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 Message Passing Paradigm

Message passing is simply sending a message from one process, component or actor to another. The recipient of the message chooses further processing based on the pattern or content of the message. Message passing is the loosest type of coupling. The components of a system are not dependent on each other; instead they use a public interface to exchange parameterless messages[Cel11]. Hence, systems that implement message passing paradigm are easily scalable and efficient. Such systems are easy to replicate and thus fault tolerance can be improved [Arm10].

The systems which communicate solely by message passing do not have a shared state. Such systems can be easily divided into isolated components. This makes the overall architecture of a system easy to understand and simplifies the separation of problems within the system [Arm10].

Message passing can either be synchronous or asynchronous. Synchronous message passing is the communication between two components where both the sender and the receiver of a message are ready to communicate. As the sender of a message has to wait for a response from the receiver, the sender is usually blocked until it gets a reply. Whereas, in asynchronous message passing, the sender simply sends a message and carries on with further processes. The receiver does not have to be ready to accept the message when the sender sends it. [Agh85] Thus, an asynchronous message passing is non-blocking in nature.

2.2 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 embodies three essential things:

- Processing

- Storage
- Communication

Actors have addresses and there is a many-to-many relationship between actors and addresses. i.e Each actor can have multiple addresses and multiple addresses can belong to one actor.

Actors communicate with each other in a non-blocking way by asynchronous message passing, which removes the need of explicit locks. An actor can send message to actors in same system or another system. An actor can also send message to itself, which is how a recursion is achieved. An actor can send a message to target actor only if it has the address of target actor. Agha ([Agh85], p35) lists three ways in which an actor, upon accepting a message, can know the address of the target actor:

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

To buffer incoming messages, 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 [Nin].

After receiving a message, an actor may perform following actions. [AH85]

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

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

An actor exists 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 [Nin].

Several programming languages like Act 1, 2 and 3, Acttalk etc. were created when actor system was newly introduced by Hewitt and Agha [Agh85; AH85].

2.2.1 Error Handling in Actor Model

Exception handling in actor model is based on idea of embracing the failures. The idea of embracing failures is also known as "let it crash" paradigm [subsection 2.3.1]. As the actors do not have shared state, this allows individual actor to fail without causing disruptions in the system. Since, an actor in an actor system is typically organized in a hierarchical structure, the actor which created a child actor can be used for supervising it. The idea of supervision in a hierarchical actor system helps to make the actor system fault tolerant [Erb12]. When an actor throws an exception the supervisor can respond to the exception in different ways. In Akka [section 2.5], the supervising actor usually reacts by either simply ignoring the exception and let the actor continue, restarting the actor or by escalating the error to its supervisor.

The "let it crash" style of programming is a non-defensive programming, which is implemented successfully in Erlang [section 2.3].

2.2.2 Differences from thread-based programming model

Thread-based Concurrency

In thread-based programming languages, the control flow of a program is divided into several threads for concurrency. The threads operate simultaneously and the control can switch from one thread to another non-deterministically. When two or more threads have a shared memory, concurrent modifications and accesses of the data might result in undesired behavior of the system, known as 'race condition'. To prevent this type of situations, such programming languages have locks. The locks make only single thread, to run sequentially, at a time for a section of program code [Sofa].

Generally, the thread-based programming models are easy to understand and implement. But, the resulting program behavior is difficult to understand because of implicit context switches and release of locks, which may even lead to a deadlock situation [Sofa].

Actors based Concurrency

Concurrency in actor based programming languages are inherent because of using asynchronous message-passing, pipelining, and the dynamic creation of actors. The concurrency in actors through pipelining is only constrained by the logical limits and the available hardware resources [Agh85]. The actors may carry out their activities in parallel as each actor resides in a completely separated space from the other actors, they are connected only via messages.

The actor based programming liberates the programmer from going into coding details about the parallelism and threads [Agh85].

As actors do not have shared state, thus the 'race condition' as discussed in section 2.2.2, do not arise in actor based programming.

2.2.3 Scaling up actor based systems

As actor based systems are highly concurrent [section 2.2.2], it is easy to scale actor based systems from a single core system to several of them across multiple data centers around the globe. The ideal case would be to just let a new system join the actor system and let it run some more actors during runtime, without the need of taking down the running system. The actors themselves do not need to know the physical location of other actors as they simply exchange messages based on the logical addresses. This makes them easy to scale up and out.

2.3 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 'processes,' which are essentially user-space threads rather than Unix processes or kernel threads, communicate only via message passing.

Erlang was designed for the writing applications that require high availability - as Armstrong mentions it in his [Arm07] the programs that "run forever". It uses concurrent processes, which have no shared memory and communicate only via asynchronous message passing. This idea is similar to the actors model proposed by Agha and Hewitt section 2.2. Thus, programs written in Erlang are concurrent, distributed, fault tolerant and thread safe. The concurrency is built into the language itself, not the the operating system [Arm10].

When an application developed in Erlang is deployed in a multicore computer, it automatically takes advantage of those multiple cores. The Erlang "processes" spread over the cores and so the programmers do not have to worry about threads [Arm10]. Erlang has mechanisms to let the programs update themselves without taking the system offline. The new changes in an application can be added while the system is up, which improves the availability of whole system. Thus, simplifying the construction of software for implementing non-stop systems [Arm07].

Error handling in Erlang is different from most of the other programming languages. The error handling is based on a "let it crash" philosophy subsection 2.3.1 which is a non-defensive style of programming [Arm07; Arm10].

2.3.1 "Let it Crash" Philosophy

The core idea in "Let it Crash" philosophy is to let the failing processes crash and make other processes, which observe this process, detect the crashes and fix them [Arm10]. This idea is in sharp contrast to other programming languages, where programmers implement exception handlers and prevent a process from getting terminated.

The concept of "let it crash" leads to clear and compact coding [Arm10].

2.4 Scala Actors

Scala is a statically typed programming language which integrates functional as well as object-oriented programming[OR14]. Scala has its own library for actor programming.

In Scala, templates for actors with user-defined behavior are normal class definitions which extend the predefined Actor class.

The scala actors library provide concurrent programming model based on actors [section 2.2]. The actors in scala are fully inter-operable with the mainstream virtual machine threads [HO09]. Scala actors are lightweight and support around 240 times more actors to run simultaneously compared to virtual machine threads [HO09].

