Towards Fearless Distributed Computing

Marc Stiegler

HP Laboratories

HPL-2009-258

Keyword(s):
Distributed programming, SCALA, message passing

Abstract:
Moving from sequential programming to concurrent, distributed programming exposes the programmer to diverse new hazards. Here we list some of the dangers, and look at the results when they are eliminated by using "relentlessly reliable references". Preliminary experience suggests that, given such "r-r-refs", programming of distributed systems is "fearless", i.e., though the hazards of writing a concurrent program are different from those in writing a sequential program, they may not be enough greater to induce greater fear. The consequence is that programmers can confidently write programs in an intrinsically concurrent, distributed style, from the first day of the first prototype. We hypothesize that such programs will scale well, exploiting both multicore chips and multi-node computing systems naturally and reliably.

External Posting Date: October 21, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: October 21, 2009 [Fulltext]

Towards Fearless Distributed Programming

Marc Stiegler

HP Labs

marc.d.stiegler@hp.com

Abstract

Moving from sequential programming to concurrent, distributed programming exposes the programmer to diverse new hazards. Here we list some of the dangers, and look at the results when they are eliminated by using "relentlessly reliable references". Preliminary experience suggests that, given such "r-r-refs", programming of distributed systems is "fearless", i.e., though the hazards of writing a concurrent program are different from those in writing a sequential program, they may not be enough greater to induce greater fear. The consequence is that programmers can confidently write programs in an intrinsically concurrent, distributed style, from the first day of the first prototype. We hypothesize that such programs will scale well, exploiting both multicore chips and multi-node computing systems naturally and reliably.

Problem

Imagine that a friend informed you he'd found a marvelous new programming environment, JD++. He explains that JD++ has all the following enhancements to Java, all related to the intriguing new things that can happen when you invoke a method on an object:

- Sometimes the target will move while its method is being invoked, and the invocation will fail.
- Sometimes the target will have its methods called from surprising, unintended invokers, breaking encapsulation if the target object was supposed to be a private instance variable. All variables in JD++ are in effect public statics, i.e., global variables.
- Sometimes a method on some object other than the intended target will get invoked, either accidentally or through intentional redirection.
- Sometimes you will hang, waiting forever for the return – the target needs to ask you a question before returning a value from the invocation, but in JD++ the code is non-reentrant.
- Sometimes the recipient is successfully invoked, but the runtime system loses the returned value. Not only has the answer been lost, but if the message is not idempotent (as with an "increment" message), you face an intractable programming decision on how to handle the failure: you do not know if the invocation was never processed (in which case you must invoke the method again) or if your invocation was processed before the return value was lost (in which case you must *not,* for the sake of correctness, invoke the method again, though you must somehow ask for the answer that was generated the last time you invoked it).

Most programmers would probably look askance at a friend who recommended switching to JD++. And yet, when we move from writing sequential programs to writing distributed programs, *we are immediately embracing these and/or related hazards*. The fanciful JD++ described above is simply normal Java being used in a distributed computing context.

Given all these risks, who can be surprised that programmers hold tight to sequential programming, and surrender to distributed computation only under dire circumstances?

Solution: Relentlessly Reliable References

All the problems described above can be thought of as failures of referential integrity: the reliability of a simple OO reference is hard to duplicate in a distributed system. Here we present the elements of the *relentlessly reliable reference* (hereafter referred to as an r-r-ref) that, with one caveat, eliminates all these problems. All these elements can be combined into a runtime support platform, putting distributed computation on a more level playing field with sequential programming. We examine one set of elements that, taken together, deliver such referential integrity across a distributed system.

Webkeys

Let us begin with the webkey[Karp09]. When integrated with the other features of the waterken platform[Close04], webkeys become r-r-refs. The webkey is a normal url that has the following format:

https://y-zbeou4362mmkqgwk.yurl.net/-/onemailbox/#63qb7bg4wmueyc

Because the webkey uses an unguessable page name (the red highlight, "63qb7bg4wmueyc"), the only entities that can reach the webkey's designated object are objects that have been handed the reference: private objects are private even if they are distributed, and they only become global if explicitly placed in a global registry. Because the webkey is encrypted using https protocol (the green highlight), there can be no interception of the message, either accidentally or on purpose, by another object. And because the domain name contains the fingerprint of the public key of the subsystem hosting the recipient (blue highlight, "zbeou4362mmkqgwk"), the runtime platform can ensure that the message is not delivered to anyone other than the intended target.

