Fabric

0.2.2

Generated by Doxygen 1.8.7

Tue Jul 8 2014 15:57:07

Contents

1 Introduction

Fabric is a language and runtime system that supports secure federated distributed computing. This manual briefly describes the contents of the Fabric distribution package, and should get you started with running Fabric. For more information, see our paper "Fabric: A Platform for Secure Distributed Computation and Storage" published in SOSP 2009 [4].

An HTML version of this manual is available.

More information about Fabric, including the latest release, can be found at the Fabric web site.

User support and feedback

If you use Fabric, we'd appreciate your letting us know. We welcome comments, bug reports, and discussion about Fabric on the Fabric users mailing list. This is a low-traffic mailing list, to which we will also post notifications of new releases of Fabric and other related announcements.

Package contents

The Fabric distribution package contains the following directories:

- `bin`: Scripts for invoking the compiler, runtime, and for configuration tasks. To get usage information for a script, run it with the `--help` option.
- `doc`: The Fabric manual, internal API documentation, and licenses.
  - `doc/api`: The Fabric internal API documentation.
  - `doc/licenses`: Licensing information for Fabric and its packaged dependencies.
  - `doc/manual`: The Fabric user manual.
- `examples`: Example Fabric programs.
- `src`: Source code for the Fabric compiler and runtime, and some libraries built using Fabric.
  - `src/compiler`: The Fabric and FabIL compilers (FabIL is the intermediate language that Fabric compiles into).
  - `src/system`: The parts of the Fabric runtime system that are implemented in Java.
  - `src/runtime`: The parts of the Fabric runtime system that are implemented in FabIL.
  - `src/lib`: Libraries that are built using Fabric, including the Fabric port of the Servlets with Information Flow (SIF) library [3].
- `etc`: Configuration files for Fabric.
- `lib`: Library dependencies for Fabric.
- `tools`: A browser to inspect the contents of a store. A useful aid in the development of Fabric programs.

2 Setting up Fabric

Requirements

This Fabric distribution builds on Linux and OS X. Fabric 0.2.2 builds against Jif 3.4.2 and Polyglot 2.6.1. Both are included in the distribution. JDK 7 or later is required; we have tested with the Oracle and the OpenJDK implementations. OpenSSL is used to sign certificates. Fabric is compiled with the Apache Ant build tool.

The following command will install the requisite tools on an Ubuntu 12.04 or 14.04 system:

```
$ sudo apt-get install openjdk-7-jdk openssl ant
```

Once the tools are installed, unpack the distribution in a location of your choice and change into the distribution directory:

```
$ tar zxf fabric-0.2.2.tar.gz
$ cd fabric-0.2.2
```

Configuring

Before using Fabric, you must configure the scripts in the `bin` directory. From the top-level directory, run:

```
$ ant bin
```

If the configurator is unable to find the JDK, Polyglot, or Jif, it will prompt you to set the appropriate properties in `config.properties`.

Building

This step is optional. Fabric comes pre-compiled as Jar files in the `lib` directory. However, if you wish to rebuild Fabric, run `ant` from the top-level directory:

```
$ ant
```

For other useful build targets, run `ant -p` from the top-level directory.

3 The Fabric language

The Fabric programming language is an extension of the Jif programming language [5], which is in turn a version of Java extended with security labels that govern the confidentiality and integrity of information used in the program, and ensure that information flows in the programs respect those security policies. Therefore, a good place to start is with the Jif manual.

Fabric extends Jif with several additional features:

- using and creating persistent objects on remote stores
- nested transactions
- remote method calls
- access labels
- provider labels
- codebases

These features are summarized below, but more information can be found in two papers about Fabric [4], [1].

Persistent objects

Fabric objects are, in general, persistent. Further, they may be stored persistently at a remote node (a storage node, or store). Applications that need persistent storage do not need a database to back them; they can record information directly in objects. Fabric supports *orthogonal persistence*: programs use objects in the same way regardless of whether they are persistent or not.