Both synchronous and asynchronous message passing can be used to in scala actors. The synchronous message passing is implemented by exchanging several asynchronous messages. The actors in scala can also communicate using 'futures'. When a future is used the requests are handled asynchronously and the sender immediately gets a representation of the future which allows sender to wait for the reply. [Hal]

The Scala Actors library is now deprecated and will be removed in future Scala releases[1]. The deprecation is in favor of the use of Akka actors [section 2.5].

2.5 Akka Toolkit

Akka is a toolkit and runtime for building highly concurrent, distributed and resilient message-drive applications on the JVM. Akka uses lightweight actors for concurrency. The actors in Akka are based on Hewitt's and Agha's model [Agh85; AH85] of actor

[1]Scala actors are deprecated in version 2.10, P. Haller and S. Tu. *Scala Actors API*. `http://docs.scala-lang.org/overviews/core/actors.html`. Last Accessed: 2014-11-30

programming. It provides the developer with a well defined API to develop large concurrent systems and allow for easy scaling out of a single machine.

2.5.1 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());
```

2.5.2 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>

2.5.3 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?

2.5.4 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.

2.5.5 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.

2.5.6 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.

2.5.7 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.

2.5.8 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());
```

2.5.9 Akka in Distributed Systems

2.5.10 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.

2.6 The Dart Language

2.6.1 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 type checking 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.

2.6.2 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.

2.6.3 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.

2.6.4 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

2.6.5 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.

2.6.6 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[2] implementation of an Isolate in Dart is — similar to a thread with properties like an actor.

2.7 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 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.[Sofd]

- 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]

[2] Dart version 1.7.2

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 2.8 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.

2.7.1 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.[Sofc]

2.7.2 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 [3] which shows how the throughput of a queue varies when prefetch count is changed when there are different number of consumers.

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

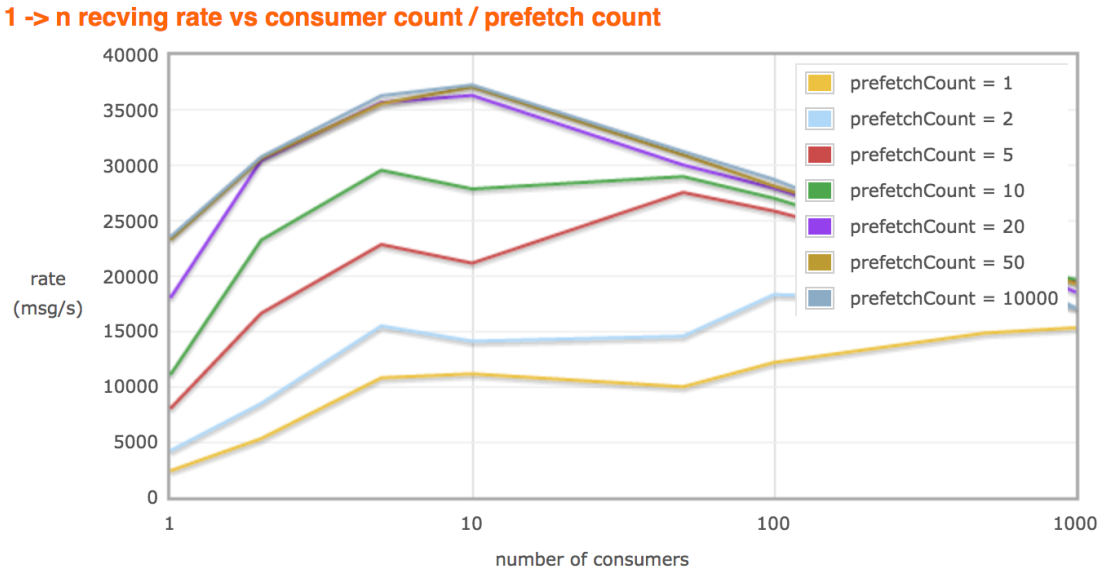

Figure 2.1: Chart showing performance variation when prefetch count is changed in the consumer[4]

2.8 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

[4]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

constraints on the architecture of servers and many features such as destination naming and reliability semantics are implementation specific.

2.8.1 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.

2.8.2 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 2.7.2 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.

2.9 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.

2.9.1 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].

2.9.2 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.

2.9.3 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

2.10 RESTful WebServices

3 System Design

3.1 Core Design Decisions

The Framework is designed to be distributed in nature with the concepts of Actors [section 2.2]. It should follow the standard actor programming concept and provide inherently distributed nature to the applications built on top of it. The framework itself should be built using the concept of 'message-passing' [section 2.1] to alleviate any possibility of concurrency issues, thus thread-safe.

- The framework may not guarantee the delivery of a message.
- All the messages sent by the framework will be based on 'fire and forget' concept.
- A message shall be delivered at most once.
- A message should always be routed through Message Queuing System, even though the target isolate may belong to same isolate system in the same logical or physical node.
- Feeding of message to an isolate should be based on pull mechanism, not push mechanism.
- Exceptions thrown at child isolates shall be handled by a 'spawnee' of that isolate. Hence, implementing the idea of supervision and "let it crash" ideology[**??**].