The webkey by itself, with these three features, eliminates two of the distributed system hazards: they ensure that no one other than the intended recipient will get the message, and they enable the creation of distributed objects that follow the conventional rules of object-oriented private instance variables: the objects referenced by private instance variables are visible only to the owner and to those entities to whom the owner explicitly hands over a reference.

Actor Location Service

How do we ensure that the message can be delivered even if the recipient moves? The waterken platform works with an actor location service: when a waterken server launches, it examines its own IP address and registers that IP address with the locator (in the example above, the actor location service is hosted at yurl.net). Hence, even if the server hosting the object moves to a new IP address in a new domain, senders can find it. Distributed system services can even use DHCP to assign IP addresses: those services will still be found reliably. Even if another service is launched at the original IP address, everything works correctly – the sender first goes to the original address, finds a server, challenges that server to identify itself as owner of the public/private key pair for the webkey fingerprint, and when that fails, the sender goes to the actor location service to find the new IP address. This is all handled by the platform runtime: the actors and objects in the application do not know or care, and neither does the programmer.

Promise Pipelining

How do we ensure that the recipient can ask the sender a question as part of the process of computing the new answer? In sequential terminology, how do we ensure the code is re-entrant?

The waterken platform uses the promise pipelining paradigm of message sending[Miller04, Stiegler04]. As a consequence, the sender does not block and wait for the recipient's answer. Nor does it spawn new threads to respond to additional invocations, which would confer all the traditional data race and lock contention risks. Rather, the sender is still available to the recipient (and to all other clients as well) in a disciplined, deadlock-free and fine-grain-data-race-free fashion, to respond to questions while it awaits an answer.

This is best seen with some example code. Here, a control program is asking a remote car's engine about its temperature. If the temperature exceeds the max, the controller shuts down the car. We will show this example, and other examples, in both Java and Scala. In this example, the code may be translated to English roughly as, "when I get the car's engine's temperature, compare it to the max temperature, and if it is too high, shut down the car. Meanwhile, while I'm waiting for the temperature data, continue doing other things."

Scala	Java

```
w.when(remoteCar.getEngine().getTemp()) {
   temp => if (temp > ENGINEMAX) {
      remoteCar.shutDown()
   }
} default()
println("this runs immediately, no waiting")
```

```
_.when(remoteCar.getEngine().getTemp(),
   new Do<Integer, Void>() {

   public Void fulfill(Integer temp) {
      if (temp > ENGINEMAX) {remoteCar.shutDown();}
      return null;
   }
});
System.out.println("this runs immediately, no waiting"):
```

Let us look briefly at how this promise pipelining example works. We ask the car for its engine in the conventional OO fashion: remoteCar.getEngine(). In sequential programming, the program would block at this point and wait for the reference to the engine object to be returned by the car. However, since the car is remote, in a promise pipelining system the requestor does not wait for the round trip from the far node. Rather, a *promise* is immediately constructed, which will be *fulfilled* later, when the engine reference is received. In the meantime, the requestor continues on to execute its next expression without waiting.

But in this example the next expression is a message to the engine asking for the temperature. We only have a promise for the engine, but that is enough. We can send a message to the promise, to be delivered once the promise is fulfilled. With pipelining, the message for the (not-yet-identified) engine is actually sent immediately to the server hosting the car, with the explanation that this message should be delivered to the engine that fulfills the promise. So no round trips are needed: the engine gets the message before its reference is ever returned to the requestor.

The getTemp() message sent to the engine's promise immediately gives us another promise for the temperature. Now we will actually take action locally with the temperature result when its promise is fulfilled. We set up a passage of code to do something with the fulfillment -- we create a when-block that will fire "when" the promise is fulfilled. In both the Java and Scala versions, the when-block will, when the temperature is received, load the fulfilled value into the variable "temp". The when-block checks to see if the temperature is excessive, and shuts the car down if it is.

