FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

A Framework for Distributed Systems Based on The Actor Programming Model and Dart language, Which Unifies Applications Across Devices, Clients and Servers, and Supports Features for Hot Deployment and Migration of Actors

Sushil Man Shilpakar

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

A Framework for Distributed Systems Based on The Actor Programming Model and Dart language, Which Unifies Applications Across Devices, Clients and Servers, and Supports Features for Hot Deployment and Migration of Actors

Ein Framework für verteilte Systeme auf der Basis des Actor-Programmiermodells und der Dart-Programmiersprache, die Anwendung in Endgeräten, Clients und Servern erlaubt und die Möglichkeit für Hot Deployment und Migration der Actors bietet

Author:	Sushil Man Shilpakar
Supervisor:	Prof. Hans Arno Jacobsen
Advisor:	Richard Billeci
Submission Date:	TODO: Submission date

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, TODO: Submission date Sushil Man Shilpakar

Acknowledgments

TODO

Abstract

TODO

Contents

1 Introduction

1.1 Background

[Concurrency issues have lately received enormous interest because of two converging trends: first,] multi-core processors make concurrency an essential ingredient of efficient program execution. Second, distributed computing and web services are inherently concurrent. Message-based concurrency is attractive because it might provide a way to address the two challenges at the same time. It can be seen as a higher-level model for threads with the potential to generalize to distributed computation. Many message passing systems used in practice are instantiations of the actor model [28,2]. A popular implementation of this form of concurrency is the Erlang programming language [4]. Erlang supports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes [3,36]. [HO09]

The purpose of this thesis is to build a framework based on the actor programming model and the Dart language. The framework unifies applications across devices, client and server and also supports migration of actors in a distributed system. So, first of all we should briefly overview the actor programming model and how it can be realized efficiently in the Dart language. Although the actor model was introduced in mid 1980s and there had been programming languages like Erlang that implemented it, only now it has started gaining wide popularity in distributed systems. Especially after the introduction of Scala and Akka, the actor model has been gaining good popularity. The Dart programming language provides a homogeneous system that encompasses both client as well as server as the Dart Virtual Machine runs in servers as well as in browsers. This particular nature of the Dart language makes it possible to create a fully distributed application in which isolates (the actor like entities of Dart language) may run everywhere — in servers, in desktop browsers and even in mobile browsers. This thesis is about creating a framework that would take the Dart's actor like entities — Isolates to a distributed system with added capabilities.

1.2 Goals

Create a framework for writing programs in the Dart Language that enforces the actor based programming model such that the final system is inherently distributed in nature. The framework should make improve the 'availability' of an application that is built on top of it. * The framework shall make the application scalable at different levels. * The framework shall support hot deployments and migration of Isolates, which eventually improves the availability of the system.

2 Literature Review

2.1 Actor Programming Model

Actor programming model is a programming paradigm designed for concurrent computation. The concept of an actor was originally introduced by Carl Hewitt[AH85] and he along with Agha[Agh85] have been involved in development of both the actor theory as well as its implementation. An actor is a fundamental unit of computation. It is neither an operating system process nor a thread, but a lightweight process. It has to embody three essential things:

- Processing
- Storage
- Communication

Actors have addresses, each actor can have multiple addresses or multiple addresses can belong to one actor. Actors communicate with each other in a non-blocking way by message passing, thus removing the need of explicit locks. An actor can send message to actors in another system or it can also send message to itself for recursion. An actor can send messages only to addresses that it receives in the message, addresses that it already had before it received the message or addresses of actors that it creates while processing the message. Each actor has a mailbox. A mailbox is a queue of messages that have been sent by other actors or processes and not yet consumed, where mailbox is also an actor. The order in which the messages are delivered is non-deterministic. When an actor receives a message it can: (Essentially, an actor is a computational agent which carries out its actions in response to accepting a communication. The actions it may carry out are:) [AH85]

- Create other actors
- Send messages to itself, other known actors or reply to actor who sent the message
- Designate how it is going to handle next message, i.e. Specify replacement behavior

There are three ways in which an actor A, upon accepting a communication K, can know of a target to which it can send a communication. These are: [Agh85]

- the target was known to the actor a before it accepted the communication
- the target became known when the message was accepted because it was contained in the message or
- the target is the mail address of a new actor created as a result of accepting the message

Actors do not have shared mutable state. All mutable state is private to the actor and all shared state is immutable Actor communicate with each other asynchronous message passing which is immutable. Each actor processes only one message at a time, and unless it is a broadcast message, a message is not processed two times.

An actor alone does nothing much, they must come in a system. An actor system is a group of actors working together in certain hierarchy. In the actor model, concurrency is inherent because of the way it is designed. Also, there is no guarantee that the message sent to an actor will arrive sequentially.

2.1.1 Error Handling in Actor Model

Error handling in actor model is based on "let it crash" philosophy. The isolated, shared nothing trait of actors allows a single actor to fail without affecting any other actors. Furthermore, the actor model itself can be used for fault tolerance, by spawning hierarchy trees of actors for supervision. Once an actor crashes, the supervising actor receives a messages and can react. A supervisor usually handles the error in actors by restarting the actor, stopping the actor, ignoring the error or escalating the error to its own supervisor. `http://berb.github.io/diploma-thesis/original/054_actors.html`

Several programming languages like Act 1, 2 and 3, Acttalk etc were created when actor system was newly introduced by Hewitt and Agha.

2.1.2 Differences From Conventional Programming Languages

In traditional programming languages, the control flow of a concurrent program is divided over a number of threads. Each thread operates concurrently and control can switch from one thread to another non-deterministically. If two threads have access to the same data (objects), they might cause erroneous behavior (so-called race conditions) because of this non-determinacy. Therefore, thread-based programming languages introduce locks (in the form of monitors, semaphores, ...) which enable the construction

of so-called critical sections, which are pieces of program code in which only one thread can run sequentially at a time.

The advantages of the thread-based model are that the model itself is easy to understand, it is efficiently implementable and it can be used to create very fine-grained synchronization (e.g. multiple readers/one writer). The disadvantages are that the resulting program behavior is very hard to understand because of implicit context switches, interleaved acquisition/release of locks which may lead to deadlock, etc. `http://soft.vub.ac.be/amop/at/tutorial/actors#threads_vs_actors`

2.1.3 Actors for Concurrency

When the application servers use the actor model, each incoming request represents a new actor. For parallelizing request operations, the actor spawns new actors and assigns work via messages. This enables parallel I/O-bound operations as well as parallel computations. Often, the flow of a single request represents a more or less complex message flow between multiple actors, using messaging patterns such as scatter/gather, router, enricher or aggregator [Hoh03].

However, implementing request logic using actors differs clearly from sequential request logic implementations. The necessary coordination of multiple actors and the less apparent flow of execution due to asynchronous messaging provides an arguably less comfortable, but more realistic abstraction towards concurrency

Another feature of the actor model is the possibility to scale the actor system as a whole by adding new machines. For instance, Erlang enables virtual machines to spawn a distributed system. In this case, remote actors can hold isolated application state, but accessible via messaging for all other actors of the entire system. `http://berb.github.io/diploma-thesis/original/054_actors.html#impl`

Since actors may carry out their actions in parallel and messages may be reordered, actor based systems are entirely concurrent. Each actors lives totally separated from all other actors and only messages, which are conceptually different from actors, are passed between them. So two actors may be executed by a single thread on a single chip that alternates execution between the two of them or they may be executed parallel within two data centers on both ends of the world. The actors themselves can and should not know, they simply exchange messages. This property makes it easy to scale actor based systems out of a single core to millions of them in multiple data centers. In an ideal case you simply change the configuration of the system and you are done without any need to modify the code. [MartinThurau]

2.2 Erlang

Erlang is one of the first fairly popular programming languages that was based on the actor model.[Vin07] It was developed by Joe Armstrong in 1986 at the Ericsson Computer Science Laboratory but was made open-source only in 1998. It was chiefly used in telephony applications as it was built to solve the problems of availability as well as scalability that existed in such applications. [Arm07]

Erlang was designed for writing concurrent programs that "run forever". It uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. It has mechanisms to allow programs to change code "on the fly" so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems. [Arm07]

Error handling in Erlang is different from conventional programming languages. In Erlang, it is based on a "let is crash" philosophy [Arm07].