Remote persistent objects are created by specifying a store to store them. For example:

```
Store s = FabricWorker.getWorker().getStore("storename");
Object o = new Object@s(args);
```

If a store is not specified, objects are created at the same store as the object `this`. Each worker node also has a local, non-persistent store. A reference to this store can be obtained by calling `FabricWorker.getWorker().get↩LocalStore()`.

Every object in Fabric has an *object label* that specifies the security of the information it contains. The object label is declared by attaching it to a field or fields of the object. (If multiple fields have labels, the object label combines all of them.)

Nested transactions

Fabric computations are organized in *transactions*, which occur, as far as the programmer can tell, atomically and in isolation from the rest of the Fabric system.

Transactions are specified with an atomic block, for example:

```
atomic {
  o1.f();
  o2.g();
}
```

The semantics of the atomic block are that statements inside the atomic block are executed simultaneously and without interference from other concurrent transactions, even those taking place at other network nodes.

Transactions may be nested freely. The results of a nested transaction are only visible to the outer transaction once it successfully commits.

Remote method calls

Unlike in most distributed object systems, computation in Fabric stays on the same network node unless the program explicitly transfers control to another node, using a remote method call.

A remote method call is specified using the syntax `o.m@w(x)`. This is the same syntax as a Java method call, except for the annotation `@w`, which specifies the worker node at which to perform the method call.

```
RemoteWorker w = FabricWorker.getWorker().getWorker("workername");
o.m@w(args);
```

Unlike in many other distributed systems with remote calls, the objects used during the computation of the method `m` need not reside at the remote worker `w`. Also note that transactions can span multiple remote calls; these calls will be executed as a single transaction.

Access labels

When an object is accessed during computation on a worker, but is not yet cached at the worker, the worker must fetch the object data from the node where it is stored. Thus, the contacted node learns that an access to the object has occurred. This side channel is called a *read channel*.

Read channels are controlled by extending Jif with a second label on each object, called the *access label*. It is a confidentiality-only label that bounds what can be learned from the fact that the object has been accessed. The access label ensures that the object is stored on a node that is trusted to learn about all the accesses to it, and it prevents the object from being accessed from a context that is too confidential.

The access label of an object is declared as part of the label of its fields. Given object label `{u}` and access label `{a}`, a label annotation `{u} @ {a}` means that the field, and by extension the object, has the corresponding labels.

For example, the following code declares an object containing public information (in field `data`) that can be accessed without leaking information, according to any principal that trusts node `n` to enforce its confidentiality:

```
class Public {
    int {} @ {⊤→n} data;
}
```

In this example, the object label is `{}` (public and untrusted), and the access label is `{⊤→n}` (readable by principal `n`).

If the access label is omitted from a field, its access label defaults to the label `{this.store→}`. For any object `o`, the pseudo-field `o.store` represents the node on which `o` is stored.

Access labels for classes and interfaces can also be specified independently of fields:

```
interface I {
    @ {⊤→n}
}
```

Provider labels

Remote method calls make it possible to invoke a method on a remote node even when that node has not previously seen the class of the object receiving the call, or its code. To make this possible, Fabric code is stored in *class objects*, which are also persistent objects in Fabric. We refer to the act of adding a class object to Fabric as *publishing* that class.

All code has an information-flow label called the *provider label*, which bounds who can have influenced the code. In fact, this label is precisely the object label of the class object.

Inside Fabric code, the provider label can be named explicitly as `provider`. Before loading code from a class object, a Fabric node checks the information flows within the code, using the provider label to implicitly keep track of the influence that the code publisher has on computations performed by the code.

Codebases

Unlike Java classes, Fabric class objects are accompanied by linkage specifications called *codebases*. There is no global mapping in Fabric from class names to class objects. Instead, each code publisher can choose their own mapping. Fabric helps to make sure that published code uses these namespaces consistently. Thus, codebases support *decentralized namespaces*; a class's own codebase defines the resolution of its dependencies. Linkage of a component's dependencies is fixed at publication, so nodes that download and compile mobile code independently can securely interact with each other.

Codebases are normally not visible in Fabric programs. However, to support evolution of running Fabric systems, it may be necessary to use two classes with the same fully qualified Java name within the same program. This is supported by the use of *explicit codebases*.