Note that although the when-block waits for the temperature promise to be fulfilled, the main body of the controller continues on to its next statement, and provides services to its clients without regard to the pending state of the promise for the temperature. As a consequence, the remote car could, if needed, ask the controller a question and get an answer back before responding to the temperature request.

Orthogonal Persistence, Idempotent Protocol, and Distributed Consistency

To answer the last question – how do we ensure we get an answer back even though the network may go down temporarily between our nodes, and how do we ensure that the recipient processes each message exactly once – we use an idempotent message protocol combined with orthogonal persistence to ensure distributed consistency.

The actual algorithm followed in processing messages in a waterken platform is somewhat complex. Conceptually, however, if the same message is received a second time, the receiving platform does not even inform the target object of the message: the platform can simply send the answer computed on first reception back to the sender. Because the sender's host knows there is no risk in resending, if the sending host is unable to connect to the recipient host, or if the connection fails in mid-operation, the sending host simply resends the message (this resending is the feature that gives these references their "relentless" nature). Neither the sender nor the recipient knows or cares that their runtime platforms are handling these network partition and offline host problems on their behalf. Also, the sender's ordering of messages to the specified target's host is invariant: alice's second message to carol will not be processed until after alice's first message has been processed (this appears as a partial ordering on carol's side: messages from bob may be interleaved with messages from alice).

Meanwhile, the hosts are also automatically checkpointing their objects and message queues. At the conclusion of the event initiated by the sent message, the recipient host checkpoints the recipient and the messages that the recipient generated during message processing – no new messages are actually sent until after this event has finished and after the messages to be sent and the recipient's new state and return value have been reliably stored.

The consequence of integrating orthogonal persistence with retriable idempotent messaging is that the system achieves eventual distributed consistency: if all inputs were to stop, eventually all the messages would complete, the promises would resolve, and the whole system would achieve a consistent state.

Caveat

There remains one circumstance under which a message is not delivered, a circumstance that makes a distributed application different from a sequential program. If the host for a recipient is shut down and never restarted, the message will never be delivered. In the case of hosts distributed to nodes in a data center with a central administrator, even this situation can be reliably eliminated – part of the administrator's job can be to restart the hosts, even if the hardware upon which the hosts ran is removed.

But even in the more general case of many distributed nodes in many different administrative domains, the permanent shutdown of a node generally means that the recipient (and its messages) have become too unimportant to be worth maintaining. So the failure to deliver the message will often not be a problem. If it is a problem, the developer does need to take an action that is not normal for the sequential programmer – he sets a timeout on the pending promise, and takes alternative action if the fulfillment does not occur in a timely fashion.

In either case, despite the recipient's absence, the rest of the system continues to function.

Issues

Even with r-r-refs, there are hazards in distributed programming not found in sequential programming. However, there are also hazards in sequential programming not found in distributed programming. Once the problems identified above have been eliminated, it becomes less clear whether the hazards in distributed systems or the hazards in sequential systems are more worrisome. Here we give several examples of sequential programming hazards, followed by several examples of remaining distributed programming hazards, for consideration.

Sequentiality Hazards

The standard sequential observer pattern gives an example of interesting risk in sequentiality[Lee06]. In this example, a group of listeners are being notified that the observed value has changed:

```
public void setValue(Object newValue) {
   myValue = newValue;
   for (int i = 0; i < myListeners.length; i++) {
      myListeners[i].valueChanged(newValue);
   }
}
```

In this ordinary example, there is an inappropriate sequential dependency among the listeners. The risk becomes evident when one considers what happens if one of the listeners throws an exception – the loop notifying the listeners is immediately exited, and listeners later in the list are not notified. This is not a common failure, but it can have severe consequences.