2.3 Scala Actors

Scala is a statically typed programming language which integrates functional and object-oriented programming quite well.[OR14]

In Scala, templates for actors with user-defined behavior are normal class definitions which extend the predefined Actor class. The Scala Actors library provides both asynchronous and synchronous message sends (the latter are implemented by exchanging several asynchronous messages). Moreover, actors may communicate using futures where requests are handled asynchronously, but return a representation (the future) that allows to await the reply. [Hal]

The actor implementation of Scala is not part of the language core, but part of its standard library. This means that actor library itself is implemented in Scala. One initial challenge of Scala actors has been introduced by the constraints of multithreading in the Java Virtual Machine (JVM). The number of possible threads is limited, there is no cooperative scheduling available and threads are conceptually less lightweight than actors are supposed to be. As a result, Scala provides a single concept of an actor, but two different mechanisms for message handling. [HO09]

Thread-based Actors

When actors call thread-blocking operations, such as receive (or even wait), the worker thread that is executing the current actor (self) is blocked. This means basically that the

actor is represented as a blocked thread. Depending on the number of actors you want to use, you might want to avoid this, since most JVMs cannot handle more than a few thousand threads on standard hardware.

Event-driven Actors

The react primitive allows and event-driven execution strategy, which does not directly couple actors to threads. Instead, a thread pool can be used for a number of actors. This approach uses a continuation closure to encapsulate the actor and its state. However, this mechanism has several limitations and obscures the control flow. [HO09] Conceptually, this implementation is very similar to an event loop backed by a thread pool. Actors represent event handlers and messages resemble events. `http://berb.github.io/diploma-thesis/original/054_actors.html#impl`

2.4 Akka Toolkit

[MartinThurau] Akka is a toolkit for building concurrent, distributed application on the JVM using the actor model. It provides the developer with a well defined API to develop large concurrent systems and allow for easy scaling out of a single machine.

2.4.1 Actor Model in Akka

[MartinThurau] The actor model is implemented in Akka with respect to the definition above. However, there are certain parts of this implementation that have subtle differences in them. Before we take a look at them, lets first start with a simple code example on how an actor actually looks if you implement it with Akka. After all, that concept is abstract enough so an example will certainly help to build a mental model of it. Listing 1 shows an example of "Hello World" using a single actor. The example is written in Java.

All basic classes of Akka actors are available with an single import in line 1. The actual Actor is implemented in the lines 3 to 8. We only need to extend the class Actor and override the receive method. This method is called each time a message is received by the actor. The method does pattern matching on the received message to decide what to do. In this case, whenever a message is received that contains a single string a message is printed to the console.[MartinThurau]

Akka Sample Java Code: [Inca]

```
public class Greeting implements Serializable {
  public final String who;
  public Greeting(String who) { this.who = who; }
```

```
}

public class GreetingActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public void onReceive(Object message) throws Exception {
    if (message instanceof Greeting)
      log.info("Hello " + ((Greeting) message).who);
    }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(Props.create(GreetingActor.class), "greeter");
greeter.tell(new Greeting("Charlie Parker"), ActorRef.noSender());
```

2.4.2 Actor System

[http://www.toptal.com/scala/concurrency-and-fault-tolerance-made-easy-an-intro-to-akka] Taking a complex problem and recursively splitting it into smaller sub-problems is a sound problem solving technique in general. In an actor-based design, use of this technique facilitates the logical organization of actors into a hierarchical structure known as an Actor System. The actor system provides the infrastructure through which actors interact with one another. Actors are objects which encapsulate state and behavior, they communicate exclusively by exchanging messages which are placed into the recipient's mailbox. In a sense, actors are the most stringent form of object-oriented programming, but it serves better to view them as persons: while modeling a solution with actors, envision a group of people and assign sub-tasks to them, arrange their functions into an organizational structure and think about how to escalate failure (all with the benefit of not actually dealing with people, which means that we need not concern ourselves with their emotional state or moral issues). The result can then serve as a mental scaffolding for building the software implementation. [Incb] "An ActorSystem is a heavyweight structure that will allocate 1...N Threads, so create one per logical application." The quintessential feature of actor systems is that tasks are split up and delegated until they become small enough to be handled in one piece. In doing so, not only is the task itself clearly structured, but the resulting actors can be reasoned about in terms of which messages they should process, how they should react normally and how failure should be handled. If one actor does not have the means for dealing with a certain situation, it sends a corresponding failure message to its supervisor, asking for help. The recursive structure then allows to handle failure at the right level. [Incb] An actor system manages the resources it is configured to use in order to run the actors which it contains. There may be millions of actors within one such system, after all the mantra

is to view them as abundant and they weigh in at an overhead of only roughly 300 bytes per instance. Naturally, the exact order in which messages are processed in large systems is not controllable by the application author, but this is also not intended. Take a step back and relax while Akka does the heavy lifting under the hood. [Incb]

<Insert Actor System Diagram Here>

2.4.3 Message Passing

[MartinThurau] An Actor in Akka is modeled as single object of functionality. Each actor has an event driven message inbox (called mailbox) that holds all incoming message until they are processed. Within the actor there is an function that takes out one message and acts accordingly. Each actor is identified by a so called ActorRef. This references are created whenever one actor creates another actor and are additionally added automatically to each message that is sent between two actors identifying the sender of the message.

Messages in Akka can be from any type. You should however be aware that these object will not be copied when to actors run on the same physical system so it is possible to create a situation where two actors share some kind of data.

Akka chooses to implement some message ordering guarantees for certain kinds of messages. Specifically Akka provides a guaranteed ordering for any two pairs of actors. So if actor A sends two messages M1 and M2 to actor B, M1 will be received before M2. However, this only is the case if the receivers uses a simple FIFO mailbox and not one of the other mailboxes that allow prioritizing certain messages so this featured should be used carefully.

These are the rules for message sends (i.e. the tell or ! method, which also underlies the ask pattern): [Incb]

- at-most-once delivery, i.e. no guaranteed delivery
- message ordering per sender–receiver pair

When it comes to describing the semantics of a delivery mechanism, there are three basic categories:

- at-most-once delivery means that for each message handed to the mechanism, that message is delivered zero or one times; in more casual terms it means that messages may be lost.
- at-least-once delivery means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.

- exactly-once delivery means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

The first one is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism. The second one requires retries to counter transport losses, which means keeping state at the sending end and having an acknowledgement mechanism at the receiving end. The third is most expensive—and has consequently worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries.

Why no guaranteed Delivery?

At the core of the problem lies the question what exactly this guarantee shall mean: [Incb] 1. The message is sent out on the network? 2. The message is received by the other host? 3. The message is put into the target actor's mailbox? 4. The message is starting to be processed by the target actor? 5. The message is processed successfully by the target actor?

2.4.4 Shared Data

[MartinThurau] Since Akka runs on top of the JVM and is implemented as a simply library it is not possible for Akka to strictly enforce a true data separation between actors (other then inspecting every single message which would certainly hurt performance). If an actor creates a mutable data structure and sends a reference to this structure to another actor (using a message that contains the reference to the structure) and both actors run on the same VM these two actors will share the same underlying data structure, thus breaking the encapsulation of the actors. This will most certainly create hard to reproduce bugs (since all actors run concurrent and the process could likely become indeterministic) so this is strongly discouraged. Additionally, this problem goes away if two communicating actors don't run within the same JVM, since the data in the messages is then actually serialized and send over to the other JVM.

2.4.5 Actor Supervision and Monitoring

As described in Actor Systems supervision describes a dependency relationship between actors: the supervisor delegates tasks to subordinates and therefore must respond to their failures. When a subordinate detects a failure (i.e. throws an exception), it suspends itself and all its subordinates and sends a message to its supervisor, signaling

failure. Depending on the nature of the work to be supervised and the nature of the failure, the supervisor has a choice of the following four options:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Stop the subordinate permanently
4. Escalate the failure, thereby failing itself

It is important to always view an actor as part of a supervision hierarchy, which explains the existence of the fourth choice (as a supervisor also is subordinate to another supervisor higher up) and has implications on the first three: resuming an actor resumes all its subordinates, restarting an actor entails restarting all its subordinates (but see below for more details), similarly terminating an actor will also terminate all its subordinates. It should be noted that the default behavior of the preRestart hook of the Actor class is to terminate all its children before restarting, but this hook can be overridden; the recursive restart applies to all children left after this hook has been executed.