For example, to specify that the name `pkg.A` should be resolved through a different codebase than the default one being used in the current code, we might declare the existence of a separate codebase `cb1`:

```
package pkg;
codebase cb1;

class B extends C {
  void m(cb1.pkg.A a) {
    ...
  }
}
```

The fully qualified name `pkg.A` is resolved to a class object through a different class name than the current class, `pkg.B`. The binding between the name `cb1` and the actual Fabric codebase object is done at the time of publication.

4 Compiling and publishing Fabric programs

Compiling

To compile a Fabric program `MyClass.fab`, run the Fabric compiler, `fabc`:

```
$ fabc MyClass.fab
```

The `fabc` compiler has many options similar to the `javac` compiler, including the `-d` option to output classes in a different directory, the `-classpath` option to specify the classpath, and the `-sourcepath` option to specify the source path. For a complete list of options, run:

```
$ fabc -help
```

Publishing

The Fabric compiler can publish code to a Fabric store, making the code available for download and use by Fabric workers. Publishing code requires a running store and a configured worker. (See Starting Fabric nodes.) To publish, the Fabric compiler needs a few additional parameters:

- the name of the store that will host the published code,
- the name of the worker to use for publishing, and
- a file to which to write codebase information.

The following command will use the worker `MyWorker` to publish `MyClass.fab` to the store `MyStore`, outputting the URL of the resulting codebase to the file `codebase.url`:

```
$ fabc -deststore MyStore -worker MyWorker -publish-only \
    -codebase-output-file codebase.url MyClass.fab
```

The Fabric compiler can also compile against published code by specifying the codebase file on the classpath:

```
$ fabc -worker MyWorker -classpath @codebase.url MyClass2.fab
```

Code dependent on published code can similarly be published:

```
$ fabc -deststore MyStore -worker MyWorker -publish-only \
    -codebase-output-file codebase2.url \
    -classpath @codebase.url MyClass2.fab
```

5 The Fabric intermediate language

Fabric's intermediate language, FabIL, is a Java dialect for writing programs that can run on the Fabric runtime system. As its name suggests, it is internally used by the Fabric compiler as an intermediate language. Therefore, FabIL is lower-level than the full Fabric language and does not enforce information-flow security. FabIL is exposed as a separate language with its own compiler, `filc`. Portions of the Fabric runtime is written in FabIL.

Like Fabric, FabIL supports using and creating persistent objects on remote stores, nested transactions, and remote method calls. Mobile code is not supported, however.

There are two primary differences between Fabric and FabIL. First, whereas Fabric has types labelled with policies for information-flow security, FabIL does not. In this respect, FabIL is more closely related to Java than it is to Fabric. The second difference is in how objects are constructed. Although FabIL programs do not enforce information-flow security, they do create full-fledged Fabric objects, which have labels. To support this, FabIL has:

- an API for creating label objects,
- syntax for specifying labels on arrays, and
- an object-construction convention that must be followed.

Creating label objects

While Fabric has built-in syntax for labels, FabIL does not. Instead, labels in FabIL are constructed explicitly through the API provided by the library class `fabric.lang.security.LabelUtil`, which exposes several static methods for constructing label objects.

Pre-built labels and policies

Method signature	Description
`Label noComponents()`	Returns the label {⊥→⊥ ; ⊥←⊥}.
`ConfPolicy bottomConf()`	Returns the confidentiality policy {⊥→⊥}.
`ConfPolicy topConf()`	Returns the confidentiality policy {⊤→⊤}.
`IntegPolicy bottomInteg()`	Returns the integrity policy {⊤←⊤}.
`IntegPolicy topInteg()`	Returns the integrity policy {⊥←⊥}.