In a promise pipelined system, if the listeners are in separate actors, the notification loop does not wait for one message to be delivered before sending the next: all the messages are sent before any replies from those messages are processed. Hence an exception thrown by one listener cannot impact the ability of another listener to receive the notification. The Java code on a waterken platform is identical to the sequential code shown above, but does not include the hazard (if the notifier wanted to be informed of failures of the listeners, it could do so by setting up when-blocks on the promises being returned from the valueChanged() messages, which would be comparable to setting up a try/catch block around the message in a sequential system).

Sequentiality bugs often result in significant denial of service failures. For example, in some versions of Linux, an SMTP server will be launched as a sequential part of the boot process. The SMTP server will wait for a network connection to come alive, halting the boot process in its tracks. Such Linux systems, when disconnected from the network, will never boot.

Another similar problem can be found in traditional SMTP-based mail processing software. When receiving a large email attachment, all other email messages must wait for the attachment to be received. If the user is using a low bandwidth low reliability connection, the MTBF of the connection may be shorter than the amount of time needed to receive the file, resulting in a system that is permanently stuck – the user can be reduced to calling his mail administrator to kill the message so other email can arrive.

This stands in stark contrast to the mail system embedded in the SCoopFS system[Karp08] built using r-r-refs. With SCoopFS, each sender has a separate reference to the in-box. Messages from multiple senders continue to arrive in parallel while a large attachment is being processed. Indeed, the attachment itself, having been broken into blocks for transmission, is automatically checkpointed in pieces during reception, so that even if the connection is repeatedly dropped, the attachment will still eventually be received intact without having to resend previously received blocks.

Perhaps the greatest weakness of sequential programming for reliable systems is that there is no graceful degradation. In most cases, a failure of any part causes the whole system to fail; a requirement to upgrade any part requires a shutdown of every part.

Remaining Distributed Hazards

Datalock

A problem in promise-like concurrency paradigms is datalock [Miller02]. It is possible to set up when-blocks that wait for each other forever, each resolving the other's promise only when its own promise is fulfilled. While it is easy to construct such intertwining when-blocks artificially, it is quite rare in practice. The reason is that the actors that created the when-blocks remain fully responsive -- only the expressions inside the when-blocks wait for a fulfillment. This is quite different from deadlocks, which cause whole threads and processes to wait forever, which can in turn lockup yet more threads and processes as they either wait for the same lock, or wait for the currently deadlocked threads to deliver results.

Both theory [Miller06] and experience suggest that datalock is a problem of limited significance. During the development of the SCoopFS system no datalock bug ever occurred. The most powerful example was the development of EC Habitats [Farmer04]. This massive multiplayer online world had over a million lines of promise-pipelining code. If any datalocks were ever created, they did not survive long enough to receive management attention.

Debugging

Even with r-r-refs, debugging a distributed system remains difficult. One technique that can be helpful is to load all the actors into a single machine for debugging on a regression test; this way one can have debuggers conveniently open on all elements of the system at the same time. And debugging tools that exploit the nature of promise pipelining are under development [Stanley09]. But today, debugging is still too often dependent on running single-node debuggers and log files to describe the inter-node interactions.

System upgrade

During the early rapid prototyping phase of sequential application development, it is normal to throw away all the data created with earlier versions of the application rather than try to "save" the data by upgrading the data as the application is upgraded. As the work proceeds, and people use the prototype to create real data with real value, the development process must slow down and even avoid some enhancements in order to support upwards compatibility with existing data.

There are ways in which such upwardly compatible evolution is made simpler by distributing the application. One can upgrade just the data associated with the nodes running modified subsystems. But distributing the work among multiple actors increases the number of kinds of reorganization that may be required. Notably, it can become necessary to split the services in a single actor into two actors, and redistribute services from one actor to other actors. The overall result is that it may be harder to upgrade a waterken application while maintaining backwards compatibility with existing data.

The orthogonal persistence of the waterken platform uses Java serialization. In general, upgrades that would be easy for a sequential program that used Java serialization to store its state will be easy in the distributed system as well. But upgrades that would be hard with such a serialized store in the sequential context can be hard indeed when redistributing is required.

