired property for most application, *if* it is required only at the appropriate levels of abstraction. ARGUS provides a mechanism for *user-defined* atomic data types."

Nested actions, or subactions, can be used to further divide actions and introduce concurrency within an action: each action can contain any number of subactions that can be either executed sequentially or concurrently. Subactions can commit or abort independently and this behavior has no impact on the parent action. However, an abort of a parent will force the abort of all subactions.

Nested actions provide support for the composition of actions into a larger action where some of the nested actions can run in parallel or might fail and have to be compensated for. For instance, if one subaction has to contact a remote guardian that may be unavailable, that subaction can abort and the operation retried against another remote guardian: the authors refer to this as "several ways to accomplish a task."

Because nested actions may fail, interact with several guardians, and have "checkpointing" behavior as they either abort or succeed, each nested actions ancestor needs to track metadata during the execution:

1. Each subaction will inherit locks from its ancestor: read locks are directly inherited, but write locks can only be inherited if all read locks can also be inherited.

2. Each subaction can abort or succeed: if the action aborts, the ancestor will restore the state, or "checkpoint", of the execution to before the operation was executed.

3. Each version created by a subaction needs to be propagated, to ensure that the effects of a previous write to a successful nested action are available to a subsequent subaction.

4. Each top-level action will track what's referred to as a *plist*, which is a list of all of the guardians contacted during the execution of the action to be used in the final two-phase commit protocol to commit the action and write to stable storage.

3.2.4 Remote Procedure Call and the Network

The authors argue for Remote Procedure Call and state that nested actions can be used for masking the communication failures inherent in the paradigm:

> "In fact, we believe the form of communication that is needed is *remote procedure call,* with *at-most-once* semantics, namely, that (effectively) either the message is delivered and acts on exactly once, with exactly one reply received, or the message is never delivered and the sender is so informed."

The authors believe that low-level issues from the system should be shielded from the user, such as packet retransmission, but do believe that the system should make a reasonable attempt of delivery for messages. However, the users believe that the notion that long delays are possible and that ultimate failure is possible, should be made present to the user. The authors propose that the subaction should be used for this: individual subactions can abort without causing the entire parent action to abort: in the case where these subactions do abort, the user should be able to take action, such as try an alternative replica to service the request.

In this model, some top-level actions can not be aborted. Consider the case of an airline reservation system: the clerk can initiate operations for making a reservation and subactions can take action and try multiple replicas if, for instance, the primary can not be contacted. However, an external event performed by a top-level action, such as printing a check, can not be undone and needs to have a compensating action for dealing with this behavior. To alleviate this problem, the authors suggest breaking these into two separate top-level actions, where they are sequenced to ensure the completion of one before the execution of the subsequent operation.

Finally, the authors acknowledge that even in this model timeouts are necessary to resolve issues that might arise from contention or deadlocks between atomic resources.

3.2.5 Semantics

We won't provide the full semantics of how the ARGUS language works, but provide a brief write up that should give you the general idea of how the system works.

In ARGUS, distributed programs are composed of a group of **guardians**. Guardians encapsulate and control access to one or more resources and makes them available through operations called **handlers**, which are called by other guardians. Guardians contain both data and processes: process execute handlers and are spawned for each call to a handler by another guardian. Processes have access to objects that make up the state of the guardian and this is how access to objects is controlled.

Guardians contain both stable and volatile objects: volatile objects are lost when nodes running a guardian fail. When a guardian fails, the language support system is responsible for recreating the guardian with the objects that were persisted to stable storage. Stable state is versioned when modifications are performed, and volatile objects live on the heap. Guardians are created dynamically and the node where the guardian runs is specified by the programmer: guardians represent a logical node of the system where communication is performed by handlers, an abstraction of state and the physical network.

Processes in guardians execute concurrently and the language controls access to atomic objects using the locking mechanisms described above: in addition, a coroutining facility allows subactions to be further divided into concurrent executions, yielding more concurrency from the system.