[MartingThurau] In Akka each running actor has a supervisor which is the parent actor that created the current one. So each given actor is part of a hierarchy of supervisor: the one parent that created the actor and all child actors, that were created by itself were it is the supervisor. But what does supervision mean? A supervisor created the children so it either delegated some kind of work to them or is a least somehow interested in them. This means it is responsible for watching its subordinates and handling problem that they themselves can not handle. If an actor detects a failure (i.e. if an exception is thrown) it suspends itself and all its children and signals a failure to its supervisor. The supervisor can now choose how to respond this particular failure by

- simply resuming the subordinate. In this case the subordinate keeps all its internals state and will continue were it left of.
- restarting the subordinate. The subordinate will in this case clear all it's internal state.
- terminate the subordinate permanently.
- escalate the failure to its own supervisor.

It has to be noted that in Akka there is no way to put a message back into the inbox. So if a actor signals a failure the current message may be lost. Also it is important

to note that restarting an actor will reset all it's internal state but not it's mailbox. So the "new" actor will continue to process all messages in the mailbox after the restart is complete. Additionally, restarting an actor will create a new actor behind the same ActorRef so other actors that had a reference to the restarted actor can and will not know, that the actor has been restarted. If this wouldn't be the case you would have to notify all other actors whenever a single actor crashed so that they could update eventually stored references. However, there might by certain situations where some actor wants to be notified an other actor is permanently stopped. For this cases Akka provides a feature called DeathWatch were the watching actor is notified by a special message type, that is delivered to its inbox. If we recap that each actor must have a parent actor the supervises it, we notice a chicken-and-egg problem: Who starts the first actor? This is done by a special actor (that is in fact not a real actor) called the "bubble-walker" (because he lives "outside the bubble") which in turn will receive all failures that were escalated to the top level. If the bubble walker ever gets such a failure he will terminate all actors which effectively stops the system.

2.4.6 Routers

[MartinThurau] Routers are a special type of actor that route incoming messages to other actors. They are indeed a "load balancer". One can either implement his own router or use some of the implementations that Akka provides. There are some simple ones in example, that can be used to load balance messages to a number of actors (on different hosts) this way distributing the workload. In Addition to these simple "distributing" routers there are others that broadcast messages to all actors or broadcast them and wait for the first result to complete.

Routers can also be used to automatically adjust the number of actors depending on the number of messages by spawning or stopping actors.

2.4.7 Event Bus

Additionally to the whole message passing Akka provides an event bus that allows parts of the code to publish events to the event bus and other parts may independently subscribe to certain events and get notified if these events occur. This features can be used to decouple actors from each other but have them be able to pass information around using the event bus.

2.4.8 Remote Actors in Akka

Akka Remoting is a communication module for connecting actor systems in a peer-to-peer fashion, and it is the foundation for Akka Clustering. The design of remoting is

driven by two (related) design decisions: 1. Communication between involved systems is symmetric: if a system A can connect to a system B then system B must also be able to connect to system A independently. 2. The role of the communicating systems are symmetric in regards to connection patterns: there is no system that only accepts connections, and there is no system that only initiates connections. [Incb]

Actors are location transparent and distributable by design. This means that you can write your application without hardcoding how it will be deployed and distributed, and then later just configure your actor system against a certain topology with all of the application's semantics, including actor supervision, retained. Sample Akka Config and Code for Remote Actors: [Inca]

```
// ------------------------------
// config on all machines
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote = akka.tcp://MySystem@machine1:2552
      }
    }
  }
}

// ------------------------------
// define the greeting actor and the greeting message
public class Greeting implements Serializable {
  public final String who;
  public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public void onReceive(Object message) throws Exception {
    if (message instanceof Greeting)
      log.info("Hello " + ((Greeting) message).who);
  }
}

// ------------------------------
// on machine 1: empty system, target for deployment from machine 2
ActorSystem system = ActorSystem.create("MySystem");

// ------------------------------
// on machine 2: Remote Deployment - deploying on machine1
```

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(Props.create(GreetingActor.class), "greeter");