Out of Order Bugs

Perhaps the most serious concern remaining in an r-r-ref system is out of order arrival bugs. Such bugs are the inverse of the bugs specific to sequential programs: just as bugs can arise from having too many sequentiality constraints, bugs can arise from having too few sequentiality constraints.

One clear example is the deposit/withdrawal problem on a checking account: Bob sends a deposit to the bank while at the same time sending a cashable check to Alice. A sequential programmer would write the following code to ensure that the deposit arrives before the check:

```
bank.deposit(money);
alice.accept(check);
```

If the programmer were to reverse these statements, it would ensure that the withdrawal occurred before the deposit, so the programmer must take some care, even in the sequential case, to get this right.

In the promise pipelined case, the ordering of the statements would make no difference. The deposit would be more likely to arrive at the bank first because it has a shorter trip, but there is no guarantee. The simplest solution for the programmer to ensure the deposit arrives first would be to put the transmission of the check in a when-block:

Scala	Java

`w.when(bank.deposit(money)) {`
`    _ => alice.accept(check)`
`} default()`
	`_.when(bank.deposit(money),`
	`    new Do<Integer, Void>() {`
	
	`    public Void fulfill(Integer amount) {`
	`        alice.accept(check);`
	`        return null;`
	`    }`
	`});`

Out of order bugs can be hard to discover and difficult to reproduce. There are mitigating factors:

- Waterken actors are coarse-grained. Whereas actor languages like Pict make every function a separate concurrent actor, and whereas actor languages like Erlang make every object a separate concurrent actor, a waterken actor is better thought of as an assembly of objects. These objects within a single actor can communicate with one another using traditional, sequential method calls. So objects with strong sequential interdependencies are often placed in the same actor, eliminating the concurrency hazards. In effect, this often allows the programmer to choose sequentiality where it makes the most sense, and concurrency where it does not.
- Having made the actors coarse-grained, the remaining ordering bugs are roughly the same as the ordering hazards that non-programmers encounter using a combination of telephones and voicemail for coordination. Such problems do not in general incur special levels of fear among phone users.

Nevertheless, out of order bugs are a serious hazard, and weigh heavily when comparing the hazards of sequential programming to those of r-r-ref programming.

Consequences of Relentlessly Reliable References

Waterken applications, with distributed object references that are relentlessly reliable, are themselves robust in the face of diverse environmental and internal failures. There is no special case software for shutting down a node: simply kill the power, and let the checkpointing and retriable idempotent message passing ensure that the system picks up where it left off. Nodes can fail and revive in any order at any time; the system degrades gracefully while subsystems are offline, continuing to process events that do not depend on the failed nodes while awaiting their revival. In some cases, failures that would catastrophically terminate a traditional sequential program with significant data loss will produce only an isolated recoverable failure in a distributed r-r-ref application.

R-r-refs also form the basis for implementing the object capability security paradigm. The properties of an object-capability system are [Stiegler07]:

- Objects are encapsulated.
- The *only* way for an object to influence the world outside itself is to send messages on references.
- The creator of an object can deny that object access to anything else simply by not providing references to anything else.

At the distributed layer of abstraction in which the subsystem of objects on one host interacts with the subsystem of objects on another host, the r-r-ref implements these properties:

- Private instance variables of objects on the remote host stay private, even if those variables are references to objects on other hosts. Encapsulation is preserved.
- The only way to communicate with an object on another host is via the r-r-ref.
- An object created on one host cannot access any object on another host that has not been explicitly provided to the first host. This is weaker than the object capability requirement, since it does not enable full confinement (though full confinement can be achieved by using the Joe-E verifier [Finifter08] in conjunction with waterken).