Creating policies

Method signature	Description
`ConfPolicy readerPolicy(Store store, Principal owner, Principal reader)`	Creates and returns the confidentiality policy {owner→reader}, allocated on `store`.
`IntegPolicy writerPolicy(Store store, Principal owner, Principal writer)`	Creates and returns the integrity policy {owner←writer}, allocated on `store`.
`ConfPolicy join(Store store, ConfPolicy p1, ConfPolicy p2)`	Creates and returns the confidentiality policy `p1` ⊔ `p2`, allocated on `store`.
`ConfPolicy meet(Store store, ConfPolicy p1, ConfPolicy p2)`	Creates and returns the confidentiality policy `p1` ⊓ `p2`, allocated on `store`.
`IntegPolicy join(Store store, IntegPolicy p1, IntegPolicy p2)`	Creates and returns the integrity policy `p1` ⊔ `p2`, allocated on `store`.
`IntegPolicy meet(Store store, IntegPolicy p1, IntegPolicy p2)`	Creates and returns the integrity policy `p1` ⊓ `p2`, allocated on `store`.

Creating labels

Method signature	Description
`Label readerPolicyLabel(Store store, Principal owner, Principal reader)`	Creates and returns the label {owner→reader ; ⊥←⊥}, allocated on `store`.
`Label writerPolicyLabel(Store store, Principal owner, Principal writer)`	Creates and returns the label {⊥→⊥ ; owner←writer}, allocated on `store`.
`Label toLabel(Store store, ConfPolicy cPolicy, IntegPolicy iPolicy)`	Creates and returns the label {cPolicy ; iPolicy}, allocated on `store`.
`Label join(Store store, Label l1, Label l2)`	Creates and returns the label `l1` ⊔ `l2`, allocated on `store`.
`Label meet(Store store, Label l1, Label l2)`	Creates and returns the label `l1` ⊓ `l2`, allocated on `store`.

Constructing arrays

In FabIL, arrays are created by specifying their label and a store to store them. For example:

```
Store store = Worker.getWorker().getStore("storename");
Label lbl = LabelUtil.noComponents();
int[] array = new int[5] ~lbl @store;
```

The ∼`lbl` component of the `new` expression gives the name of a variable containing the label for the array. If the label is not specified, the array is created with the same label as the object `this`. Like in Fabric, the `@store` component gives the store on which to create the array; if it is omitted, the array is created on the same store as `this`.

Object-construction convention

Fabric ensures that final fields really are final: it should not be possible to observe final fields of an object before they have been initialized. This property, inherited from Jif, is important because final fields of type `label` or `principal` may be used in security policies. It also has implications for how constructors are written in Fabric and, consequentially, FabIL.

To ensure that final fields really are final, Fabric constructors must initialize all final fields before calling the superclass constructor. For example, a class in Fabric might look like the following:

```
package geometry;

class NCPoint extends Point {
  final String{} name;
  Colour{} c;

  NCPoint(String name, int x, int y, Colour c) {
    this.name = name; // Initialize all final fields.
    super(x,y);       // Then call super class's constructor.
    this.c = c;
  }
}
```

In Java, no code is allowed to precede the call to the superclass constructor. Therefore, operationally, Fabric separates the *allocation* of objects from their *initialization*. This means that when a Fabric class is translated to FabIL, no explicit constructors are created. Instead, Fabric constructors are translated into *initializer methods*. For example, the Fabric compiler translates the above class into the following FabIL class:

```
package geometry;

class NCPoint extends Point {
  String name; // intended to be final
  Colour c;

  NCPoint geometry$NCPoint$(String name, int x, int y, Colour c) {
    this.name = name;      // initialize "final" fields
    geometry$Point$(x,y);  // call super class's initializer
    this.c = c;            // initialize other fields
    return this;
  }

  // Specifies the object's label and access policy. Called by the
  // initializer for fabric.lang.Object.
  public Object $initLabels() {
    this.$updateLabel = LabelUtil.noComponents();
    this.$accessPolicy = LabelUtil.bottomConf();
    return this;
  }
}
```

To be compatible with Fabric, programs written in FabIL should follow this same convention. There are a few things to note:

- Although the field `name` is intended to be final, its `final` flag is removed.
- No constructors are declared. Instead, the class has the initializer method `geometry$NCPoint$`, which implements the constructor's functionality. The call to the superclass's constructor is turned into a call to the appropriate initializer method in the superclass.
- The name of the initializer method is derived from the fully qualified name of the class: dots are replaced with dollars, and an extra dollar is appended at the end.
- The method `$initLabels()` is declared for specifying the object's label and access policy. This can depend on the object's "final" fields, because this method is called by the initializer for fabric.lang.Object, after all of the "final" fields are initialized.
- Initializer methods and `$initLabels()` return `this`.