ARGUS provides static type checking at compile time and loading of new guardians and handlers while the system is running: to ensure this happens safely, types must be known by the system before loading a handler that uses these types. Guardians and handlers are first class and can be used as both arguments and return values from handler invocations.

As the authors highlight, the major drawbacks of the system are in security and scheduling. ARGUS provides no mechanism for securing guardians from particular guardians or preventing nodes from launching certain types of guardians. ARGUS also has no mechanism for priority servicing of calls, backpressure, or load-shedding, leading the system susceptible to scheduling problems when overloaded.

Finally, the authors acknowledge that **concurrent programming is hard**. While ARGUS goes to great efforts to simplify the problem of maintaining consistency in a distributed application, as highlighted by the banking example, the ARGUS system made it all to easy to introduce deadlocks by having concurrent processes in a guardian, or concurrent actions, take out read/write locks in the wrong order.

3.3 Impact and Implementations

ARGUS was arguably the first system to provide a distributed programming paradigm where consistency was integrated in the language as a first class concept. ARGUS built on top of CLU, and provided a series of abstractions directly assisting programmers in writing code that maintained consistency, as the authors felt this was the main challenge in distributed programming.

ARGUS served as the platform where other ideas would blossom: both **Remote Procedure Call** was used as the primary abstraction for making remote calls, where subactions were used to mask omissions. The optimizations that **Promises** provided to pipeline RPCs and ensure ordering of requests and responses also were originally developed in the ARGUS system.

The idea of compensating transactions in ARGUS for top-level actions that can not be "recovered" also appeared later in the SAGA design [17].

Finally, languages like Bloom [1], while not providing abstractions for consistency, take an alternative approach by providing analysis tools for determining where consistency could be violated and cause program anomalies. The authors propose to plug in a distributed consensus mechanism, such as Zookeeper, to ensure consistency while performing these changes.

4 Promises

4.1 Relevant Reading

- *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, Liskov and Shrira, PLDI 1988 [30].
- *Multilisp: A language for concurrent symbolic computation*, Halstead, TOPLAS 1985 [19].

4.2 Commentary

Outside of early mentions from Friedman and Wise on a *cons* cell with placeholder values [16] and Baker and Hewitt's work on incremental garbage collection for speculative execution in parallel processes [3], *futures* originally appeared as one of the two principal constructs for parallel operations in MultiLisp. MultiLisp attempted to solve a main challenge of designing a language for parallel computation: how can parallel computation be introduced into a language in a way that fits with the existing programming paradigm. This problem is motivated by the fact that computer programmers will need to introduce concurrency into applications because automated analysis may not be able to identify all of the points for parallelism. Halstead decides there is quite a natural fit with a Lisp/Scheme: expression evaluation can be done in parallel. MultiLisp introduces two main concepts: *pcall*, to evaluate the expressions being passed to a function in parallel and introduce concurrency into evaluation of arguments to a function, and *futures*, to introduce concurrency between the computation of a value and the use of that value. Halstead also notes that futures closely resemble the "eventual values" in Hibbard's Algol 68, however were typed distinctly from the values they produced and later represented. [19]

In 1988, Liskov and Shrira introduce the concept of a *promise*: an efficient way to perform asynchronous remote procedure calls in a type-safe way [30]. Simply put, a promise is a placeholder for a value that will be available in the future. When the initial call is made, a promise is created and the asynchronous call to compute the value of the promise runs in parallel with the rest of the program. When the call completes, the value can be "claimed" by the caller.

An excerpt motivation from *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems (Liskov and Shrira, PLDI 1988)*:

> "Remote procedure calls have come to be the preferred method of communication in a distributed system because programs that use procedures are easier to understand and reason about than those that explicitly send and receive messages. However, remote

> calls require the caller to wait for a reply before continuing, and therefore can lead to lower performance than explicit message exchange."