3.2 The Framework

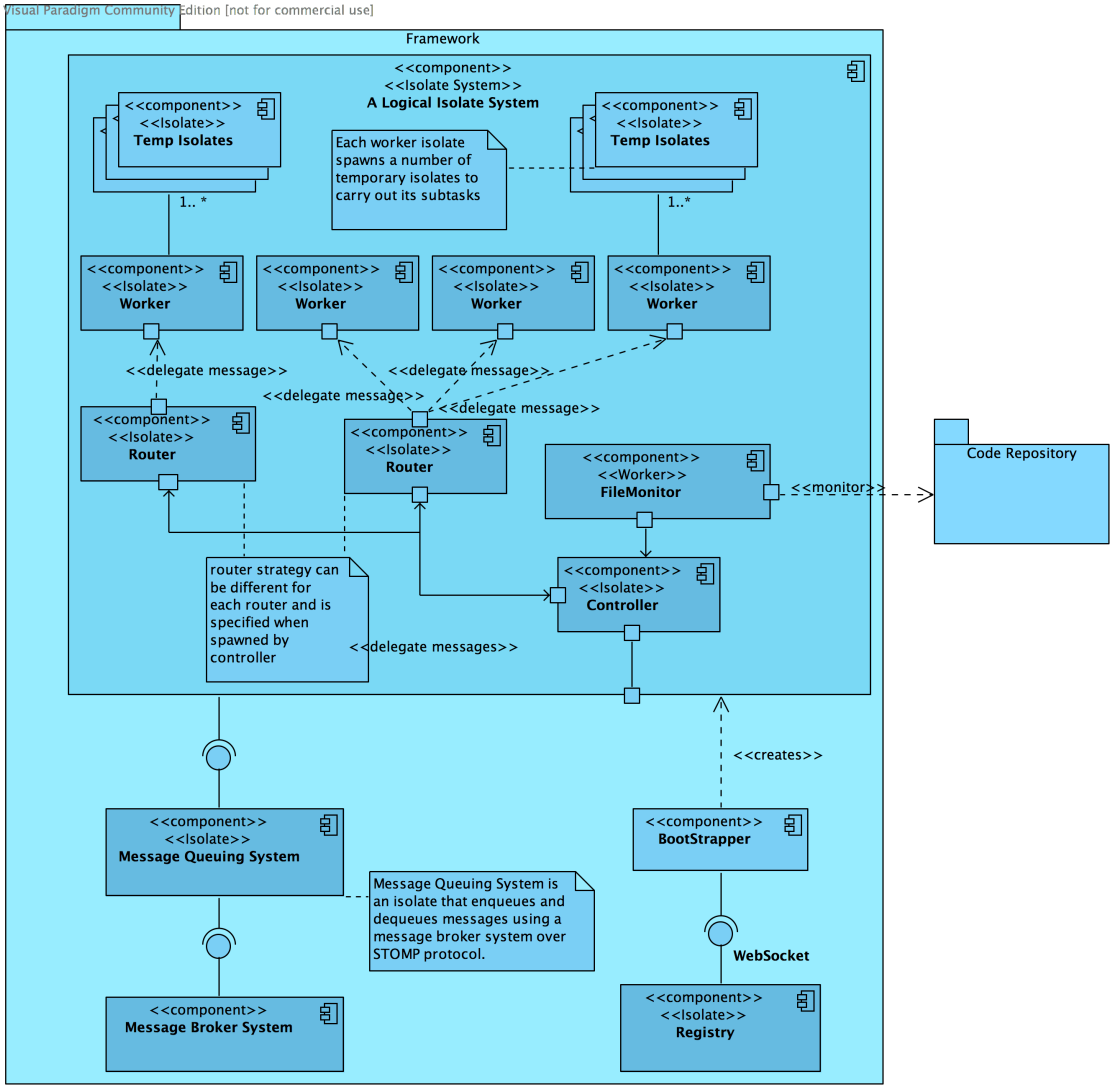

Figure 3.1: Architecture of the framework

The framework comprises of an Isolate System, a Registry, a Message Queuing System, a Message Broker System and an Activator. The Figure 3.1 depicts the general overview of different components and sub-components of the framework.

3.2.1 Isolate System

An Isolate System is analogous to an actor system. Just as an actor system consists of a group of actors working together, an isolate system is composed of a group of 'Worker' isolates working closely. It consists of different hierarchies which forms a logical organizational-like structure. The top level isolate is the Isolate System itself and the bottom most are the 'Worker' isolates.

A 'Bootstrapper' in physical node can start up several Isolate Systems. Nevertheless, a logical Isolate System is not limited to a single physical node. The 'Worker' 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, an isolate system needs the WebSocket address of Message Queuing System, and a 'name' for itself. The name is simply an alias, and should not be confused with the unique id as another isolate system with the same name can exist in other nodes but the unique id is exclusive for a particular instance of an isolate system.

The bootstrapping of an isolate system includes: generating a new id which is unique for itself, opening up a 'ReceivePort', and connecting to a 'Message Queuing System'. After opening up a 'ReceivePort', the isolate system starts listening on that port for messages so that it can receive incoming messages from 'Controller'. While connecting to the Message Queuing System, if a connectin could not be established, it simply keeps on retrying at certain interval. Furthermore, should the connection be lost at any time after being connected with the MQS, the Isolate System automatically keeps on trying to re-establish the connection at regular interval. Since, the connection to MQS takes place asynchronously, the isolate system simply moves forward and spawns a controller, regardless of the establishment of connection with the MQS.

Adding Worker Isolates to The Isolate System

As an isolate system is a top level isolate, it spawns a controller. The controller spawns one or several routers and each router spawns a worker isolates. The Figure 3.2.1 shows the message flow sequence in different components while starting up an isolate system and deploying a worker isolate in it.

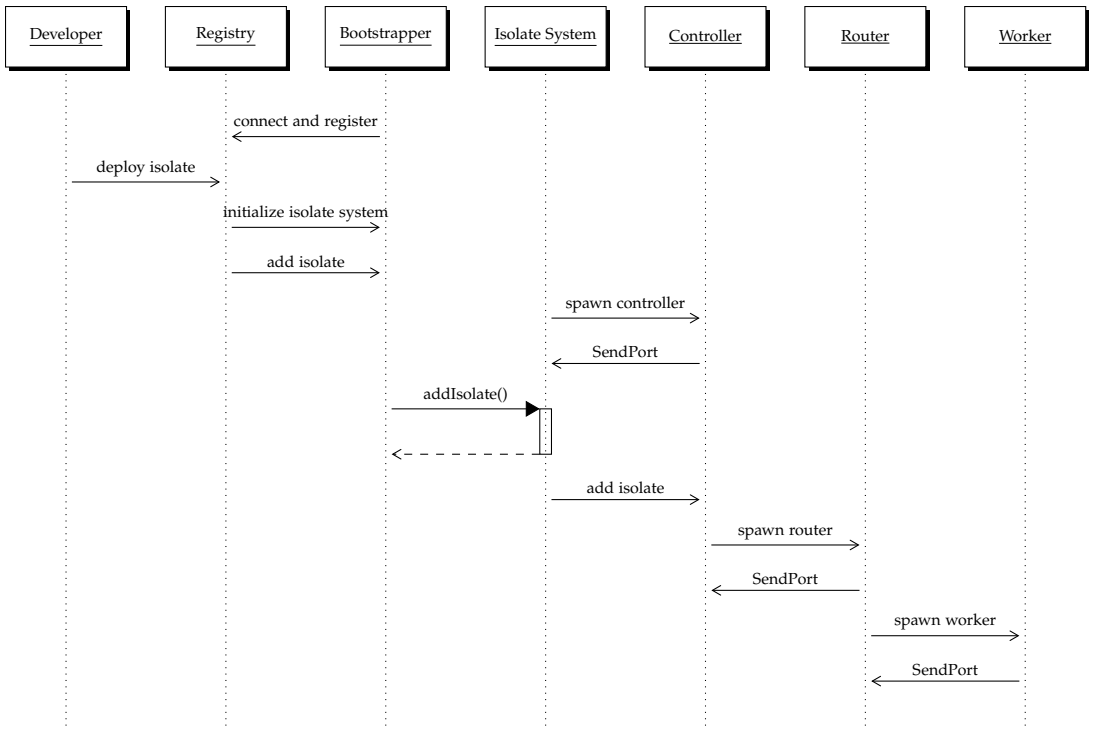

Figure 3.2: Adding a Worker isolate to an isolate system

When an isolate system is first initialized, it is an empty system without any Workers running in it. The Workers can be added with appropriate load balancer once the isolate system has been initialized. The 'addIsolate' method starts up worker isolates into the isolate 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 isolate 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 node, the location can be repeated. For instance ["localhost", "localhost"] results in spawning of two identical isolates in local machine which is load balanced by the type of specified router.