// ------------------------------
// on machine 3: Remote Lookup (logical home of "greeter" is machine2, remote
    deployment is transparent)
ActorSystem system = ActorSystem.create("MySystem");
ActorSelection greeter =
    system.actorSelection("akka.tcp://MySystem@machine2:2552/user/greeter");
greeter.tell(new Greeting("Sonny Rollins"), ActorRef.noSender());
```

2.4.9 Akka in Distributed Systems

2.4.10 Clustering in Akka

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. It does this using gossip protocols and an automatic failure detector.

node A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a host- name:port:uid tuple. cluster A set of nodes joined together through the membership service. leader A single node in the cluster that acts as the leader. Managing cluster convergence, partitions [*], fail-over [*], rebalancing [*] etc.

2.5 The Dart Language

2.5.1 Overview

Dart is an open-source, class-based, single-inheritance, pure object-oriented programming language developed by Google. Dart is optionally typed and supports reified generics. The runtime type of every object is represented as an instance of class Type which can be obtained by calling the getter runtimeType declared in class Object, the root of the Dart class hierarchy. Dart programs may be statically checked. The static checker will report some violations of the type rules, but such violations do not abort compilation or preclude execution. Dart programs may be executed in one of two modes: production mode or checked mode. In production mode, static type annotations have absolutely no effect on execution with the exception of reflection and structural type tests.

1. Reified type information reflects the types of objects at runtime and may always be queried by dynamic typechecking constructs (the analogs of in- stanceOf, casts, typecase etc. in other languages). Reified type information includes class

declarations, the runtime type (aka class) of an object, and type arguments to constructors.

2. Static type annotations determine the types of variables and function declarations (including methods and constructors).

3. Production mode respects optional typing. Static type annotations do not affect runtime behavior.

[ECM14] Dart programs are organized in a modular fashion into units called libraries. Libraries are units of encapsulation and may be mutually recursive.
-> Garbage collection !

Developers of dart believe that JavaScript has been pushed to its limit and the web apps developed in JavaScript are far too slow even though JavaScript engines are quite fast. They claim Dart offers a better solution to build larger and more complex web apps.[7] Some of its important features are:

- Familiar syntax, thus easy to learn
- Compiles (Translates) to JavaScript
- Runs in client as well as on server
- Dart supports types, but it is optional
- Can scale from small script to large and complex applications • Support safe concurrency with isolates.

Dart provides a homogeneous system that encompass both client as well as server as the Dart VM (Virtual Machine) can be embedded in browsers. A version of Chromium – 'Dartium' already has Dart VM built into it.

2.5.2 Advantages of Dart

* Translates to JavaScript so that the code can be run in the web-browsers that do not have Dart VM yet * Currently in 20th position in most popular programming languages * Optionally typed language

Important Concepts

- Everything you can place in a variable is an object, and every object is an instance of a class. Even numbers, functions, and null are objects. All objects inherit from the Object class.

- Specifying static types (such as num in the preceding example) clarifies your intent and enables static checking by tools, but it's optional. (You might notice when you're debugging your code that variables with no specified type get a special type: dynamic.)
- Dart parses all your code before running it. You can provide tips to Dart—for example, by using types or compile-time constants—to catch errors or help your code run faster.
- Dart supports top-level functions (such as main()), as well as functions tied to a class or object (static and instance methods, respectively). You can also create functions within functions (nested or local functions).
- Similarly, Dart supports top-level variables, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as fields or properties.
- Unlike Java, Dart doesn't have the keywords public, protected, and private. If an identifier starts with an underscore (_), it's private to its library.

2.5.3 Dart and JavaScript

Dart is a web programming language developed by Google. It is a fairly new language and competes with JavaScript. The code written in Dart can also be translated, using a tool – 'dart2js', to JavaScript code so that it can run in any modern browser. Furthermore, Dart allows developers to code in a uniform way for both server as well as client since Dart Virtual Machine (VM) can be embedded in browsers. A variant of Chromium — Dartium browser has an embedded Dart VM.

2.5.4 Asynchronous Programming in Dart

Asynchronous programming often uses callback functions, but Dart provides alternatives: Future and Stream objects. A Future is like a promise for a result to be provided sometime in the future. A Stream is a way to get a sequence of values, such as events. Future, Stream, and more are in the dart:async library. The dart:async library works in both web apps and command-line apps

2.5.5 Asynchronous Message Sending

Messages are the sole means of communication among isolates. Messages are sent by invoking specific methods in the Dart libraries; there is no specific syntax for sending a

message. In other words, the methods supporting sending messages embody primitives of Dart that are not accessible to ordinary code, much like the methods that spawn isolates.

2.5.6 Isolates

[ECM14] Dart code is always single threaded. There is no shared-state concurrency in Dart. Concurrency is supported via actor-like entities called isolates. An isolate is a unit of concurrency. It has its own memory and its own thread of control. Isolates communicate by message passing. No state is ever shared between isolates. Isolates are created by spawning. Spawning an isolate is accomplished via what is syntactically an ordinary library call, invoking one of the functions 'spawnUri()' or 'spawnFunction()' defined in the 'dart:isolate' library. However, such calls have the semantic effect of creating a new isolate with its own memory and thread of control. The newly spawned isolate shares the same code as the spawner isolate. [Goo] An isolate's memory is finite, as is the space available to its thread's call stack. It is possible for a running isolate to exhaust its memory or stack, resulting in a run-time error that cannot be effectively caught, which will force the isolate to be suspended. These peculiar characteristics of isolates make them closer to ideal actors of Hewitt's actor model, however the hidden message passing system takes them farther from being ideal actors.In Dart, when an isolate is spawned, usually the initial message contains a sending port, so that spawner and "spawnee" can communicate with each other. The "spawnee" can later use the same port to reply to the spawner. An isolate can spawn another isolate which can further spawn other isolates and have control over them. Thus, the spawner can supervise the "spawnee". The spawner can pause the "spawnee" or terminate it.

Modern web browsers, even on mobile platforms, run on multi-core CPUs. To take advantage of all those cores, developers traditionally use shared-memory threads running concurrently. However, shared-state concurrency is error prone and can lead to complicated code. Instead of threads, all Dart code runs inside of isolates. Each isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate. [Kat12]

Spawning an Isolate

* Using top level function -> Isolate.spawn * Using URI and spawn from file -> Isolate.spawnUri() -> Really cool feature

SendPort of Isolate

ReceivePort of Isolate

Communication Between Two Isolates

TODO: A Sample Code with Isolates

Limitations of Isolates

Although the communication between Isolates takes place exclusively by asynchronous message passing, which perfectly suits for distributed systems. There is no possibility of communication with isolate spawned in another dart virtual machine. A message exchange between two isolates is possible only if the isolates are spawned locally in the same dart virtual machine. Thus, a hindrance in making them distributed.

Difference from Actor

Although, not having shared state, and making message-passing as the only way to communicate between isolates, the Dart isolates differ from actors in many ways. The most significant difference is the principle behind spawning of actor and spawning of isolate. An actor is supposed to be a very lightweight and cheap to spawn but spawning isolates in Dart takes significant amount of time and resource. The implementation of actor which are found in other languages like Erlang and Akka framework can be considered much closer to Hewitt's actor model. The number of actors that can be spawned per GigaByte of heap memory in Akka reaches up to 2.7 millions[citation needed!] where as in dart it only reaches up to few hundred [100?]. Based on these observations, it would be appropriate to say that the current[1] implementation of an Isolate in Dart is — similar to a thread with properties like an actor.

2.6 RabbitMQ - A Message Broker System

RabbitMQ is a messaging broker that serves as an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.`http://www.pivotal.io/products/pivotal-rabbitmq` A message broker is an intermediary program that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. `http://whatis.techtarget.com/definition/message-broker` Messaging enables software applications to connect and scale. Applications can connect to each

[1] Dart version 1.7.2

other, as components of a larger application, or to user devices and data. Messaging is asynchronous, decoupling applications by separating sending and receiving data. In RabbitMQ, messages are routed through exchanges before arriving at queues. RabbitMQ features several built-in exchange types for typical routing logic.[Sofc]