The general motivation behind the work by Liskov and Shrira can be thought as the following critiques of two models of distributed programming.

- The Remote Procedure Call (RPC) paradigm is preferable by programmers because it is a familiar programming model. However, because of the synchronous nature of RPC, this model does not scale in terms of performance.

- The message passing paradigm is harder for programmers to reason about, but provides the benefit of decoupling of request and response, allowing for asynchronous programming and the subsequent performance benefits.

Promises attempts to bridge this gap by combining the remote procedure call style of building applications, with the asynchronous execution model seen in systems that primarily use message passing.

The first challenge in combining these two programming paradigms for distributed programming is that of order. Synchronous RPC imposes a total order across all of the calls in an application: one call will fully complete, from request to response, before moving to the next call, given a single thread of execution. If we move to an asynchronous model of RPC, we must have a way to block for a given value, or result, of an asynchronous RPC if required for further processing.

Promises does this by imagining the concept of a *call-stream*. A *call-stream* is nothing more than a stream of placeholder values for each asynchronous RPC issued by a client. Once a RPC is issued, the *promise* is considered *blocked* asynchronous execution is performed, and once the value has been computed, the *promise* is considered *ready* and the value can be *claimed* by the caller. If an attempt to *claim* the value is issued before the value is computed, execution blocks until the value is available. The stream of placeholder values serves as an implicit ordering of the requests that are issued; in the Argus [27] system that served as the implementation platform for this work, multiple streams were used and related operations sequenced together in the same stream[9].

4.3 Impact and Implementations

While promises originated as a technique for decoupling values from the computations that produced them, promises, as proposed by Liskov and

[9] Promises also provide a way for stream composition, where processes read values from one or more streams once they are *ready*, fulfilling placeholder *blocked* promises in other streams. One classic implementation of stream composition using *promises* is the Sieve of Eratosthenes.

Shrira mainly focused on reducing latency and improving performance of distributed computations. The majority of programming languages in use today by practitioners contain some notion of *futures* or *promises*. Below, we highlight a few examples.

The Oz [22] language, designed for the education of programmers in several different programming paradigms, provides a functional programming model with single assignment variables, streams, and promises. Given every variable in Oz is a dataflow, and therefore every single value in the system is a promise. Both Distributed Oz [20] and Derflow (an implementation of Oz in the Erlang programming language) [11] provide distributed versions of the Oz programming model. The Akka library for Scala also provides Oz-style dataflow concurrency with Futures.

More recently, promises have been repurposed by the JavaScript community to allow for asynchronous programs to be written in direct style instead of continuation-passing style. ECMAScript 6 contains a native Promise object, that can be used to perform asynchronous computation and register callback functions that will fire once the computation either succeeds or fails [44].

5 Emerald

5.1 Relevant Reading

- *The development of the Emerald programming language*, Black, Andrew P and Hutchinson, Norman C and Jul, Eric and Levy, Henry M, HOPL 2007 [10].
- *Distribution and Abstract Types in Emerald*, A. Black and N. Hutchinson and E. Jul and H. Levy and L. Carter, IEEE 1987 [7].
- *Emerald: A general-purpose programming language*, Raj, Rajendra K. and Tempero, Ewan and Levy, Henry M. and Black, Andrew P. and Hutchinson, Norman C. and Jul, Eric, Software Practice and Experience, 1991 [34].
- *Object Structure in the Emerald System*, Black, Andrew and Hutchinson, Norman and Jul, Eric and Levy, Henry, OOPLSA '86 [6].
- *Typechecking Polymorphism in Emerald*, Black, Andrew P. and Hutchinson, Norman, Technical Report CRL 91/1, Digital Cambridge Research Laboratory, 1991.
- *Getting to Oz*, Hank Levy, Norm Hutchinson, and Eric Jul, April 1984.