routerType – The type of load balancing technique that one would like to use to effectively distribute incoming messages. By default, the framework, provides three types of routers: Round-Robin, Random and Broadcast. If the developer wants to use his own customized load balancer instead of using the options provided by the framework, the absolute path to the location of source code, which can also be a remote URI, of the custom router implementation can be provided.

hotDeployment – This argument is optional and is set to 'true' by default. Setting it to 'true' enables continuous monitoring of the source code. 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 redeploying the system.

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 The Top Level Isolate System

Typically, a message in an isolate system 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 [subsection 2.6.6], only one message is handled at a time.

A Message Queuing System sends the data over WebSocket in JSON string format, which should be deserialized to Map data type before further processing. As the received message contains the queue name from which it is dequeued, the name of the queue is then parsed and transformed to name and address of the corresponding isolate. The message is then forwarded to the Controller that this instance of the isolate system has spawned.

The messages arriving from Controller is either a dequeue requests or a message that should be sent to the Message Queuing System for enqueuing. The dequeue requests are sent from the isolates that have completed certain task and are ready to accept another message. For the dequeue requests, the sender of the message is identified, which is used to figure out the corresponding name of the queue name. Then the pull request to dequeue from that queue is forwarded to MQS via open WebSocket port.

For the messages that are supposed to be delivered to another isolate, the name of the target isolate is used to figure out the name of the queue and then sent to the MQS for enqueuing.

The 'Bootstrapper' of a node that creates an isolate system can send messages to isolate system by directly invoking the functions provided by the isolate system. The bootstrapper can request the information about the isolates this instance of isolate system is running. For which, the isolate system delegates the message to its controller and waits asynchronously for the response from the controller. The request, to fetch a list of running worker isolates, is triggered when a user sends the request to view details of an isolate system via a web interface or via RESTful web service provided by the 'Registry' subsection 3.2.2.

Another type of message is the message to terminate a worker isolate, which is also forwarded to the controller as the isolate system does not directly manage the running worker isolates. Thus, the message is forwarded to the controller which is next in the hierarchy. In contrast, when the shutdown command for the isolate system is triggered via web or REST interface, the isolate system closes all the open ports including WebSocket ports and ReceivePorts, and then wait for the 'Garbage Collector' to and clean up the memory reserved by it.

Controller

Every isolate system has a single controller, which is spawned by the top level isolate of the isolate system. A controller stays idle until it receives a message to create an isolate. Basically, a controller spawns and manages all the routers of an isolate system. Additionally, a controller takes care of the 'hot deployment' feature for which it spawns a 'FileMonitor' for each router if the feature is enabled. When a RESTART message is received from a FileMonitor, the controller sends a RESTART_ALL message to the designated router, which restarts all the Worker isolates the router has spawned.

A controller is also responsible for replying to the query of list of isolates an isolate system is running. It achieves this by keeping a detailed record of each Router and number of Worker isolates each Router is handling, which is updated as soon as an isolate is killed or a new isolate is added.

As a Controller is the 'spawner' of Routers and the 'spawnee' of the top level isolate, it forwards the messages as well as dequeue requests coming from Routers to the top level isolate of an isolate system.

Router

A router is spawned by a controller. The router creates and is responsible for a group of identical Workers isolates. Since an isolate is single threaded, creation of multiple instances of an isolate is desirable for concurrency. When a message arrives in a router from a controller, the router, based on its defined routing policy, delegates the message to one of the worker isolates. The routing policy can be chosen at the time of deployment of a worker isolate.

A router uses a routing policy to distribute message among the group of isolates it is handling. The default routing policies that are available in the framework are listed in the Table 3.1

Table 3.1: List of routing techniques provided by the framework

Router	Description
Round Robin	Messages are passed in round-robin fashion to its Worker isolates.
Random	Randomly picks one of its Worker isolates and sends the message to that Worker isolate.
Broadcast	Replicates and sends message to all of its Worker isolates.

In addition to the available routing policies of the framework, it is also possible to add a new Routing technique by simply extending the 'Router' class which requires 'selectWorker' function to be implemented. The overridden 'selectWorker' function may either return a list of Workers or a single Worker. The ability to implement a custom router opens up possibilities for numerous load balancing techniques. For instance, a simple multicasting router that replicates a message only to the Workers that are spawned locally can be implemented by selecting such Workers using their deployment paths and returning them as a 'List'.

As the router manages the Worker isolates it has spawned, it is responsible for effectively terminating and restarting the Worker isolates. It also buffers the messages that might arrive while the workers are not ready to accept the messages yet; usually, during the creation of Worker isolates and while restarting them.

If a router does not receive any message from a Worker for a certain amount of time, the router sends a PING message to check if the Worker isolate is alive and ready to accept more messages. If the Worker isolate responds with a PONG message, the router sends a request to fetch messages to controller. This mechanism is present in the framework to prevent the 'starvation' for a Worker isolate in case the dequeue message, that might have been sent earlier, could not reach the Message Queuing System because

of a network issue or unavailability of MQS.

Worker

The 'Worker' of the framework is an abstract class, which should be inherited by the isolate that the programmer creates. The 'Worker' first unwraps the messages that arrives from the router and retrieves headers from it. 'Sender' and 'replyTo' headers of the message is collected before forwarding message to the child class, that extends this abstract class. By unwrapping the messages that is encapsulated by various headers, the abstract Worker class makes sure that the message is delivered to the target implementation of the Worker isolate immutated and in intended form.

To extend the 'Worker' isolate, one must implement 'onReceive' function which handles incoming messages and carry out the business logic tasks. However, if a task is too complex, the Worker isolate can divide the tasks into subtasks and spawn temporary isolates to carry out those subtasks concurrently. The temporary isolates can be terminated once the subtask has been carried out.