- Several RabbitMQ servers on a local network can be clustered together, forming a single logical broker.
- Queues can be mirrored across several machines in a cluster, ensuring that even in the event of hardware failure the messages are safe.
- RabbitMQ can transmit messages over HTTP in different ways, one of them is by using WebSockets.

RabbitMQ support several protocols for enqueuing and dequeuing messages:

- AMQP (Several versions) Stands for Advanced Message Queuing Protocol. Designed as an open replacement for existing proprietary messaging middleware. Two of the most important reasons to use AMQP are reliability and interoperability. As the name implies, it provides a wide range of features related to messaging, including reliable queuing, topic-based publish-and-subscribe messaging, flexible routing, transactions, and security. AMQP exchanges route messages directly—in fanout form, by topic, and also based on headers.[Pip]

  There's a lot of fine-grained control possible with such a rich feature set. You can restrict access to queues, manage their depth, and more. Features like message properties, annotations and headers make it a good fit for a wide range of enterprise applications. This protocol was designed for reliability at the many large companies who depend on messaging to integrate applications and move data around their organization. In the case of RabbitMQ, there are many different language implementations and great samples available, making it a good choice for building large scale, reliable, resilient, or clustered messaging infrastructures.

- STOMP 2.7 STOMP is a text-based messaging protocol emphasizing (protocol) simplicity. It defines little in the way of messaging semantics, but is easy to implement and very easy to implement partially (it's the only protocol that can be used by hand over telnet).

  RabbitMQ supports STOMP (all current versions) via a plugin.

- MQTT (Message Queue Telemetry Transport) was originally developed out of IBM's pervasive computing team and their work with partners in the industrial sector. Over the past couple of years the protocol has been moved into the open

source community, seen significant growth in popularity as mobile applications have taken off, and it is in the process of moving into the hands of a standards body.

The design principles and aims of MQTT are much more simple and focused than those of AMQP—it provides publish-and-subscribe messaging (no queues, in spite of the name) and was specifically designed for resource-constrained devices and low bandwidth, high latency networks such as dial up lines and satellite links, for example. Basically, it can be used effectively in embedded systems. [Pip]

MQTT is a binary protocol emphasizing lightweight publish / subscribe messaging, targetted towards clients in constrained devices. It has well defined messaging semantics for publish / subscribe, but not for other messaging idioms.

RabbitMQ supports MQTT 3.1 via a plugin.

- HTTP HTTP is of course not a messaging protocol. However, RabbitMQ can transmit messages over HTTP in three ways:

  Management Plugin The management plugin supports a simple HTTP API to send and receive messages. This is primarily intended for diagnostic purposes but can be used for low volume messaging without reliable delivery.

  Web-STOMP Plugin The Web-STOMP plugin supports STOMP messaging to the browser, using WebSockets or one of the fallback mechanisms supported by SockJS.

  JSON-RPC channel Plugin The JSON-RPC channel plugin supports AMQP 0-9-1 messaging over JSON-RPC to the browser. Note that since JSON RPC is a synchronous protocol, some parts of AMQP that depend on asynchronous delivery to the client are emulated by polling.

2.6.1 Message Queues in RabbitMQ

What is a queue? How are they enqueued? How are they dequeued?

If the queue receives messages at a faster rate than it can pump out to consumers then things get slower. As the queue grows, it will require more memory. Additionally, if a queue receives a spike of publications, then the queue must spend time dealing with those publications, which takes CPU time away from sending existing messages out to consumers: a queue of a million messages will be able to be drained out to ready consumers at a much higher rate if there are no publications arriving at the queue to distract it. Not exactly rocket science, but worth remembering that publications arriving at a queue can reduce the rate at which the queue drives its consumers.[Sofb]

2.6.2 RabbitMQ and prefetch-count

The default QoS prefetch setting gives clients an unlimited buffer, and that can result in poor behavior and performance. But what should you set the QoS prefetch buffer size to? The goal is to keep the consumers saturated with work, but to minimize the client's buffer size so that more messages stay in RabbitMQ's queue and are thus available for new consumers or to just be sent out to consumers as they become free.

AMQP defaults to sending all the messages it can to any consumer that looks ready to accept them. The maximum number of these unacknowledged messages per channel can be limited by setting the prefetch count. However, small prefetch counts can hurt performance. Here is the result of a benchmark [2] which shows how the throughput of a queue varies when prefetch count is changed when there are different number of consumers.

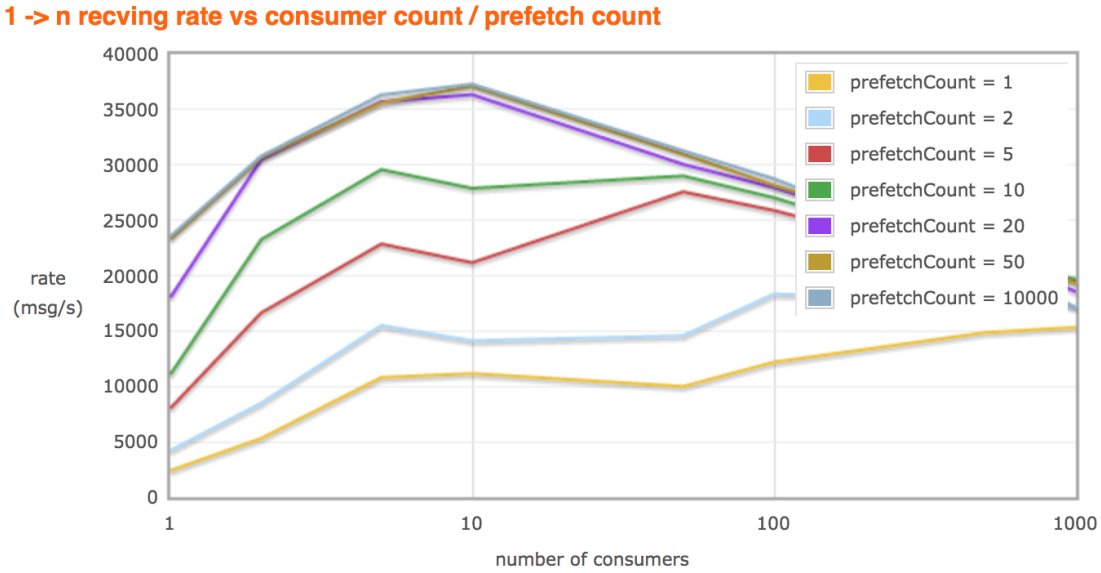

Figure 2.1: Chart showing performance variation when prefetch count is changed in the consumer[3]

[2] posted by Simon MacMullen on April 25th, 2012 at 2:47 pm

[3] P. Software. *RabbitMQ Performance Measurements, part 2*. `http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/`. Last Accessed: 2014-11-20

2.7 STOMP

STOMP is the Simple (or Streaming) Text Orientated Messaging Protocol. STOMP provides an interoperable wire format so that STOMP clients can communicate with any STOMP message broker to provide easy and widespread messaging interoperability among many languages, platforms and brokers. [STO] STOMP is a simple interoperable protocol designed for asynchronous message passing between clients via mediating servers. It defines a text based wire-format for messages passed between these clients and servers. It is an alternative to other open messaging protocols such as AMQP and implementation specific wire protocols used in JMS brokers such as OpenWire. It distinguishes itself by covering a small subset of commonly used messaging operations rather than providing a comprehensive messaging API. The main philosophies driving the design of STOMP are simplicity and interoperability. STOMP is designed to be a lightweight protocol that is easy to implement both on the client and server side in a wide range of languages. This implies, in particular, that there are not many constraints on the architecture of servers and many features such as destination naming and reliability semantics are implementation specific.

2.7.1 Protocol Overview

STOMP is a frame based protocol, with frames modeled on HTTP. A frame consists of a command, a set of optional headers and an optional body. STOMP is text based but also allows for the transmission of binary messages. The default encoding for STOMP is UTF-8, but it supports the specification of alternative encodings for message bodies. A STOMP server is modeled as a set of destinations to which messages can be sent. The STOMP protocol treats destinations as opaque string and their syntax is server implementation specific. Additionally STOMP does not define what the delivery semantics of destinations should be. The delivery, or "message exchange", semantics of destinations can vary from server to server and even from destination to destination. This allows servers to be creative with the semantics that they can support with STOMP. STOMP does not, however, deal in queues and topics—it uses a SEND semantic with a "destination" string. The broker must map onto something that it understands internally such as a topic, queue, or exchange. Consumers then SUBSCRIBE to those destinations. Since those destinations are not mandated in the specification, different brokers may support different flavors of destination. So, it's not always straightforward to port code between brokers. [Pip]

A STOMP client is a user-agent which can act in two (possibly simultaneous) modes:

- as a producer, sending messages to a destination on the server via a SEND frame

- as a consumer, sending a SUBSCRIBE frame for a given destination and receiving messages from the server as MESSAGE frames.

2.7.2 STOMP Library for Dart

Given that dart is fairly new language, it does not yet have AMPQ client that can be used with RabbitMQ. As mentioned above in section 2.6.2 RabbitMQ also supports STOMP Protocol. An open source STOMP client in dart is available which was created by 'Potix corporation'. It can perform most of the basic with a message broker system like connecting, creating queue, subscribing, enqueuing as well as dequeuing. Although it has those basic functionalities, it still has some limitations and incompleteness like lack of support of 'Heartbeat', only support STOMP version 1.2 or above.

2.8 WebSockets

The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the origin-based security model commonly used by web browsers. The protocol consists of an opening handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g., using XMLHttpRequest or <iframe>s and long polling). [FM11] The WebSocket Protocol provides a single TCP connection for traffic in both directions. Combined with the WebSocket API [WSAPI], it provides an alternative to HTTP polling for two-way communication from a web page to a remote server. It can be used for variety of web applications: games, stock tickers, multiuser applications with simultaneous editing, user interfaces exposing server-side services in real time, etc.