These texts, and more, are available from the languages website, http://www.emeraldprogramminglanguage.org.

5.2 Commentary

The Eden Programming Language (EPL) was a distributed programming language developed on top of Concurrent Euclid [24] that extended the existing language with support for remote method invocations. However, this support was far from ideal: incoming method invocation requests would have to be received and dispatched by a single thread while the programmer making the request would have to manually inspect error codes to ensure that the remote invocation succeeded.

Eden also provided location-independent mobile objects, but the implementation was extremely costly. In the implementation, each object was a full Unix process that could send and receive messages to each other: these messages would be sent using interprocess communication if located on the same node, resulting in latencies in the milliseconds. Eden additionally implemented a "kernel" object for dispatching messages between processes, resulting in a single message between two objects on the same system taking over 100 milliseconds, the cost of two context switches at the time. To make applications developed in EPL more efficient, application developers would use a lightweight heap-based object implemented in Concurrent Euclid (that, appeared as a single Eden object consuming a single Unix process) for objects that needed to communicate, but were located on the same machine, that would communicate through shared memory. The next problem follows naturally: the single abstraction provided resulted in extremely slow applications, so, a new abstraction was provided to compensate leaving the user with two different object models.

In a legendary memo entitled "Getting to Oz", the language designers of Eden and the soon-to-be language designers began discussions to improve the design of Eden. This new language would be entitled "Emerald"[10].

We enumerate here the list of specific goals the language designers had for Emerald, outside of the general improvements they wanted to make on Eden.

1. Convinced distributed objects were a good idea and the right way to construct distributed programs, they sought to *improve the performance of distributed objects.*

2. Objects should stay relatively cost-free, for instance, if they do not take advantage of distribution: *no-use, no-cost.*

3. *Simplify* and reduce the dual object model, remove explicit dispatching, error handling and other warts in the Eden model.

4. To support *the principle of information hiding* and have a single semantics for both large and small, local or distributed, objects.

5. Distributed programs can fail: the network can be down, a service can be unavailable, and therefore a language for building distributed applications needs to *provide the programmer tools for dealing with these failures.*

6. *Minimization of the language* by removing many of the features seen in other languages and building abstractions that could be used to extend the language.

7. *Object location needs to be explicit* even as much as the authors wanted to follow the *principle of information hiding* as it directly impacts performance. Objects should be able to move moved, but moving an object should not change the operational semantics of the language[11].

[10] As in, the Emerald city from "The Wonderful Wizard of Oz" referencing the original runtime for the Oz language "Toto", and the nickname for Seattle.

[11] This dichotomy is presented as the *semantics* vs. the *locatics* of the language and the authors soon realized that one aspect of the language influenced both of these: failures.

5.3 Impact and Implementations

The technical innovations for Emerald, a system that was under primary development from 1983 to 1987 (and later continued by various graduate students) are numerous. We highlight a few of the most important technical innovations below:

1. Emerald presented a *single object model* for both distributed and local objects. Each object has a globally unique identifier, internal state, and a set of methods that could be invoked. These objects could run in their own process, if necessary, or not. (In fact, objects encapsulated their processes and launched them on object invocation.)

2. Objects could exist with different implementations in this unified model: *global* objects, or objects that could be accessed either locally or remote; *local* objects, that were optimized for local access only as determined as best as possible at compile time; and *direct*, or objects that represented primitive types such as integers, booleans, etc.

3. Emerald was a statically-typed language, that had dynamic type checking for objects that were received over the wire. This was achieved using a notion of *protocols* and *conformity*-based typing. Dynamic type checking would be performed by ensuring types at runtime were compatible based on their interfaces. This was done by forming a type lattice and computing both the *join* and *meet* based on the *abstract*, or the compile-time provided type, and the *concrete* implementation that were provided at runtime.

4. Emerald's type system also provided *capabilities*, where types could either be *restrict*ed to a higher type in the type lattice, or *view*ed at a lower type in the type lattice.