Object construction is done in two stages. First, the implicit default constructor is called to allocate the object. Then, the desired initializer method is called to initialize the object:

```
new NCPoint().geometry$NCPoint$("origin", 0, 0, Colour.BLACK)
```

The FabIL compiler does not enforce any aspect of this convention. It is up to the programmer to ensure that this convention is followed.

6 The Fabric runtime system

There are three types of nodes in the Fabric design: stores, workers, and dissemination nodes. In the current implementation, there are no separate dissemination nodes; rather each worker and store participates as a peer in the dissemination network.

6.1 Creating Fabric nodes

Fabric nodes run relative to an *application home* directory, which contains the nodes' configuration and state. As a transitional naming mechanism, the application home can also contain naming information, specifying how to contact other Fabric nodes.

This guide demonstrates how to create and configure a Fabric node. As a running example, the node we create will be named `valinor` and will use `/opt/fabric` as the application home. The commands given are relative to the Fabric installation directory.

The name of the node can be any legal DNS hostname. This should ideally match the host machine's DNS name. However, with the transitional naming mechanism, a node's name need not have a DNS entry, nor is it required to match the host machine's DNS name.

Create a certificate authority

Fabric uses SSL to secure communication between nodes, so each node must have an X.509 certificate signed by a certificate authority (CA). While commercial CAs may be used, the Fabric distribution uses OpenSSL to provide a custom CA for convenience.

This Fabric distribution comes with a CA whose key pair and certificate have been pre-generated. Using this pre-generated CA avoids having to distribute the CA certificate across separate Fabric installations, but is insecure. For security, we recommend creating your own CA:

```
$ bin/make-ca
```

By default, this creates a CA in `etc/ca` of the Fabric installation, overwriting any previously generated CA, and saves the CA certificate in `etc/ca/ca.crt`.

Generate a key pair and certificate

Each Fabric node needs a key pair and a signed X.509 certificate. These can be created in one of two ways. The first method is provided for convenience. The second method is suggested for better flexibility.

Method 1: Using `make-node`

This method creates the key pair and certificate in a single step, using the CA in the Fabric installation to sign the certificate.

```
$ bin/make-node --app-home /opt/fabric --name valinor
```

This creates a Java keystore `/opt/fabric/etc/keys/valinor.keystore` containing the node's private key and certificate.

Before importing the certificate into the node's keystore, the CA certificate will be displayed, and you will be asked whether it should be trusted. You can use the `--trust-ca-cert` option to automatically trust the CA certificate without prompting:

```
$ bin/make-node --app-home /opt/fabric --name valinor --trust-ca-cert
```

Method 2: Using `genkey`

This method creates the key pair and certificate in multiple steps, and is useful if you are using a commercial CA, or if the CA is on a separate Fabric installation.

Generate the key pair and the certificate-signing request (CSR).

```
$ bin/genkey --app-home /opt/fabric --name valinor
```

This creates a Java keystore `/opt/fabric/etc/keys/valinor.keystore` containing the node's private key, and a CSR in `/opt/fabric/etc/csr/valinor.csr`.

Have the CSR signed by a CA. To sign using Fabric's CA facility:

```
$ bin/ca-sign /opt/fabric/etc/csr/valinor.csr /tmp/valinor.crt
```

This creates a signed certificate in `/tmp/valinor.crt`.