The 'send', 'reply' and 'ask' functions are provided by this abstract Worker class to send a message to another worker isolate. These functions automatically add the information of sender and receiver in the header of the message that is sent out.

Sending a Message To send a message from one Worker isolate to another, the framework provides 'send' function. It takes 'message' and 'address' of the target Worker isolate as its argument. The reply path can also be optionally set, so that the replied message from target isolate is sent to a different worker isolate for further processing. The named parameter[1] 'replyTo' can be used with the address of actor that is supposed to received the replied message; eg:

```
send("A simple text message", "demosystem/printer");
send("Another message", "demosystem/jsonConverter", replyTo: "demosystem/printer");
```

Asking For a Reply Sometimes a Worker isolate might need a reply from another isolate for further processing or before replying to the sender of the message. In such case, the worker isolate can specifically ask the target isolate to reply to this particular instance of worker isolate.

For instance, a sample use case can be, a worker isolate maintaining a connection with a browser via HTTP. In this case, as the port cannot be serialized and passed to other isolates through messages, another instance of similar isolate will not be able to respond to the request made in that connection.

[1] Dart's named paramter feature

Similar to the 'send' function, the 'ask' function takes 'message' and 'address' of the target worker isolate as its argument; eg:

```
ask("current time", "demosystem/timeKeeper");
```

Replying to a Message To reply to a message, the framework provides the 'reply' function. It expects a single argument — 'message', because the respose is sent to the worker isolate specified by the sender. The 'reply' can be used in response to any of 'send' or 'ask' messages; eg:

```
reply("Current time is: $time");
```

Proxy

A 'Proxy' is the special type of a Worker isolate. When a Worker isolate is supposed to be spawned in a remote node, the router instead spawns a Proxy isolate in local node. Once the Proxy isolate is created, it connects to the 'Isolate Deployer' of the remote node where the Worker isolate is intended to be spawned. After establishing connection over a WebSocket with Isolate Deployer of the remote node, the proxy isolate forwards the request to spawn the worker isolate to the Isolate Deployer. After successful spawning of isolate in the remote node, the proxy isolate simply forward the messages that is sent to it by the spawner router. Each proxy worker maintains a separate WebSocket connection with an 'Isolate Deployer'.

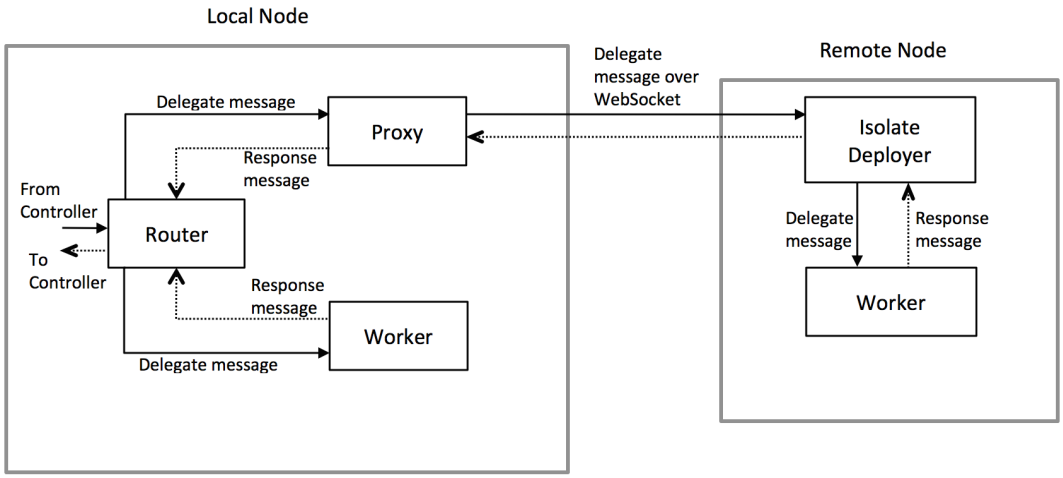

Figure 3.3: A Proxy Worker

FileMonitor

The controller spawns a 'FileMonitor' only if the 'Hot Deployement' flag for an worker isolate is set while deploying. The spawned 'FileMonitor' monitors the md5 checksum of the source code using which the Worker isolate is spawned. If a change in the source file is detected, it simply sends a RESTART command to the controller, which eventually forwards it to the target router. The router then restarts all of its worker isolates.

3.2.2 The Registry

The Isolate Registry is a central node where other nodes, which are running Bootstrapper, connect and register themselves. The registry simply keeps the record of the connected nodes, assigns a unique id to each and queries them about the running isolate systems when required. The registry provides RESTful API and a web interface /footnotethe web interface can be opted out as it has to be started separately through which one can have an overview of the full system and manage the deployments of the isolate systems as well as individual isolates.

The basic tasks that a registry carries out are:

- Bootstraps an isolate system, during runtime, in local or remote node
- Provides a way to deploy, update or remove an isolate system
- Returns information about the deployed isolates by querying the individual isolate system of a node.

RESTful API of Registry

The registry provides a REST API to perform the operations on the connected nodes. One can send a 'GET' request to the registry to fetch the list of the nodes that are connected to the registry. Using the 'id' of a node from the replied list, one can deploy an isolate system or add an isolate to already deployed system by sending appropriate 'POST' request.

The REST endpoints exposed by the registry are listed in table Table 3.2.

A sample GET and POST query to deploy an isolate system

GET request to fetch a list of connected systems

Request: 'GET `http://54.77.239.254:8000/registry/system/list`'
Response Status Code: 200 OK
Response Body:

Table 3.2: List of endpoints exposed by the registry

Method	Endpoint
GET	/registry/system/list
GET	/registry/system/{bootstrapperId}
POST	/registry/deploy
POST	/registry/system/shutdown

```
[
  {
    "bootstrapperId": "266393094",
    "ip": "54.77.239.244",
    "port": "50189"
  },
  {
    "bootstrapperId": "12133208",
    "ip": "54.77.239.243",
    "port": "50192"
  }
]
```

POST request to deploy an isolate system

Request: 'POST `http://54.77.239.254:8000/registry/deploy`'

Request Body:

```
{
  "bootstrapperId" :"266393094",
  "action": "action.addIsolate",
  "messageQueuingSystemServer": "ws://54.77.239.200:42043/mqs",
  "isolateSystemName" :"sampleSystem",
  "isolateName" :"consumer",
  "uri" :"http://54.77.239.221/sampleSystem/bin/Consumer.dart",
  "workersPaths" :["localhost",
     "ws://54.77.239.243:42042/activator"],
  "routerType" :"random",
  "hotDeployment" :true
}
```