5. Synchronization between processes in Emerald was achieved using *monitors* to achieve mutual exclusion with condition signaling and waiting [23].

6. Mobility in Emerald was provided using explicit placement primitives. Processes could be *moved* to a new location, *fix*ed at a precise location, and *locate*d. Emerald also provided two new parameter evaluation modes based on mobility: *call-by-move*[12], or move the parameter object to the invocation location, and *call-by-visit*, by remote access of the parameter object from the invocation's location. When objects were moved, the old placement would store a *forwarding address* that would be used to route messages forward; timestamps were used to

[12] This is a departure from systems like Argus [27] that assumed all arguments were passed using call-by-value.

detect routing loops and reference the most recent object and to avoid stable storage of the forwarding addresses, reliable broadcast was used to find lost pointers.

7. Errors related to network availability were not considered exceptions; therefore special notation for handling these errors was provided to the programmer.

Emerald (and Eden's) influence on programming languages throughout the history of programming languages is paramount. Emerald specifically innovated in two main areas: distributed objects and type systems, which is interesting because the innovations in type systems were only done to support the development of distributed objects in the Emerald system.

The idea of type *conformity* over a type lattice with both *concrete* and *abstract* types influenced the further development of *protocols*, mechanisms to specify the external behavior of an objects, in the ANSI 1997 Smalltalk standard.

Developers of the Modula-3 Network Objects [4] system took what they felt was the most essential and the best of both the Emerald and SOS [36] systems. This system forewent the mobility of objects in favor of marshalling. In the author's own words:

> "We believe it is better to provide powerful marshaling than object mobility. The two facilities are similar, because both of them allow the programmer the option of communicating objects by reference or by copying. Either facility can be used to distribute data and computation as needed by applications. Object mobility offers slightly more flexibility, because the same object can be either sent by reference or moved; while with our system, network objects are always sent by reference and other objects are always sent by copying. However, this extra flexibility doesnt seem to us to be worth the substantial increase in complexity of mobile objects." [10]

Both Java's RMI system and Jini (and their predecessor OMG's CORBA) were influenced by Emerald as well, however not without a fierce discussion on the merits of distinguishing between remote and local method invocations, motivated by legendary technical report [25] by Sun Microsystems' research division[13].

Jim Waldo, author of the aforementioned technical report, writes:

[13] While we acknowledge the lineage here beginning with systems like Eden and Emerald, the majority of the criticisms of this technical report are targeted towards OMG's CORBA system.

> "The RMI system (and later the Jini system) took many of the ideas pioneered in Emerald having to do with moving objects around the network. We introduced these ideas to allow us to deal with the problems found in systems like CORBA with type truncation (and which were dealt with in Network Objects by doing the closest-match); the result was that passing an object to a remote site resulted in passing [a copy of] exactly that object, including when necessary a copy of the code (made possible by Java bytecodes and the security mechanisms). This was exploited to some extent in the RMI world, and far more fully in the Jini world, making both of those systems more Emerald-like than we realized at the time." [10]

The authors eventually come to a similar conclusion to many distributed systems practitioners today and other critics in their research area: that availability, reliability, and the network remain the paramount challenges and add fuel to the fire against any location-transparent semantics provided by mobile objects. These still remain as much of a challenge for the developers of distributed programs today, as they did in 1983 at the start of the development lineage from Eden to Emerald:

> "Mobile objects promise to make that same simplicity available in a distributed setting: the same semantics, the same parameter mechanisms, and so on. But this promise must be illusory. In a distributed setting the programmer must deal with issues of availability and reliability. So programmers have to replicate their objects, manage the replicas, and worry about 'one copy semantics'. Things are not so simple any more, because the notion of object identity supported by the programming language is no longer the same as the applications notion of identity. We can make things simple only by giving up on reliability, fault tolerance, and availability but these are the reasons that we build distributed systems." [10]