Once you have a signed certificate, import it and the CA's certificate into the node's keystore:

```
$ bin/import-cert --keystore /opt/fabric/etc/keys/valinor.keystore \
    --ca etc/ca/ca.crt /tmp/valinor.crt
```

Before importing the certificates, the CA certificate will be displayed, and you will be asked whether it should be trusted. You can use the `--trust-ca-cert` option to automatically trust the CA certificate without prompting:

```
$ bin/import-cert --keystore /opt/fabric/etc/keys/valinor.keystore \
    --ca etc/ca/ca.crt /tmp/valinor.crt --trust-ca-cert
```

Import other CA certificates

If this node will be communicating with nodes whose certificates are signed by other CAs, you will also need to import the certificates of those CAs.

```
$ bin/add-trusted-ca --keystore /opt/fabric/etc/keys/valinor.keystore \
    other-ca.crt
```

Before importing the certificate into the node's keystore, the CA certificate will be displayed, and you will be asked whether it should be trusted. You can use the `--no-prompt` option to automatically trust the CA certificate without prompting:

```
$ bin/add-trusted-ca --no-prompt \
    --keystore /opt/fabric/etc/keys/valinor.keystore \
    other-ca.crt
```

Configure the node

Our node `valinor` reads its configuration information from the files `etc/config.properties` and `etc/config/valinor.properties` in the application home. The first file, `config.properties`, specifies configuration values that are common to all nodes that share the application home. These values can be overridden in the node-specific file `valinor.properties`.

The Fabric distribution offers `etc/config/EXAMPLE.properties.in` as a template for a configuration file. Copy this to `/opt/fabric/etc/config/valinor.properties` and edit the file:

```
$ cp etc/config/EXAMPLE.properties.in /opt/fabric/etc/config/valinor.properties
$ vim /opt/fabric/etc/config/valinor.properties
```

Fabric nodes are configured using several parameters.

Configuration parameters common to all nodes

Required

- `fabric.node.password` specifies the password for the node's keystore file.

Optional

- `fabric.node.keystore` specifies the name of the node's keystore file. By default, this is `NODE_NAM↩E.keystore`. In our example, this is `valinor.keystore`. This file must be located in the `etc/keys` directory of the application home.
- `fabric.node.hostname` specifies the IP address or the DNS name of the node's host machine. By default, this is the same as the node's name.
- `fabric.node.fetchmanager.class` specifies the class for the dissemination layer implementation. Valid options include `fabric.dissemination.PastryFetchManager` and `fabric.↩dissemination.DummyFetchManager`. By default, `PastryFetchManager` is used.
- `fabric.worker.port` specifies the network port on which the node should listen for remote calls. The default worker port is 3372.
- `fabric.worker.adminPort` specifies the network port on which the worker should listen for administrative connections. The default administrative port is 3572.

Configuration parameters for worker nodes

Required

- `fabric.worker.homeStore` specifies the name of the store that will hold the worker's principal object.

Configuration parameters for storage nodes

Optional

- `fabric.store.port` it specifies the network port on which the store should listen for connections. The default store port is 3472.
- `fabric.store.db.class` specifies the class for the store's back-end database. Valid options include `fabric.store.db.BdbDB` and `fabric.store.db.MemoryDB`. The default is `fabric.store.↩db.BdbDB`.

Configure name resolution

We must specify how other nodes can contact the node we just created. Ideally, this is done by adding a DNS entry with with an `A` (or `AAAA`) record containing the host machine's IP address and a `TXT` record specifying the node's network-port configuration. A store listening on port 3472 should have a `TXT` record `"fabric-store: 3472"`. Similarly, a worker listening on port 3372 should have a `TXT` record `"fabric-worker: 3372"`. (Default values

are assumed if these records do not exist.) This use of the `TXT` record prevents the node's network-port configuration from being tied to the node's name (à la http://example.com:8080/) and allows the port configuration to change over time.

However, because this use of DNS is non-standard, we provide a transitional mechanism for resolving node names. To resolve a node, Fabric first looks in the `etc/config` directory of the application home. If it finds a `.properties` configuration file for the node, then it uses the configuration information found in that file. Otherwise, DNS is used.

When resolving a node, this transitional naming mechanism must be used if (a) the node's name does not match the host machine's DNS name, or (b) the node has a non-default port configuration and no corresponding `TXT` records in DNS.