Response Status Code: 200 OK

A sample GET query to fetch details of an isolate system

GET request to get details of an isolate system
Request: 'GET `http://54.77.239.254:8000/registry/system/266393094`'
Response Status Code: 200 OK
Response Body:

```
{
  "sampleSystem": [
    {
      "id": "sampleSystem/consumer",
      "workerUri":
          "http://54.77.239.221/sampleSystem/bin/Consumer.dart",
      "workersCount": 2,
      "workersPaths": [
        "localhost",
        ws://54.77.239.243:42042/activator"
      ],
      "routerType": "random",
      "hotDeployment": true
    }
  ]
}
```

A sample POST query to terminate a Worker isolate

POST request to terminate an isolate of an isolate system
Request: 'POST `http://54.77.239.254:8000/registry/system/shutdown`'
Request Body:

```
{
  "bootstrapperId" :"266393094",
  "isolateSystemName" :"sampleSystem",
  "isolateName" :"consumer"
}
```

Response Status Code: 200 OK

An example of terminating an Isolate System

POST request to terminate an isolate of an isolate system
Request: 'POST `http://54.77.239.254:8000/registry/system/shutdown`'
Request Body:

```
{
  "bootstrapperId" :"266393094",
  "isolateSystemName" :"sampleSystem"
}
```

Response Status Code: 200 OK

The registry generates all the information about isolate and isolate systems "on the fly". Thus, it does not need to persist any data.

The Web Interface for the Registry

The deployment of isolates can also be managed by using a web interface provided by the registry. The Web Interface should be started up separately in a different port. The Web Interface, internally communicates with the registry via the REST API [3.2.2] which is exposed by the Registry.
<TODO: A Screenshot here or in the Appendix>

3.2.3 Message Queuing System (MQS)

Since, the basis of this system is message passing, the Message Queuing System is an important component of this framework. The MQS is a top level isolate that fetches messages from message broker system and dispatches to respective isolate of the isolate system. Whenever a new isolate system starts up, it opens up a new WebSocket connection with the MQS. The messages are exchanged between the isolate system and the MQS through the WebSocket connection. The MQS keeps track of the unique-id of an isolate system so that it can identify the origin of the message.

If a message is supposed to be enqueued, the MQS ignores the unique-id and simply forwards the message to the 'Enqueuer' isolate. Whereas, if the message is a dequeue request, the MQS forwards the the message to a 'Dequeuer' isolate along with the unique-id of the isolate system. The unique-id is used to identify the WebSocket port through which the request arrived. Thus, making sure that the dequeued message is sent to the correct requester. This is especially required if a cluster, of identical isolate systems, is running on different nodes.

The MQS should be started up separately along with few command line arguments to connect to message broker system. The required command line arguments are: ip address, port, username and password to connect to Message Broker System. The 'prefetchCount' is an optional argument which defaults to 1, if not provided explicitly. A 'prefetchCount' is a Quality of Service header for Message Broker System which allows a subscriber of a queue to hold the defined quantity of unacknowledged messages.

Since, in Dart[2] the passing of sockets to isolates is not yet possible, so the main isolate has to pipe all the input/output data. In this case the MQS is the top level isolate which has to handle all incoming and outgoing messages.

Enqueuer

An enqueuer is a separate isolate. A Message Queuing System has only one enqueuer, which basically receives messages from the MQS and sends messages to a message broker system – RabbitMQ [2.7.2] via STOMP [2.8] protocol.

Dequeuer

As opposed to Enqueuer, a Message Queuing System maintains each dequeuer for each topic. The topic corresponds to each router running in the isolate system. Whenever a message arrives from a new isolate, the MQS spawns a new dequeuer isolate. The dequeuer then subscribes to a new message queue in the message broker system via STOMP [2.8] protocol. If the queue does not exist in the message broker system, the message broker system automatically creates the queue.

If a dequeuer is idle for too long, i.e. if the Dequeuer isolate has not received any dequeue requests for certain interval[3], then the MQS terminates the dequeuer isolate for that particular queue. Nevertheless, as soon as the MQS receives a dequeue request, it spawns a new Dequeuer, if one does not exist yet.

The dequeuer subscribes messages from Message Broker System with such options that the new messages do not arrive to the subscriber unless previously dequeued messages have been acknowledged. This throttles the flow of messages from message broker system and keeps itself and the isolates from being overwhelmed by large number of messages, which might induce 'out of memory' issues.

Messages in dequeuer keeps on arriving as long as there are messages in the queue and the messages are being acknowledged. The messages that are in the buffer of dequeuer stay in unacknowledged state unless they are flushed and sent out to the

[2]Dart version 1.7.2
[3]by default the timeout is 10 seconds

requesting isolate of an isolate system. As soon as a message is acknowledged the dequeuer receives another message from the message broker.

Multiple Instances of MQS

It is possible for a system to have multiple Message Queuing Systems for scaling up the system. If there are multiple identical isolate systems connected to different instances of MQS, each MQS will have a dequeuer which subscribes to the same queue. Nevertheless, a message is dispatched by the message broker system to only one of the dequeuers, which is distributed in round robin fashion. Thus, messages are fairly distributed among the subscribers.

3.2.4 Activator

An activator simply starts up two isolates: a 'System Bootstrapper' and an 'Isolate Deployer'. 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. The activator requires a WebSocket address of the Registry as a command line argument. Nevertheless, it is also possible to start up System Bootstrapper and Isolate Deployer separately.

System Bootstrapper

The System Bootstrapper registers itself to the 'Registry' via a WebSocket connection as soon as it is started. The activator forwards the path of the WebSocket to the system bootstrapper. But, if a System Bootstrapper is started separately then the path of the Registry should be passed as a command line argument.

Isolate Deployer

An Isolate Deployer starts up a Worker isolate in a node. The isolate is spawned without a local isolate system and as a part of an isolate system running in another node. This functionality expands the isolate system beyond a physical system. An isolate system can deploy number of instances of an isolate in several different nodes.

An isolate deployer running in a remote machine is able to handle requests from multiple 'Proxies' from several isolate systems. Each proxy opens up a separate WebSocket channel with the isolate deployer.