The authors of Eden and Emerald express an extremely interesting point early on in their paper on the history of Emerald [10]: the reason for the poor abstractions requiring manual dispatch of method invocations to threads, and the explicit error handling from network anomalies, was because as researchers working on a language, *they were not implementing distributed applications themselves.* To quote the authors:

> "Eden team had real experience with writing distributed applications, we had not yet learned what support should be provided. For example, it was not clear to us whether or not each incoming call should be run in its own thread (possibly leading to

> excessive resource contention), whether or not all calls should run in the same thread (possibly leading to deadlock), whether or not there should be a thread pool of a bounded size (and if so, how to choose it), or whether or not there was some other, more elegant solution that we hadnt yet thought of. So we left it to the application programmer to build whatever invocation thread management system seemed appropriate: EPL was partly a language, and partly a kit of components. The result of this approach was that there was no clear separation between the code of the application and the scaffolding necessary to implement remote calls." [10]

I will close this section on Emerald with a quote from the authors.

> "We are all proud of Emerald, and feel that it is one of the most significant pieces of research we have ever undertaken. People who have never heard of Emerald are surprised that a language that is so old, and was implemented by so small a team, does so much that is 'modern'. If asked to describe Emerald briefly, we sometimes say that its like Java, except that it has always had generics, and that its objects are mobile." [10]

6 Hermes

6.1 Relevant Reading

- *Implementing Location Independent Invocation*, Black, Andrew P and Artsy, Yeshayahu, IEEE Transactions on Parallel and Distributed Systems, 1990 [9].
- *Customizable and extensible deployment for mobile/cloud applications*, Zhang, Irene and Szekeres, Adriana and Van Aken, Dana and Ackerman, Isaac and Gribble, Steven D and Krishnamurthy, Arvind and Levy, Henry M, 2014 [46].

6.2 Commentary

The general idea behind the Remote Procedure Call (RPC) paradigm is that it supports the transfer of control between address spaces. This paradigm allows programmers to write distributed applications without having to have knowledge of data representations or specific network protocols. Even though we know that there is quite a bit semantically different between remote and local calls [25, 9], the authors posit that the most fundamental difference is that of *binding*, or, how to figure out which address space to direct the call to.

Traditionally, this has been done one of two ways: *default or automatic* binding where the RPC system makes the choice for the programmer; or *clerks*, an application specific module used for determining where the place the call. Default binding is fairly straightforward when there is only one server (or a group of semantically equivalent servers) to service the request. Clerks are fairly expensive, as one must be written for each type of request that needs to be serviced. If the service the RPC call is being made to is *pure*, for instance providing as fast Fourier transform as the authors put it, it is easy to choose automatic binding to select a server based on latency or availability. However, it is more challenging if services host application data. In their example, they consider an employee directory at Digital where application data is partitioned by company, and further by other groupings. If this mapping changes infrequently, a static mapping can be distributed to all of the clients; but, what happens if objects are mobile and this changes more frequently?

One of the fantastic things about this paper is how forward thinking the design is for an actual industrial problem at Digital Equipment Corporation. I consider this one of the early versions of what we now call an "industry" research report, even though the system never was productized and the work was mainly performed by researchers in a lab. The application deals with expense vouchers for employees: each form needs to be filled in by an employee, approved by various managers, filed, and eventually results in a payout of actual cash. Each of the managers that are involved in approving the form may be located in different buildings in different continents. The application design assumes Digital's global network of 36,000 machines and assumes that centralizing the records for each form in a centralized database is infeasible. Instead, the design is based on mobile objects for both data and code; forms should be able to move around the network as required by the application.