The WebSocket Protocol is designed to supersede existing bidirectional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure (proxies, filtering, authentication). Such technologies were implemented as trade-offs between efficiency and reliability because HTTP was not initially meant to be used for bidirectional communication.

2.8.1 Security

The WebSocket Protocol uses the origin model used by web browsers to restrict which web pages can contact a WebSocket server when the WebSocket Protocol is used from a web page. Naturally, when the WebSocket Protocol is used by a dedicated client

directly (i.e., not from a web page through a web browser), the origin model is not useful, as the client can provide any arbitrary origin string. It is similarly intended to fail to establish a connection when data from other protocols, especially HTTP, is sent to a WebSocket server, for example, as might happen if an HTML "form" were submitted to a WebSocket server. This is primarily achieved by requiring that the server prove that it read the handshake, which it can only do if the handshake contains the appropriate parts, which can only be sent by a WebSocket client. In particular, at the time of writing of this specification, fields starting with |Sec-| cannot be set by an attacker from a web browser using only HTML and JavaScript APIs such as XMLHttpRequest [XMLHttpRequest].

2.8.2 Establishing a Connection

When a connection is to be made to a port that is shared by an HTTP server (a situation that is quite likely to occur with traffic to ports 80 and 443), the connection will appear to the HTTP server to be a regular GET request with an Upgrade offer. In relatively simple setups with just one IP address and a single server for all traffic to a single hostname, this might allow a practical way for systems based on the WebSocket Protocol to be deployed. In more elaborate setups (e.g., with load balancers and multiple servers), a dedicated set of hosts for WebSocket connections separate from the HTTP servers is probably easier to manage. At the time of writing of this specification, it should be noted that connections on ports 80 and 443 have significantly different success rates, with connections on port 443 being significantly more likely to succeed, though this may change with time.

2.8.3 WebSocket URIs

This specification defines two URI schemes, using the ABNF syntax defined in RFC 5234 [RFC5234], and terminology and ABNF productions defined by the URI specification RFC 3986 [RFC3986].
ws-URI = "ws:" "//" host [":" port] path ["?" query]
wss-URI = "wss:" "//" host [":" port] path ["?" query]

host = <host, defined in [RFC3986], Section 3.2.2>
port = <port, defined in [RFC3986], Section 3.2.3>
path = <path-abempty, defined in [RFC3986], Section 3.3>
query = <query, defined in [RFC3986], Section 3.4>

The port component is OPTIONAL; the default for "ws" is port 80, while the default for "wss" is port 443.

The URI is called "secure" (and it is said that "the secure flag is set") if the scheme component matches "wss" case-insensitively.

The "resource-name" (also known as /resource name/ in Section 4.1) can be constructed by concatenating the following

- "/" if the path component is empty
- the path component
- "?" if the query component is non-empty
- the query component

2.9 RESTful WebServices

3 System Design

3.1 Core Design Decisions

The Framework is designed to be distributed in nature with the concepts of Actors. It should follow the standard actor programming concept and have the inherently distributed nature. The framework itself should be built using the concept of message passing to alleviate any possibility of concurrency issues, thus thread-safe.

- The framework does not guarantee the delivery of a message.
- All the messages send by the framework is based on 'fire and forget'concept.
- A message is delivered at most once.
- A message is always routed through Message Queuing System, even though the target isolate can belong to same isolate system in the same logical or physical node.
- A message is only fetched after an isolate sends a PULL request. Nonetheless, the delivery of message to an isolate is based on the implementation of the load balancer – i.e. router.

3.2 Architectural Overview

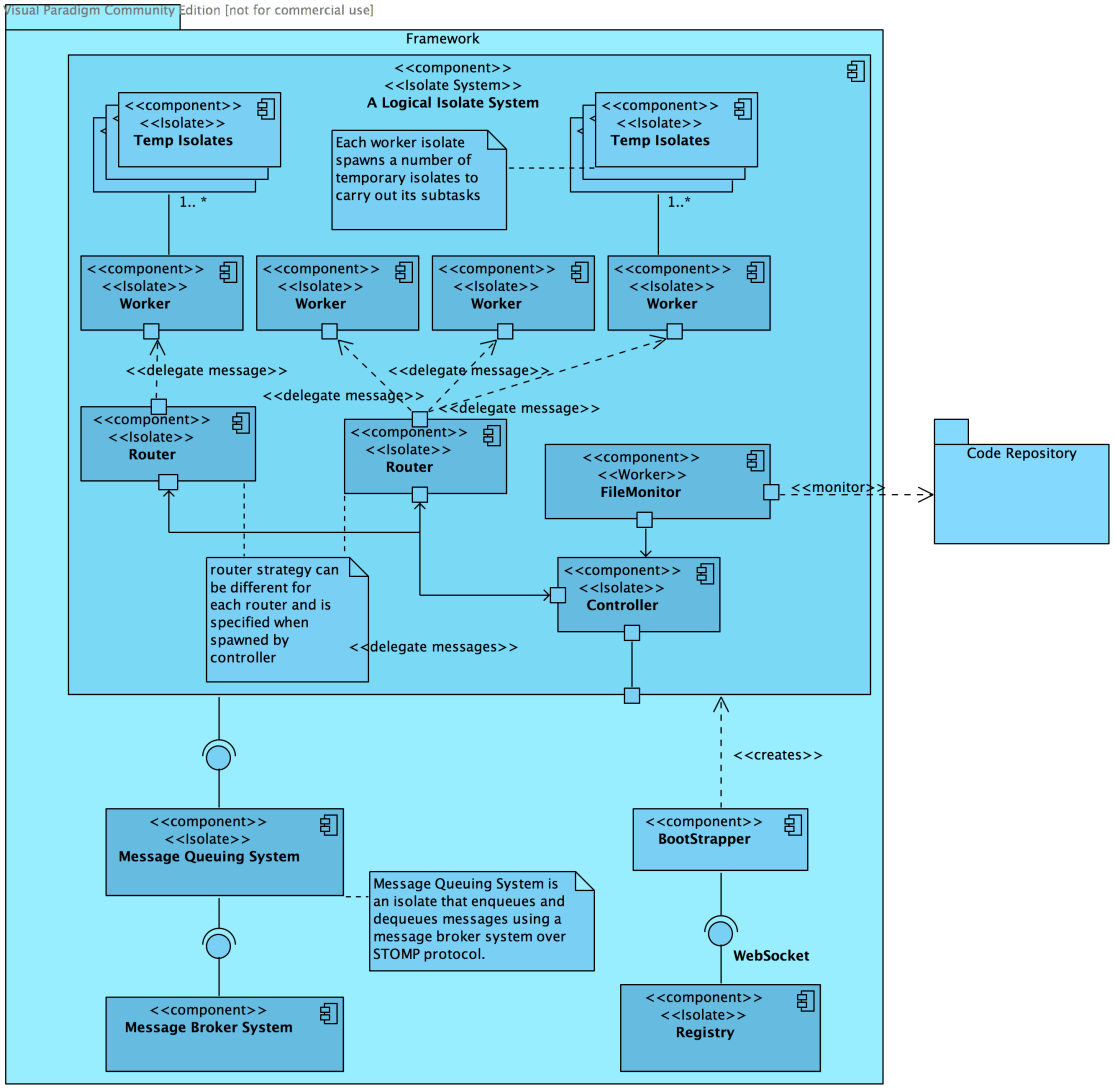

Figure 3.1: Architecture of the framework

3.3 The Framework

The framework comprises of an Isolate System, a Registry, a Message Queuing System and a Message Broker System.

3.3.1 IsolateSystem

An Isolate System is analogous to an actor system. Just as actor system is comprised of a group of actors working together, a group of related isolates form an isolate system. A 'Bootstrapper' in physical node can bootstrap several Isolate Systems. Nevertheless, a logical Isolate System is not limited to a single physical node. The isolates spawned by an Isolate System can be distributed across several remote systems. Each Isolate system has its unique id, which is a UUID. It is generated when the isolate system is bootstrapped. For bootstrapping, isolate system needs the WebSocket path to Message Queuing System, and a 'name' for itself. The name should not be confused with the uniqueId as IsolateSystem with same name can exist in other nodes but the uniqueId is unique for a particular instance of Isolate System. When an Isolate System is bootstrapped, it does several things. It generates a new Id which is unique for itself, then opens up a 'ReceivePort' and listens for any messages in that port so that it can receive to incoming messages from 'Controller'. Then it tries to connect to Message Queuing System. If the Message Queuing System could not be connected, it simply keeps on retrying at certain interval. Furthermore, if after establishing connection with the MQS, the connection is lost in between because of a network issue or a hardware failure, the IsolateSystem automatically keeps on trying to re-establish the connection at regular interval. The connection to MQS via websocket takes place asynchronously thus whether the connection is established or not, it simply goes on and spawns a controller.

Adding an Isolate to the Isolate System

When an Isolate System is first initialized, it is an empty system without any isolates running in it. Isolates can be added with appropriate load balancer once the isolate system has been initialized. The 'addIsolate' function can be used to startup isolates into the system. It requires following arguments:

name – A name for pool of isolates. A deployed isolate has its own name but overall the name is concatenated with the name of isoalte system to denote the hierarchy. For instance, an isolate with name 'account' becomes 'bank/account' where 'bank' is the name of the isolate system.

sourceUri – The location of the source code from which the isolate shall be spawned. The path can either be absolute path to the local file system or the full http or https URI.

workersPaths – List of destinations where each of the isolates should be spawned. To spawn locally 'localhost' should be used, whereas to spawn in remote node,

WebSocket path like: "ws://192.168.2.9:42042/activator" should be used. The number copy of isolates that should be spawned is determined by the length of this list. If multiple copies of isolates should be spawned in a machine, the location can be repeated. For instance ["localhost", "localhost"]; this list results in spawning of two identical isolates in local machine which is load balanced by the type of router chosen.

routerType – The type of load balancing technique that one would like to use to effectively distribute incoming messages. The framework, by default, provides three types of routers: Round-Robin, Random and Broadcast. For instance, to use the 'Random' router provided by the framework 'Router.RANDOM' can be used as the value for routerType. If the user wants to use his custom Router instead of using the options provided by the framework, absolute path to the location of source code, which can also be URI, of the Custom router implementation can be provided as

hotDeployment – This argument is optional and is by default set to true. Setting it to true enables continuous monitoring of the source code for any change. If any change in source code is identified, the instances of isolates, spawned by this 'addIsolate' function, in current isolate system will be restarted, without the need of restarting.

args – Custom additional arguments to be passed into each instance of spawned isolate. This argument is also optional and can be safely ignored.

Message Handling in IsolateSystem

Typically, the message in an IsolateSystem can arrive from three sources: Message Queuing System via WebSocket, Controller via ReceivePort or Bootstrapper via direct function call. As Dart is a single threaded programming language, only one message is handled at a time. The messages arriving from MQS is deserialized from JSON string to Map data type before further processing. As the message received from the MQS contains the source queue, the source queue is parsed and transformed to name of corresponding isolate of the queue (as of current version, the queue name corresponds to the name of the isolate) and then forwarded to the Controller that this instance of IsolateSystem has spawned. The messages arriving from Controller as either messages that should be enqueued or dequeue requests from the isolates that have completed certain task and are free to accept another message. For the messages that are supposed to be enqueued, the target isolate's name is converted to the name of the queue and then sent to the MQS via WebSocket. For the dequeue requests, the

sender of the message is identified, which is then converted to corresponding queue name and the fetch request is forwarded to MQS via WebSocket. The Bootstrapper of a node that creates an Isolate System can send messages to isolate system by directly invoking the functions provided by the IsolateSystem. The Bootstrapper can request the information about the isolates this instance of isolate system is running. For which, the IsolateSystem delegates the message to Controller. This request is triggered when a user requests a List of isolates running in an isolate system via a web interface or via RESTful web service of the 'Registry'. The message to stop an isolate is also forwarded to the controller as the IsolateSystem does not directly manage the running isolates. Thus, the message is forwarded to the Controller which is next in the hierarchy. But, when the shutdown command for the isolate system is triggered by the user via web or REST interface, the isolate system closes all the open ports including WebSocket ports and ReceivePorts for isolates, and wait for the 'Garbage Collector' to clean up the memory reserved by it.