Though r-r-refs do not supply full object-capability security, they supply enough of the properties that valuable security characteristics emerge. While a secure distributed system will still require extra work to compose patterns of secure cooperation[Stiegler07b], subsystems can come to life in the first prototype properly secured. For example the following object implements a money exchange system (derived from the purse protocol [Miller00]). When separately hosted as an independent actor, the nature of the references needed to interact with it enforces the necessary security policies. There is no code just for security:

Scala

```
@serializable @SerialVersionUID(1L)
class PurseX(var balance: Int) extends Purse {
    def getBalance(): Promise[Int] = {
        return ref(balance);
    }
    def deposit(purse: Purse): Promise[Int] = {
       val realPurse = purse.asInstanceOf[PurseX]
       val transfer = realPurse.balance
       realPurse.balance = 0
       balance += transfer;
       return ref(transfer);
    }
   def withdraw(amount: Int): Purse = {
      assert(amount <= balance && amount >= 0);
      balance -= amount;
      return new PurseX(amount);
  }
}
@serializable @SerialVersionUID(1L)
object MoneyMaker {
    def make(q: Eventual): Purse = {
        new PurseX(100000)
 }   }
```

Java

```
class PurseX extends Serializable, Purse {
   private static final long serialVersionUID = 1L;
    public static Purse make(Eventual _) {
       return new PurseX(100000);
    }

    public Promise<Integer> getBalance() {
       return ref(balance);
    }
    public Promise<Integer> deposit(Purse purse) {
       PurseX realPurse = (PurseX) purse;
       int transfer = realPurse.balance;
       realPurse.balance = 0;
       balance += transfer;
       return ref(transfer);
    }
    public Purse withdraw(int amount) {
       assert(amount <= balance && amount >=0);
       balance -= amount;
       return new PurseX(amount);
    }
}
```

Conclusion

Different programmers will draw different conclusions about the relative hazards of sequential programming versus relentlessly-reliable-reference distributed programming. Our own experience in the development of the SCoopFS system was as follows: at the beginning of the project, when we were just learning how to use r-r-refs, we avoided spawning separate actors and strictly minimized the number of nodes. Over the 18 months of development, our attitudes gradually transitioned, until at the end of the development effort we were actively seeking, when adding new features, opportunities to modularize components into separate actors. Our design principle transformed over time from, "For quick easy development, can I keep this next task sequential?" to "For quick solid results, can I make this next task distributed?"

References

[Close04] "Waterken Server", Tyler Close, 2004, http://waterken.sourceforge.net/.

[Farmer04] "Habitat Redux", Randy Farmer, Chip Morningstar, http://habitatchronicles.com/Habitat/historical/HabitatRedux.ppt

[Finifter08] "Verifiable Functional Purity in Java, Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. 15th ACM Conference on Computer and Communication Security (CCS 2008), October 27-31, 2008. http://naveen.ksastry.com/papers/pure-ccs08.pdf

[Karp09] "Not 1 Click for Security", Alan Karp, Marc Stiegler, Tyler Close, Symposium on Usable Privacy and Security 2009. http://www.hpl.hp.com/techreports/2009/HPL-2009-53.pdf

[Lee06] "The Problem with Threads", Edward Lee, Technical Report No. UCB/EECS-2006-1, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html

[Miller00] “Capability Based Financial Instruments”, Financial Cryptography 2000, Mark Miller, Chip Morningstar, Bill Frantz, http://www.erights.org/elib/capability/ode/.

[Miller02] “Event Loop Concurrency”, Mark Miller, 2002, http://www.erights.org/elib/concurrency/event-loop.html

[Miller04] “Promise Pipelining”, http://www.erights.org/elib/distrib/pipeline.html

[Miller06] “Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control”, Mark Miller, dissertation, Johns Hopkins University, 2006. http://www.erights.org/talks/thesis/

[Stanley09] “Causeway: a message-oriented distributed debugger”, Terry Stanley, Tyler Close, Mark S. Miller, HP Labs Tech Report, 2009, http://www.hpl.hp.com/techreports/2009/HPL-2009-78.pdf

[Stiegler04] “E in a Walnut: Distributed Computing”, http://wiki.erights.org/wiki/Walnut/Distributed_Computing

[Stiegler07]:"Emily: A High Performance Language for Enabling Secure Cooperation," Marc Stiegler, c5, pp.163-169, Fifth International Conference on Creating, Connecting and Collaborating through Computing (C5 '07), 2007, http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=04144948

[Stiegler07b] “A PictureBook of Secure Cooperation”, Marc Stiegler, http://www.erights.org/talks/efun/SecurityPictureBook.pdf