3.3 Some Key Features

3.3.1 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 node 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 'FileMonitor' [section 3.2.1] monitors changes in the code repository. When a change is detected, the 'FileMonitor' isolate sends a RESTART message along with the target router to notify the controller. The controller takes care of pushing the message to relevant router, and the router takes care of terminating and re-spawning the worker isolates.

This hot deployment capability improves 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 a set of isolates that should be updated is restarted at runtime. This increases overall up-time of the application and keeps other components working even in the time of modifications.

3.3.2 Migration of Isolates and Isolate Systems

Relocation of Worker isolates or an isolate system during runtime i.e. killing a set of Worker isolates or an isolate system at one node and bringing up same set of Worker isolates in another node is the migration of isolates or isolate system. The concept of hot deployment and migration brings enormous possibilities in a distributed system. Some of them are:

- Migration of actors/isolates allows an application to scale in an easy way. With this capability, it is also possible for an application to be brought 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 worker isolates during runtime can make the application survive the hardware failure.

3.3.3 Remote Isolates

The current isolate implementation in Dart[4] cannot communicate with other isolates over a network. The Worker isolates in this framework have an ability to communicate with the isolates that may be running in a remote node. So, there can be isolates running in any node. The communication underneath is taken care of by the framework so the implementer of this framework does not have to worry if an isolate is remotely spawned or locally spawned.

Two isolates, although, running in two different virtual machines, can still belong to a same logical isolate system.

3.4 Typical Message Flow in the System

The framework is based on 'fire-and-forget' principle of message sending. The Figure 3.4 shows a simple message flow while enqueuing a message and the Figure 3.4 shows the process of dequeuing a message. A message is serialized to JSON string before sending via a SendPort of an isolate and deserialized after receiving from ReceivePort.

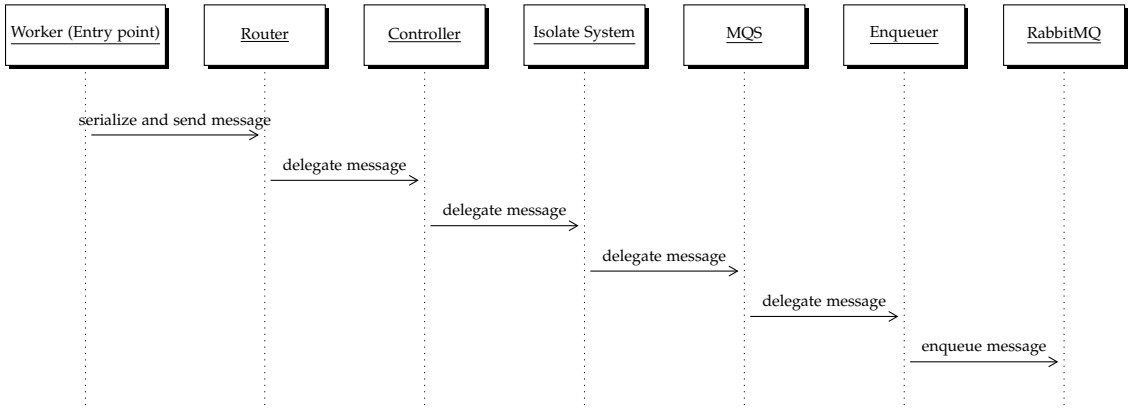

Figure 3.4: Enqueuing a message

[4]Dart version 1.7.2

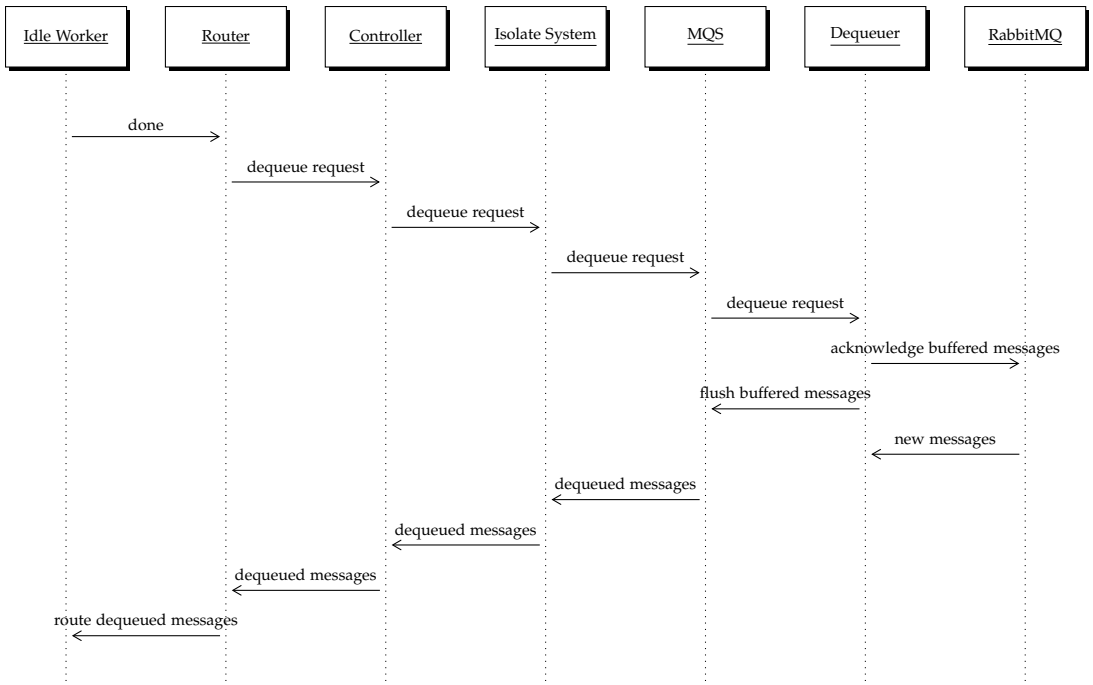

Figure 3.5: Dequeuing a message

3.4.1 Sample message formats

The message sent through different components while enqueuing

Original Message:

```
"Test"
```

Worker:

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

Router:

```
\{senderType: senderType.router, id: sampleSystem/producer, action:
    action.send, payload: \{sender: sampleSystem/producer, to:
    sampleSystem/consumer, message: Test, replyTo: null\}\}
```

Controller:

```
\{senderType: senderType.controller, id: sampleSystem/producer,
    action: action.send, payload: \{sender: sampleSystem/producer,
    to: sampleSystem/consumer, message: Test, replyTo: null\}\}
```

Top level isolate:

```
{targetQueue: sampleSystem.consumer, action: action.enqueue,
    payload: {sender: sampleSystem/producer, message: Test, replyTo:
    null}}
```