Controller

Every Isolate System has a single controller, which is spawned by the IsolateSystem. A controller stays idle until it receives a message to create and isolate from the IsolateSystem. Basically, a controller spawns and manages all the Routers of an isolate system. Additionally, a controller also takes care of 'hot deployment' feature and spawns a 'FileMonitor' for a Router if the feature is enabled. When a RESTART message is received from a FileMonitor, the controller sends a RESART_ALL message to the designated Router, which restarts all the Worker Isolates the Router has spawned. The controller is also responsible for replying to the query of list of isolates an isolate system is running. It achives this by keeping a detailed record of each Router and number of Worker Isolates each Router is handling.

Router

A router is spawned by the controller. It spawns and is responsible for a group of identical workers. Since an isolate is single threaded, multiple instances of an isolate can be created by a router for concurrency. When a message arrives to a router from a controller, the router, based on its defined routing policy, delegates the message to one of the worker isolates. For proper load balancing among a group of isolates, a router uses a routing policy. The routing policies that are available in the framework are:

Round Robin Router Messages are passed in round-robin fashion to child isolates.

Random Router Randomly picks an isolate and the message is passed to that isolate.

Broadcast Router Replicated and sends message to all child isolates.

Worker

Worker isolates carry out the business logic tasks. However, if a certain task is too complex, the isolate can divide the tasks into subtasks and spawn temporary isolates to carry out those subtasks. The temporary isolates, can again spawn other child temporary-isolates to further divide the subtasks into sub-subtask. The temporary isolates terminate once the subtask has been carried out.

Proxy

A 'Proxy' is a special type of Worker. When an isolate is supposed to spawned in a remote node, a proxy isolate is spawned in local node, where the isolate system resides. As the name suggests, the 'Proxy' acts as a worker for the router which spawned it but forwards the messages it receives to the remotely spawned isolate via IsolateDeployer. The communication with Isolate Deployer takes place via WebSocket connection where the proxy worker connects as a WebSocket client. Each 'Proxy' worker maintains a separate WebSocket connection with an 'IsolateDeployer'.

FileMonitor

FileMonitor is also a Worker Isolate. If the 'Hot Deployement' flag for an isolate is set while spawning, a 'FileMonitor' for that isolate is spawned. The spawned 'FileMonitor' monitors the md5 checksum source code of the file from where the Worker Isolate was spawned. If the file is changed, it simply sends the restart command to the controller, which eventually forwards it to the router to restart all of its child isolates.

3.3.2 The Registry

The Isolate Registry is a central registry system where meta-data of isolates and isolate systems are stored. It bootstraps all isolates systems and stores the current deployed location of every isolate system. It's functions are:

RESTful API of Registry

The registry provides a REST API to perform the operations on the connected nodes. One can query the 'Registry' using the 'GET' method of REST to fetch the list of the nodes that are connected to the Registry.

- GET list of connected nodes

- GET details of the running isolate systems on the node
- POST command to shutdown an isolate system
- POST command to kill a particular isolate pool

The functions of Registry are:

- Bootstrap an isolate system, during runtime, in local or remote virtual machine
- Provide a way to deploy, update or remove an 'isolate system'
- Return information about the isolate systems running isolates by querying the individual isolate systems of a node

The registry does not need to persist any data as all the information about isolate systems are queried and generated "on the fly".

Management Interface

The deployment of isolates can be managed through the web interface provided by Isolate Registry

3.3.3 Message Queuing System (MQS)

Since, the basis of this system is message passing, the Message Queuing System is an important part of this framework. The Message Queuing System is an isolate that fetches messages from message broker system and dispatches to respective the isolate system, of connected node, where the isolate belongs. Whenever a new isolate system starts up, the isolate system opens up a new WebSocket connection with the message queuing system. It consists of two major components: Enqueuer – which enqueues messages and Dequeuer – which dequeues messages

Enqueuer

Enqueuer is a separate isolate. A Message Queuing System has only one enqueuer, which basically receives message from the MQS and sends message to message broker system – RabbitMQ 2.6.2 via STOMP 2.7 protocol.

Dequeuer

As opposed to Enqueuer, a Message Queuing System maintains several Dequeuers.

3.3.4 Activator

An activator, basically, brings a node to "life". It starts up two isolates: Systembootstrapper and IsolateDeployer. Every node that is supposed to be running an Isolate System or become a part of Isolate System by running isolates must be running an Activator. Nevertheless, starting up of SystemBootstrapper and IsolateDeployer separately is also possible.

SystemBootstrapper

When a SystemBootstrapper is started, it registers itself to the 'Registry'.

IsolateDeployer

An Isolate Deployer deploys a single isolate in a remote node. It expands the isolate system beyond a physical node, as the isolate system can deploy number of instances of an isolate in different nodes. An Isolate Deployer running in a remote machine can handle requests from multiple 'Proxy Workers' from other systems. Each 'Proxy Worker' opens up a separate WebSocket channel with the Isolate Deployer

3.4 Key Features

3.4.1 Hot Deployment of Isolates and Isolate Systems

It is possible for the source code, of an isolate, to reside in a remote repository and fetched by the controller of a node when required. For instance: isolate source code can reside in a git repository hosted in GitHub. So that as soon as new code is committed in the repository, it gets immediately picked up by the application and the change gets reflected without restarting the application. After a Virtual Machine is bootstrapped, changes like: addition, update or removal of isolates in an isolate system can take place. In such case, the isolates can be killed and redeployed when it has finished processing tasks and is sitting idle. A dedicated isolate monitors changes in the code repository. When a change is detected, the 'FileMonitor' isolate sends a special message to notify the controller that spawned it. The registry takes care of pushing the message to relevant controllers, and the controllers then take care of restarting or removing isolates. This hot deployment capability increases the availability of an application. Whenever there is any change in a component of an application, the whole application does not need to be re-deployed, instead, only the set of isolates that should be updated can be restarted at runtime. This increases overall up-time of the application and keeps other components working even in the time of modification.

3.4.2 Migration of Isolates and Isolate Systems

Relocation of Isolates or an IsolateSystem during runtime i.e. killing a set of Isolates or an Isolate System at one location and bringing up same set of Isolates in another location is the migration of Isolates or Isolate System. The concept of hot deployment and migration brings enormous possibilities in a distributed system. Mostly it improves the availability of the system. Some of them are listed below:

- Migration of actors/isolates allows an application to scale in an easy way. With this capability an application can bring up the most frequently used isolates near to the server where it is accessed the most.
- Related and dependent isolates can be migrated to the same server, if it is evident that it improves performance of the entire system.
- In case of hardware failure on a system which is running a certain set of isolates, migration of actors during runtime can make the application survive the hardware failure.

3.4.3 Remote Isolates

The Isolates in Dart lack functionality of communicating with remote isolates over a network. The isolates in this framework have an ability to communicate with the isolates that may be running in some remote Virtual Machine. So, there can be isolates running in any node. The communication underneath is taken care of by the framework and the implementer who is using this framework does not have to worry if an isolate is remotely spawned or locally spawned. Because of the ability to spawning an isolate in. Two isolates, although, running in two different virtual machines, can still belong to same logical isolate system.

3.5 Typical Message Flow in the System

The framework is based on Fire-and-Forget principle of message sending. The message is serialized before sending via a SendPort of an Isolate.

3.5.1 Enqueuing a Message

Sample format of message at different levels:
Extension of Worker: “Test” - A string

Worker: senderType: senderType.worker, id: mysystem/producer/88f52440-5060-11e4-f396-97cebb949945, action: action.send, payload: sender: mysystem/producer, to: mysystem/consumer, message: Test, replyTo: null

Router: senderType: senderType.router, id: mysystem/producer, action: action.send, payload: sender: mysystem/producer, to: mysystem/consumer, message: Test, replyTo: null