To contact such a node, `valinor` must have a copy of the node's `.properties` configuration file in the `etc/config` directory of the application home. Similarly, if `valinor` relies on the transitional mechanism to be found, then its `.properties` configuration file must be copied to any node that will contact `valinor`.

6.2 Starting Fabric nodes

Continuing with the example from the previous section, our example node will be named `valinor` and will use `/opt/fabric` as the application home. The commands given are relative to the Fabric installation directory.

Starting a store

To start `valinor` as a store:

```
$ bin/fab-store --app-home /opt/fabric --name valinor
```

While the store is running, you are presented with a shell for the store's colocated worker. You can use this shell to run Fabric applications from within the colocated worker. (See Running Fabric programs.)

The node's class path must include any non-mobile application classes that it will use. This can be specified with the `--jvm-cp` option:

```
$ bin/fab-store --jvm-cp /path/to/classes --app-home /opt/fabric \
    --name valinor
```

Starting a worker

The following command starts `valinor` as a worker that is initially idle, but is available to receive remote calls.

```
$ bin/fab --app-home /opt/fabric --name valinor
```

While the worker is running, you are presented with the worker's shell. You can use this shell to run Fabric applications from within the worker. (See Running Fabric programs.)

The worker's class path must include any non-mobile application classes that it will use. This can be specified with the `--jvm-cp` option:

```
$ bin/fab --jvm-cp /path/to/classes --app-home /opt/fabric \
    --name valinor
```

7 Running Fabric programs

This guide demonstrates how to run a Fabric program. As a running example, we will use a worker named `valinor`, with `/opt/fabric` as the application home. The commands given are relative to the Fabric installation directory.

From the command line

The following command starts the worker, executes the class `hello.Main`, and shuts down the worker. If the worker is already running, then the command attaches to the already-running worker and executes `hello.Main` from within that instance.

```
$ bin/fab --jvm-cp /path/to/classes --app-home /opt/fabric \
    --name valinor hello.Main
```

The `--jvm-cp` option specifies a path containing the (non-mobile) class `hello.Main` and any non-mobile classes it uses.

To execute a mobile class, specify the class's codebase-relative name. For example,

```
$ bin/fab --app-home /opt/fabric --name valinor \
    fab://helloStore/35/hello.Main
```

From the worker shell

If a class name is not given when starting a worker, then the worker starts idle and the user is given a worker shell:

```
$ bin/fab --app-home /opt/fabric --name valinor

Worker started

valinor>
```

Stores also have worker shells for their colocated worker:

```
$ bin/fab-store --app-home /opt/fabric --name valinor

Worker started
Store started

valinor>
```

To execute a non-mobile class from within the worker, simply enter the class's name into the shell:

```
valinor> hello.Main
Hello, world!

valinor>
```

The worker must have the requisite non-mobile classes on its class path, as specified by the `--jvm-cp` command-line option when starting the worker.

To execute a mobile class, enter the class's codebase-relative name:

```
valinor> fab://helloStore/35/hello.Main
Hello, world!

valinor>
```

8 Example programs

The `examples` directory in the distribution contains several example Fabric programs. Each example includes a separate README describing how to build and run the example. The Fabric nodes required to run each example come pre-configured. We briefly list the examples here.

- `examples/hello`: Every programmer's favorite program, ported to Fabric. This example creates a persistent object containing the message "hello world" and then outputs that message on the console.
- `examples/sif-hello`: A demonstration of the Fabric port of the Servlets with Information Flow (SIF) library [3]. This example shows how web services can be built on top of Fabric.
- `examples/travel`: A more complete demonstration of Fabric's features. This application involves coordination between an airline, bank, and customer to negotiate the purchase of a ticket. Each principal (airline, bank, and customer) also has a web-based user interface written using the SIF library.
- `examples/auction`: A mobile bidding-agent program. This application demonstrates the mobile-code support of Fabric [1]. It models an auction in which participants submit confidential strategies for bidding and selling.
- `examples/friendmap`: Another mobile-code demonstration. This program models a mash-up of social network and a mapping service to map a friend's confidential location.
- `examples/OO7`: This is an implementation of the OO7 Object Oriented Database Benchmark [2]. It is written using FabIL, the intermediate language for Fabric, and thus does not benefit from the static information-flow checking that the full Fabric language provides.
- `examples/blog`: This is a simple web application implemented in FabIL. It is similar in structure to the Course Management System that we used for evaluating performance of Fabric [4].

