Programming Models and Languages for Distributed Computation

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

March 6, 2016

Contents

1 Promises

1.1 Relevant Reading

- *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, Liskov and Shrira, PLDI 1988 [21].
- *Multilisp: A language for concurrent symbolic computation*, Halstead, TOPLAS 1985 [12].

1.2 Commentary

Outside of early mentions from Friedman and Wise on a *cons* cell with placeholder values [11] and Baker and Hewitt's work on incremental garbage collection for speculative execution in parallel processes [1], *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. [12]

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 [21]. 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 [20] system that served as the implementation platform for this work, multiple streams were used and related operations sequenced together in the same stream[1].

1.3 Impact and Implementations

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

[1] 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 [15] 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 [13] and Derflow (an implementation of Oz in the Erlang programming language) [8] 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 [26].

2 Emerald

2.1 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 [7].

- *Distribution and Abstract Types in Emerald*, A. Black and N. Hutchinson and E. Jul and H. Levy and L. Carter, IEEE 1987 [4].

- *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 [23].

- *Object Structure in the Emerald System*, Black, Andrew and Hutchinson, Norman and Jul, Eric and Levy, Henry, OOPLSA '86 [3].

- *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`.

2.2 Commentary

The Eden Programming Language (EPL) was a distributed programming language developed on top of Concurrent Euclid [17] 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"[2].

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.

[2] 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.

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[3].

2.3 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.

[3]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. Synchronization between processes in Emerald was achieved using *monitors* to achieve mutual exclusion with condition signaling and waiting [16].

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*[4], 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 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 [2] system took what they felt was the most essential and the best of both the Emerald and SOS [24] 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

[4]This is a departure from systems like Argus [20] that assumed all arguments were passed using call-by-value.

> always sent by copying. However, this extra flexibility doesnt seem to us to be worth the substantial increase in complexity of mobile objects." [7]

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 [18] by Sun Microsystems' research division[5].

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

> "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." [7]

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

[5] 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.

can make things simple only by giving up on reliability, fault tolerance, and availability but these are the reasons that we build distributed systems." [7]

The authors of Eden and Emerald express an extremely interesting point early on in their paper on the history of Emerald [7]: 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." [7]

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." [7]

3 Hermes

3.1 Relevant Reading

- *Implementing Location Independent Invocation*, Black, Andrew P and Artsy, Yeshayahu, IEEE Transactions on Parallel and Distributed Systems, 1990 [6].

3.2 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 quit a bit semantically different between remote and local calls [18, 6], 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.

3.3 Impact and Implementations

Both the Eden [5] and Emerald [7] 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[6].

Today, idempotence [14] 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 [10]. Shapiro *et al.* [25] 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 [22] propose similar for large-scale distributed computations; whereas Conway *et al.* [9] 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 [19].

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] Andrew P. Black, personal communication.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

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

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

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

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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