Controller: senderType: senderType.controller, id: mysystem/producer, action: action.send, payload: sender: mysystem/producer, to: mysystem/consumer, message: Test, replyTo: null

IsolateSystem: targetQueue: mysystem.consumer, action: action.enqueue, payload: sender: mysystem/producer, message: Test, replyTo: null

MessageQueuingSystem:

Enqueuer:

3.5.2 Dequeuing a Message

Sample format of message at different levels: Dequeuer: Message Queuing System: IsolateSystem: Controller: Router: Worker: Extension of Worker:

3.5.3 Sending a Message

To send a message from one Worker Isolate to another, simply the 'send' function can be invoked. The 'send' function takes 'message' and 'address' of the target isolate as its argument. The reply path can also be optionally set, so that the replied message from target isolate is sent to totally different actor for further processing. The named parameter 'replyTo' can be used with the address of actor that is supposed to received the replied message.

3.5.4 Asking for a Reply

Sometime the Isolate might just need a reply from another Isolate for further processing or before replying to the sender of the message. In such case, the Isolate can specifically ask the target isolate to reply to this particular instance of Isolate. For instance, a sample use case can be, when an Isolate is maintaining a port connection with a browser. As the port cannot be serialized and passed around through messages, another instance of similar isolate will not be able to serve the reply for the request made through that particular port.

3.5.5 Replying to a Message

3.5.6 Control Messages

As the current implementation of Isolates in Dart does not provide the features to send the control messages like KILL, PING, . Although, it is not implemented yet, as these features are mentioned in documentation of

KILL

RESTART

PING

3.5.7 Some Implementation Overview

Some insight about the message structure to explain how stuffs work in the framework:

How SEND works? When a Worker Isolate sends a message using 'send' function of the Worker class, the message is encapsulated and further information about sender and receiver are added to the message. The message is sent to the spawner, which in this case is the 'Router'. The router agains forwards it to 'Controller' which again forwards to the top level isolate – IsolateSystem. The Isolate System adds another level of encapsulation and information about queue name, in which queue it should be enqueued, to the message so that the Message Queuing System knows the destination queue. If the Worker Isolate is expecting to consume another message after sending a message, it should send a PULL Request for another message, which can be performed by invoking the 'done' function.

How ASK works? Ask function has certain subtle differences from the 'send' function. The 'ask' function should be used when the sender of the message expects something in reply. The Worker class adds the full path of the isolate along with the unique-id of the isolate when an 'ask' message is constructed. This is to make sure that the response from the target isolate reaches this particular instance of the isolate. The Router, which can also be called a load balancer, when receives a message with full address of isolate, routes the message to the isolate with the unique-id contained in the message. The message is simply discarded by the Router, if the isolate with the given unique-id is not found in the list of isolates the Router is maintaining. This is possible when the isolates have been restarted or for some reason the isolate was killed.

How REPLY works? The 'reply' function is simply a convenience for the implementer. The 'reply' function simply invokes 'send' message with the sender's address as the target isolate. If the message contains 'replyTo' then the message will be replied to the address contained in 'replyTo' instead of the original sender. The 'reply' function can be used to reply message in both – 'send' and 'ask' cases.

How Kill works? The is a special control message sent to the isolates as well as isolate system to shutdown themselves. If a KILL message is sent to an Isolate, the message is queued at the last of the isolate, (no futher messages after KILL message? in isolate level or router level, check code ! I think the router buffers the messages until the isolates are restarted and start sending them again once they come back up). Once it finishes processing the already queued messages and encounters the KILL message, the isolate closes its ReceivePort[1] and sits idle. After sometime it gets Garbage Collected and cleaned up. However, the implementation is slightly different because even after waiting an isolate to be Garbage Collected, the memory sometimes does not get freed up as expected. As a workaround for this, the isolate made to throw a custom Exception when it receives KILL message. This way it is sure to get shutdown forcefully and the memory it kept occupying gets freed.

How Restart works? Restarting and Isolate is basically kill an isolate and spawning it up again. However, during restart the messages that arrive to the router after issuing KILL message are buffered in the router itself and are sent out once the isolates are spawned. For instance, when the 'HotDeployment' feature is enabled during the time of deploying an Isolate in an Isolate System, if the source code of the isolate is modified and saved, then the each of the isolates that the router has spawned gets restarted. During which messages that arrive after RESTART message are buffered in the spawning router.

How Shutdown IsolateSystem works? An Isolate System that is running in a node can be shutdown via Web Interface or via POST request to the Registry. When a request to shutdown a complete IsolateSystem is sent, the IsolateSystem closes all the ports including the WebSocket connection with Message Queuing System and the forceful shutdown is carried out by throwing out and Exception as a workaround to free up the memory consumed after it is shutdown.

[1] A Worker Isolate receives message from Router via a ReceivePort

3.5.8 Clustering

3.6 Dart Libraries Used in Construction

3.6.1 STOMP

3.6.2 Path

3.6.3

3.7 Sample Implementation of Worker using the Framework

4 Case Study

4.1 A Sample Producer/Consumer Program

4.2 A Sample Requester/Supplier Program

4.3 System Setup for Test

* Used Amazon EC2 instances * A File Server that serves file over HTTP * A RabbitMQ server with one Message Queuing System * 3 other Message Queuing Systems * 32 Nodes

5 Results

5.1 Benchmarkings of Producer/Consumer Program

5.2 Benchmarkings of Requester/Supplier Program

6 Discussions

6.1 Performance Findings

6.2 Non-Functional Requirements of the Framework

6.2.1 Scalability

Scalable at different component levels. Isolates, Isolate System, Message Queuing System and clustering of Message Broker (RabbitMQ). Better scalability is obtained when the Message Queuing System is scaled out.

6.2.2 Availability

Can be designed to make highly available. Apparently, no downtime to deploy new code to a running system.

6.2.3 Reliability

6.3 Problems/Issues

6.3.1 Dart Induced Issues

* The Unimplemented functionalities of Isolates - KILL - PAUSE - PING - Supervision of Isolates * Isolate are not lightweight. * Running Isolates in web (Dartium) *

6.3.2 Performance

7 Conclusion

7.1 TODO

8 Recommendations

8.1 TODO

9 Appendices

9.1 TODO

List of Figures

List of Tables

Bibliography

[Agh85] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. English. 1985, pp. 34–35.

[AH85] G. Agha and C. Hewitt. "Concurrent programming using actors: Exploiting large-scale parallelism." English. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Maheshwari. Vol. 206. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1985, pp. 19–41. ISBN: 978-3-540-16042-7. DOI: 10.1007/3-540-16042-6_2.

[Arm07] J. Armstrong. "A History of Erlang." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pages. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238850.

[ECM14] ECMA. *ECMA-408: Dart Programming Language Specification*. 2014. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), June 2014.

[FM11] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). Internet Engineering Task Force, Dec. 2011.

[Goo] Google. *Dart API Reference*. https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:isolate.Isolate. Last Accessed: 2014-11-19.

[Hal] P. Haller. *Scala Actors*. http://www.scala-lang.org/old/node/242. Last Accessed: 2014-11-19.

[HO09] P. Haller and M. Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming." In: *Theor. Comput. Sci.* 410.2-3 (Feb. 2009), pp. 202–220. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.09.019.

[Inca] T. Inc. *Akka Toolkit*. http://akka.io. Last Accessed: 2014-11-19.

[Incb] T. Inc. *Akka Toolkit*. http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf. Last Accessed: 2014-11-19.

[Kat12] S. L. Kathy Walrath. *Dart: Up and Running*. Last Accessed: 2014-11-19. O'Reilly Media, Oct. 2012. ISBN: 978-1-4493-3084-2.

[OR14] M. Odersky and T. Rompf. "Unifying Functional and Object-oriented Programming with Scala." In: *Commun. ACM* 57.4 (Apr. 2014), pp. 76–86. ISSN: 0001-0782. DOI: 10.1145/2591013.

[Pip] A. Piper. *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP*. http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html. Last Accessed: 2014-11-22.

[Sofa] P. Software. *RabbitMQ Performance Measurements, part 2*. http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/. Last Accessed: 2014-11-20.

[Sofb] P. Software. *Sizing your Rabbits*. https://www.rabbitmq.com/blog/2011/09/24/sizing-your-rabbits/. Last Accessed: 2014-11-22.

[Sofc] P. Software. *What can RabbitMQ do for you?* http://www.rabbitmq.com/features.html. Last Accessed: 2014-11-20.

[STO] STOMP. *STOMP - Simple Text Oriented Messaging Protocol*. http://stomp.github.io/stomp-specification-1.2.html. Last Accessed: 2014-11-20.

[Vin07] S. Vinoski. "Concurrency with Erlang." In: *Internet Computing, IEEE* 11.5 (Sept. 2007), pp. 90–93. ISSN: 1089-7801. DOI: 10.1109/MIC.2007.104.