9 Version history

Version 0.2.2 (Jul 2014)

- Language support for specifying the access policy for a class or interface independently from fields.
- Support for local deadlock detection.
- SIF has been refactored to better support building non-HTML web apps.
- Class hashing is now less sensitive to the version of Java being used.
- Various bug fixes and performance improvements.
- Updated to Jif 3.4.2 and Polyglot 2.6.1.

Version 0.2.1 (Jun 2013)

Issued fresh node certificates for examples. The old ones had expired.

Version 0.2.0 (Oct 2012)

- Support for mobile code [1]
  - Provider labels
  - Access labels
  - Type fingerprint checking for remote calls and objects
- Support for codebases
- Support for heterogeneous field labels

- Workers have interactive consoles. These can be used to run Fabric programs or invoke the Fabric compiler from within the worker.
- Workers listen for administrative connections. If a second instance of a worker is launched, it will attach to the first instance on the administrative port and execute commands within the first instance. This is a convenience feature to enable scripting of worker commands.
- Added `DummyFetchManager`, which implements a degenerate dissemination layer.
- Added `bin/dump-bdb`, which dumps the contents of a store's BDB-backed object database
- Various performance improvements.

Version 0.1.0 (Sep 2010)

Initial release. [4]

10 Credits

The current Fabric developers are:

- Owen Arden
- Jed Liu
- Tom Magrino

The Fabric project group is supervised by Prof. Andrew C. Myers at the Cornell University Computer Science Department.

Past Fabric developers and contributors are:

- Michael D. George
- Samarth Lad
- Xin Qi
- Robert Soulé
- K. Vikram
- Lucas Waye
- Danfeng Zhang
- Xin Zheng

Nate Nystrom was involved in the early stages of the Fabric project.

Steve Chong provided guidance on extending Jif and provided quick bug fixes to the base Jif compiler.

Thanks also to Matthew Loring and Isaac Sheff for submitting bug reports.

We thank Hakim Weatherspoon for the use of the Fractus cloud infrastructure for running the performance tests that helped improve this release.

Sponsors

The development of the Fabric project has been supported by a number of funding sources, including:

- National Science Foundation grants 0430161, 0541217, 0627649, and 0964409
- Microsoft Corporation
- AF-TRUST, which receives support from the DAF Air Force Office of Scientific Research (FA9550-06-1-0244) and the NSF (0424422)
- U.S. Air Force Research Laboratories NICECAP grant FA8750-08-2-0079
- Office of Naval Research grants N00014-09-1-0652 and N00014-13-1-0089
- MURI grant FA9550-12-1-0400, administered by the U.S. Air Force
- an NDSEG fellowship

Fractus is provided by an AFOSR DURIP award, grant FA2386-12-1-3008.

This work does not necessarily represent the opinions, expressed or implied, of any of these sponsors.

References

[1] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE 2012 Symposium on Security and Privacy*, pages 191–205, San Francisco, CA, USA, May 2012. Software release at <http://www.cs.cornell.edu/projects/fabric/>. 3, 14

[2] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proc. ACM SIGMOD 1993 International Conference on Management of Data*, pages 12–21, Washington, DC, USA, May 1993. 14

[3] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security Symposium*, pages 1–16, Boston, MA, USA, August 2007. See <http://www.cs.cornell.edu/jif/sif/>. 2, 14

[4] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symposium on Operating System Principles (SOSP)*, pages 321–334, Big Sky, MT, USA, October 2009. Software release at <http://www.cs.cornell.edu/projects/fabric/>. 1, 3, 14, 15

[5] Andrew C. Myers. JFlow: Practical mostly-static information flow control. pages 228–241, San Antonio, TX, USA, January 1999. Software release at <http://www.cs.cornell.edu/jif/>. 3