Message Queuing System:

```
{topic: sampleSystem.consumer, action: action.enqueue, payload: {
    sender: sampleSystem/producer, message: Test, replyTo: null}}
```

Enqueuer:

```
{sender: sampleSystem/producer, message: Test, replyTo: null}
```

Sample format of a message sent at different components while dequeuing

Dequeuer:

```
{"sender":"mysystem/producer","message":"Test","replyTo":null}
```

Message Queuing System:

```
{senderType: senderType.dequeuer, topic: mysystem.consumer, payload:
    {sender: mysystem/producer, message: Test, replyTo: null}
```

Top level isolate of isolate system:

```
{senderType: senderType.isolate_system, id: mysystem, action:
    action.none, payload: {to: mysystem/consumer, message: {sender:
    mysystem/producer, message: Test, replyTo: null}}}
```

Controller:

```
{senderType: senderType.controller, id: controller, action:
    action.none, payload: {to: mysystem/consumer, message: {sender:
    mysystem/producer, message: Test, replyTo: null}}}
```

Router:

```
{senderType: senderType.router, id: mysystem/consumer, action:
    action.none, payload: {sender: mysystem/producer, message: Test,
    replyTo: null}}
```

Worker:

```
"Test"
```

3.4.2 Some Implementation Overview

Some insight about the implementation of selected functions of the framework:

How 'send' works?

When a Worker Isolate sends a message using 'send' function of the Worker class, the message is encapsulated with further information about the sender and the receiver are added to the message. The encapsulated message is forwarded to the spawner isolate, which in this case, is the router. The router agains forwards it to the controller which again forwards to the top level isolate. Then the top level isolate adds another level of encapsulation and headers 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 abstract 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 a Worker 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?

This is a special control message sent to the isolates as well as isolate system to shutdown themselves. If a KILL message is sent to a Worker isolate, the message is enqueued to the end of the Worker isolate and no futher messages after KILL a message is sent to that isolate by the router. The router then buffers the messages until the isolates are restarted and starts sending them again once the Worker isolates are spawned. Once the Worker isolate finishes processing the already queued messages and encounters the KILL message, the isolate closes its ReceivePort[5] and sits idle. After sometime it gets 'Garbage Collected' and is cleaned up. But, sometimes the garbage collector cannot clean up the isolate and the memory leak occurs. Thus, as a workaround for this, a custom Exception is thrown deliberately by the isolate to terminate itself. The exception is thrown only after it closes all the ports. This way it is sure to get shutdown forcefully and the memory it kept occupying gets freed.

The 'beforeKill' method can be overridden to perform other operations before the isolate throws custom Exception for shutting itself down.

How RESTART works?

Restarting an isolate is basically a combined process of killing an isolate and spawning it up again. However, during the restart, after issuing the KILL message, the messages may keep coming from the controller to the router. These messages are buffered in the router itself. The buffered messages are flushed and sent out once the Worker isolates are spawned. For instance, when the 'Hot Deployment'**??** feature is enabled, if the source code of the isolate is modified and saved, the each of the isolates that the router has spawned gets restarted. During which the messages that arrive after RESTART message are buffered in the router.

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

How shutting down an isolate system works?

An isolate system that is running in a node can be shutdown via the Web Interface or via POST request to the registry. When a request to shutdown an isolate system is sent, the isolate system closes all the ports including the isolate ports as well as the WebSocket connection with Message Queuing System. After that the forceful shutdown is carried out by throwing out a custom Exception. This is a work-around to free up the memory consumed after it is shutdown, because the feature to immediately terminate an isolate is yet to be implemented in dart.

3.4.3 Clustering

Clustering can be achieved in the framework in several levels.

- By deploying worker isolates in several remote nodes. i.e. taking advantage of the concept of 'Remote Isolates' [subsection 3.3.3].
- By deploying replicas of an isolate system in different nodes. An isolate system with same name can exist in another node even though they connect to the same MQS.
- The Message Queuing System itself, can also be replicated where replicas of isolate system may connect to different instance MQS.
- Since, several instances of RabbitMQ [subsection 2.7.2] can form a logical group, sharing common configuration, properties, users, queues etc., a cluster of message broker system can be formed. Which allows, Message Queuing Systems to connect to the different member of a cluster.

3.5 Dart Libraries Used in Construction

Table 3.3: List of libraries directly used by the framework

Library	URL
path	`https://pub.dartlang.org/packages/path`
uuid	`https://pub.dartlang.org/packages/uuid`
crypto	`https://pub.dartlang.org/packages/crypto`
stomp	`https://pub.dartlang.org/packages/stomp`

3.6 A Sample Implementation of Worker Using The Framework

```
import 'dart:isolate';
import 'package:isolatesystem/worker/Worker.dart';

main(List<String> args, SendPort sendPort) {
  Consumer printerIsolate = new Consumer(args, sendPort);
}

class Consumer extends Worker {
  Consumer(List<String> args, SendPort sendPort) : super(args, sendPort);

  @override
  onReceive(message) {
    switch(message['action']) {
      case "print":
        print("message['content']");
        break;
      case "send_back":
        reply(message['content']);
        break;
    }
    done();
  }
}
```

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 Future Directions

Some notes for future directions: * If a code upgrade fails, the isolate should be reverted to its previous state. * Lightweight isolates -> future dart impl? * Detect all sort of exceptions in child isolate -> which would make it possible to implement supervisors and let it crash philosophy. -> future dart impl?, should be available in near future in dart. *

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.

[Arm10] J. Armstrong. "Erlang." In: *Commun. ACM* 53.9 (Sept. 2010), pp. 68–75. ISSN: 0001-0782. DOI: 10.1145/1810891.1810910.

[Cel11] J. Celko. *Joe Celko's SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 2011. ISBN: 978-0-12-382022-8.

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

[Erb12] B. Erb. "Concurrent Programming for Scalable Web Architectures." Diploma Thesis. Institute of Distributed Systems, Ulm University, Apr. 2012.

[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.

[HT] P. Haller and S. Tu. *Scala Actors API*. http://docs.scala-lang.org/overviews/core/actors.html. Last Accessed: 2014-11-30.

[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.

[Nin] C. Ninners. *The Actor Model (everything you wanted to know, but were afraid to ask)*.

[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] U. o. B. Software Languages Lab. *Concurrent Programming with Actors*. http://soft.vub.ac.be/amop/at/tutorial/actors. Last Accessed: 2014-11-30.

[Sofb] 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.

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

[Sofd] 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.