

JThings - a parallel/distributed middleware tailored to Research

Luc Hogie

Cnrs/Inria/Université Côte d'Azur

December 4, 2020

What is this all about?

This is about JThings, a new middleware for parallel/distributed computing. JThings is being designed to:

- meet the needs of our on-going/future R&D projects
 - investigate IOT/decentralized algorithms (Map/Liquori)
 - provide scientific tools to our applications (Gemoc)
 - compute very large graphs (Coati)
- besides, be helpful to Researchers in Computer Science
- avoid the pitfalls of existing tools

<https://github.com/lhogie/pafadipo>

Latest product of a software suite?

JThing benefits from the experience we gained in past devs:

BigGrph set of modules dedicated to distributed graph computing. Supported by Inria. Now discontinued.

JMR disk-based distributed batch processor.

JMaxGraph multi-thread graph library for processing large graphs.

MPI4Lectures multi-thread message-passing library geared towards teaching at University of Luxembourg

Also, JThings learns a lot from ProActive, a distributed implementation of the Fractal specification developed by another team in our lab.

In a few words...

JThings allows the user to execute software components on networked computers. These components will interact with each other by sending/receives messages, thereby enabling the execution of a distributed application made of services within the components.

Components: what they *are*

There is no global consensus on what a component is. In JThings, a component is an object augmented with the following features:

- it organizes, along with other components, in a multi-graph
- it is able send/receive *messages* to/from other components, using a variety of transport protocols
- it forwards incoming messages
- it exposes *services* which extend its functionality

Components: what they *are used for*

JThings components can be used to several purposes:

- managing computational resources in a cluster
- simulating complex systems by representing domain-specific entities and their interactions
- providing services in a distributed application

Built-in services

Ranging from simplest to more sophisticated:

- `exit` shuts down the entire distributed system

- `ping pong` emulates the ubiquitous ping command

- `error log` archives and disseminates internal errors

- `service lifecycle` starts/stops services remotely

- `routing` guides messages traveling across the graph

Built-in services

Ranging from simplest to more sophisticated:

bencher provides performance information about the host computer

maps constructs a graph representing the topology of the network

deployment enables new components to be started

time series database stores and serves numerical time-based information on user-defined metrics

publish-subscribe notifies subscriber components of new publications on particular topics

Service

A service exposes to other components functionality for a particular concern.

- it proposes a high-level API for communication
- it defines a set of *actions*

Actions are what perform user-defined computations

- has a name to executable code
- triggered by *messages* which carries parameters
- by default, actions are all executed in *parallel*
- at runtime, they can send messages (temporary results, current state, progress information, etc)

Just like components and services, actions can be created/deleted at runtime.

Message

- carries a content (that can be anything)
- to a set of components defines in a "to:" address
- has an optional "reply-to:" (that is automatically fed in the case of a synchronous emission)

An address consists of:

- an optional set of recipients components (unicast if $|R| = 1$, multicast if $|R| > 1$, broadcast if $R = null$)
- a recipient service that is expected on all recipients components
- a recipient action

Communication

- message emission is asynchronous
- no guaranty of delivery
- if a message has "reply-to" information, the sender obtains a queue that will store response messages.

Queues enable:

- asynchronous on-the-fly processing of incoming messages
- synchronous collect and classification

Communication protocols

JThings currently support the following transport protocols:

- TCP: ensures reception
- UDP: is quick
- IPC: connect components in child/parent processes
 - locally, mostly useful for tests
 - remotely through SSH, across NATs and firewalls
- intra-process: method calls enables large local simulations

Deployment of components

New components can be deployed anywhere a SSH connection is possible (and rsync is available).

- ① determines shared file systems among computers
- ② update binaries
 - ① incremental update of Java bytecode
 - ② installation the right JVM if necessary
- ③ executes a JVM and starts JThing in it

The parent component initially communicate with the new component through the standard I/O streams of the SSH local process. If the new component is declared to be autonomous, it remains alive even if the I/O streams get closed.

Interoperable with external tools using REST

- the REST service launches a HTTP server
- which serve JSON documents provided by specific REST actions in services
- the REST interface gives access to the component system as a whole, regardless of which component exposes it
- executing a REST action is done via the URL:

`http://host:port/component/service/action/parms1,parm2,...,parm3`

Conclusion: most notable features

- ➊ multihop mobile overlay network of components
- ➋ deployment of new remote/local component
- ➌ synchronous and asynchronous communications
- ➍ support for unicast/multicast/broadcast
- ➎ reactive message programming and stream processing
- ➏ massively parallel computations
- ➐ multi-protocol, REST/JSON interface
- ➑ many base services for platform management and demo

Adding/deleting actions

Actions can be declared by:

calling `Service.registerNewAction()` allows adding/deleting new actions at runtime

adding annotated methods improves readability by isolating the code, allowing the specification of a return and parameters type. Such an action can be unregistered but its implementation code remains, and can still be invoked from within its class.

Sending a message

Actions can be declared by:

```
To to = new To(  
  MyResultType r = (MyResultType) send(content ,  
                                         Set.of(myRemoteComponent) ,  
                                         myService , myAction)  
    .setTimeout(5)  
    .collect()  
    .ensureResults(1)  
    .first().content ;
```

Conclusion