The Hermes system is broken into three components: a naming service, a persistent store known as a collection of *storesites*, and routing layer that sits above the RPC system. Each object in the system is given a globally unique identifier, a source *storesite* and a *temporal address descriptor* or *tad*. The *temporal address descriptor* is a pair composed of a Hermes node identifier and a monotonically advancing timestamp: this pair represents where an object is located at a given time. This information is also persisted in the objects *storesite*. As objects move around the network, the *tad* is updated at the source node and 2PC used to coordinate a change with the record at the objects's *storesite*.

When remote procedure calls are issued, the callee attempts to issue the call locally if the objects is local. If not, and a forwarding pointer, or *tad* exists, the message is routed to that node. Forwarding pointers are followed a number of times until a maximum hop count is reached; at this point the call is returned to the callee who begins the process again with the last known forwarding pointer. Along the path of forwarding, the *tad* is updated as each hop occurs, reducing the number of hops needed for the next request through that node. This is possible because of the monotonicity of the temporal addresses.

If a node has no local knowledge of where that object is, either because it is not running locally or because there exists no temporal address, a request is made to the naming service to request the *storesite* for the object, and the address of the current location retrieved from the *storesite*.

However, in this model failures may occur. If the RPC arrives at the destination of the object and the call invoked and completed, but the response packets dropped, what happens? In this case, an invocation sequencer is required to ensure that the operation only performed if it has not previously completed. The authors suggest developers write operations that are idempotent, to ensure they can be replayed without issue or additional overhead.

6.3 Impact and Implementations

Both the Eden [8] and Emerald [10] programming languages both had notions of distributed objects. Eden used hints to identify where to route messages for objects, but timed them out quickly. Once timed out, a durable storage location called a *checksite* would be checked, and if that yielded no results, broadcast messages would be used. Emerald, a predecessor to

Hermes, used forwarding addresses, but used a broadcast mechanism to find objects when forwarding addresses were not available. In the event the broadcast yielded no results, an exhaustive search of every node in the cluster was performed. All of these decisions were fine for a language and operating system designed mainly for research.

Emerald was more advanced in several ways. Emerald's type system allowed for the introduction of new types of objects, whereas the Hermes system assumed at system start all possible object types were known to the system. Emerald could also migrate processes during invocation, something that the Hermes system could not.

While the system could tolerate some notion of failures while following forwarding addresses, by resorting to usage of the information located at the *storesite*, the system had no way to prevent issues with partitions: where an invocation may fail because the object is inaccessible. However, given the relative independence of objects in the system, this would only affect objects (or users) located on the partitioned machine.

The design of Hermes was completed in a year and a half, written in Modula-2+, and was demonstrated functional in the laboratory with a LAN composed of a small number of nodes. According to one of the authors of the paper, the system never was turned into a product, mainly because Digital did not have a team at the time responsible for turning advanced research projects into actual distributed systems products[14].

The Sapphire [46] system presented at OSDI '14 bears a similar resemblance to the Hermes system and its Emerald roots. While Sapphire focuses on the separation of application logic from deployment logic through the use of interfaces and interface inheritance in object-oriented programming languages, Sapphire uses many of the techniques presented in both Emerald and Hermes: transparent relocation based on annotations or for load balancing; location independent method invocation through the use of forwarding pointers, and fallback to a persistent data store to find the canonical location of a particular object.

Today, idempotence [21] has been a topic of study in distributed systems, as it assists in designing deterministic computations that must happen on unreliable, asynchronous networks; a place where it is impossible to reliably detect failures [15]. Shapiro *et al.* [37] propose the use of data structures that are associative, commutative, and idempotent as the basis for shared state in distributed databases. Meiklejohn and Van Roy [32] propose similar for large-scale distributed computations; whereas Conway *et al.* [13] propose similar for protocol development. Lee *et al.* propose a system called RIFL for ensuring exactly-once semantics for remote procedure calls by uniquely identifying each call and fault-tolerant storage of the results [26].

[14] Andrew P. Black, personal communication.

Acknowledgements

I would like to thank the following people who provided helpful editing with the material: Sean Cribbs, Stuart Marks, and Steve Vinoski.

References

[1] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260. Citeseer, 2011.

[2] J. Armstrong. erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[3] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977.

[4] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 217–230, New York, NY, USA, 1993. ACM.

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[6] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 78–86, New York, NY, USA, 1986. ACM.

[7] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan 1987.

[8] A. P. Black. Supporting distributed applications: Experience with eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 181–193, New York, NY, USA, 1985. ACM.

[9] A. P. Black and Y. Artsy. Implementing location independent invocation. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):107–119, 1990.

[10] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The development of the emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 11–1. ACM, 2007.

[11] M. Bravo, Z. Li, P. Van Roy, and C. Meiklejohn. Derflow: Distributed deterministic dataflow programming for erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 51–60, New York, NY, USA, 2014. ACM.

[12] K. Claessen and H. Svensson. A semantics for distributed erlang. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 78–87. ACM, 2005.

[13] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[14] J. A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, 1979.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[16] D. P. Friedman and D. S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, April 1978.

[17] H. Garcia-Molina and K. Salem. *Sagas*, volume 16. ACM, 1987.

[18] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990.

[19] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[20] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of distributed oz. In *Proceedings of the second international symposium on Parallel symbolic computation*, pages 176–187. ACM, 1997.

[21] P. Helland. Idempotence is not a medical condition. *Queue*, 10(4):30:30–30:46, Apr. 2012.

[22] M. Henz, G. Smolka, and J. Würtz. Oz-a programming language for multi-agent systems. In *IJCAI*, pages 404–409, 1993.

[23] C. A. R. Hoare. *Monitors: An operating system structuring concept.* Springer, 1974.

[24] R. C. Holt. A short introduction to concurrent euclid. *ACM Sigplan Notices*, 17(5):60–79, 1982.

[25] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant. A note on distributed computing. 1994.

[26] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86. ACM, 2015.

[27] B. Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[28] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of argus. *ACM SIGOPS Operating Systems Review*, 21(5):111–122, 1987.

[29] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):381–404, 1983.

[30] B. Liskov and L. Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM, 1988.

[31] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in clu. *Communications of the ACM*, 20(8):564–576, 1977.

[32] C. Meiklejohn and P. Van Roy. Lasp: a language for distributed, eventually consistent computations with crdts. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 7. ACM, 2015.

[33] H. Miller, P. Haller, and M. Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP 2014–Object-Oriented Programming*, pages 308–333. Springer, 2014.

[34] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software: Practice and Experience*, 21(1):91–118, 1991.

[35] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Innovations in internetworking. chapter Design and Implementation of the Sun Network Filesystem, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988.

[36] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. Sos: An object-oriented operating system-assessment and perspectives. *Computing Systems*, 2(4):287–338, 1989.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types.* PhD thesis, Inria–Centre Paris-Rocquencourt, 2011.

[38] H. Svensson and L.-A. Fredlund. A more accurate semantics for distributed erlang. In *Erlang Workshop*, pages 43–54. Citeseer, 2007.

[39] H. Svensson and L.-Å. Fredlund. Programming distributed erlang applications: Pitfalls and recipes. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 37–42. ACM, 2007.

[40] A. S. Tanenbaum and R. van Renesse. *A critique of the remote procedure call paradigm.* Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1987.

[41] S. Vinoski. It's just a mapping problem [computer application adaptation]. *Internet Computing, IEEE*, 7(3):88–90, 2003.

[42] S. Vinoski. Convenience over correctness. *Internet Computing, IEEE*, 12(4):89–92, 2008.

[43] E. F. Walker. Orphan detection in the argus system. Technical report, DTIC Document, 1984.

[44] Wikipedia. Futures and promises — wikipedia, the free encyclopedia, 2016. [Online; accessed 4-March-2016].

[45] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java tm system. In *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)-Volume 2*, pages 17–17. USENIX Association, 1996.

[46] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 97–112, 2014.