RELEASE

SEVENTH FRAMEWORK PROGRAMME

ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D6.3 (WP6): Distributed Erlang Component Ontology

Due date of deliverable: 30th June 2013
Actual submission date: 30th September 2013

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: Erlang Solutions Ltd. Revision: 1.0

Purpose: To define a common ontology and a set of concepts for systems in the Erlang domain. This is to remove ambiguities when referring to an Erlang system, to its components, to the relations between its components and to the operations which can be performed on them.

Results: The main results of the deliverable are as follows:

- the first ontology for Erlang systems.
- a description of how the ontology relates to Wombat and SD-Erlang as well as what the relationship between Wombat and SD-Erlang is.

Conclusion: With this initial DECO (Distributed Erlang Component Ontology) in place we have a foundation for talking with less ambiguity about Erlang systems consisting of heterogeneous Erlang releases.

We have also described the relationship between Wombat, SD-Erlang and DECO. DECO has been quite useful in understanding where the differences and similarities between Wombat and SD-Erlang are, and also how to integrate SD-Erlang into Wombat.

Application outside ESL is still pending and will be started in the near future.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		✳
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Distributed Erlang Component Ontology

Torben Hoffmann <torben.hoffmann@erlang-solutions.com>
Francesco Cesarini <francesco.cesarini@erlang-solutions.com>
Enrique Fernandez <enrique.fernandez@erlang-solutions.com>
Simon Thompson <s.j.thompson@kent.ac.uk>
Natalia Chechina <Natalia.Chechina@glasgow.ac.uk>

Contents

Abstract

This deliverable defines an ontology for Erlang systems that will allow people to talk about how their various components fit together.

It is also shown how the terms from the ontology can be used to describe what Wombat is doing and how SD-Erlang fits in with both of them.

Executive Summary

The objective of D6.3 is to provide an ontology for describing distributed Erlang systems, namely DECO (Distributed Erlang Component Ontology) and show how it can be used to talk about Erlang systems as well as resolve the connection between DECO and SD-Erlang.

DECO provides the vocabulary needed to talk about distributed Erlang systems using well-defined terms, which has hitherto been missing for systems. Inside an Erlang application there has been solid concepts around for decades — e.g., supervisors and generic servers — that software architects of Erlang programs have used to communicate and reason about their programs. The void for systems has been due to a lack of representation of concepts above the application level in Erlang/OTP. DECO aims to fill this void on the conceptual level, whereas the deployment tool Wombat (see Work Package 4 deliverables for details) is the concrete implementation of these concepts.

It is shown how the deployment tool Wombat and SD-Erlang relate to each other and to DECO.

1 Introduction

This deliverable is part of the *Case Studies* work package (WP6) of the RELEASE [pro] project. According to the *Description of Work* of the project, the purpose of this work package is *"to demonstrate and validate the tools and methodologies developed within the context of the project"*.

Until now there has been excellent concepts in place for talking about the internal structure of an Erlang release due to the widespread and well-documented OTP library. However, when the talk turns to Erlang systems there has been no uniform way to talk about how those systems are composed.

In this deliverable we introduce DECO (Distributed Erlang Component Ontology) as an answer to this void.

DECO wasn't part of the original deliverables in RELEASE since we changed the focus from continuous integration to deployment and monitoring. The more we worked on deployment the more it became obvious that we were lacking a well-defined way of talking about Erlang systems and their structure.

The concepts of DECO are gradually introduced and a list of all terms and their definitions is given for reference.

1.1 What DECO is and isn't

Just to be clear about what the scope of DECO a short statement about what it is and what it isn't is required.

DECO is for describing distributed Erlang systems running possibly several different Erlang releases. Hopefully it will become the common vocabulary for talking about distributed Erlang systems.

The terms and concepts of DECO are therefore coined with this purpose in mind and it might not be easy — or even possible — to use DECO to describe general distributed systems.

It would be interesting to contrast DECO with things like the Reference Model for Open Distributed Processing (RM-ODP)[LMTV11], but that is considered outside the scope of this project.

Partner Contributions to WP6.

- University of Glasgow has worked with ESL on resolving the connection between DECO and SD-Erlang which is presented in section 3.
- University of Kent has helped with the clarification of terms in DECO.

2 DECO– Distributed Erlang Component Ontology

When describing and discussing architectures and operations and maintenance issues of saleable Erlang systems, it has become clear that there is a lack of common terminology to describe Erlang systems beyond the level of Erlang/OTP.

Erlang/OTP does a good job of defining the concepts of an Erlang release by means of its nodes, applications, supervision trees and behaviours. However, there is no such vocabulary when it comes to Erlang systems that may comprise a number of different Erlang releases connected in various standard and non-standard ways.

DECO— or Distributed Erlang Component Ontology — is an attempt to define the terms and concepts needed to describe Erlang systems.

In an Erlang system consisting of many nodes there are two main things to be concerned about at the system level:

- which Erlang release is a node running?
- how does communication between any two nodes in the system take place?

In addition to this logical architecture of a system there is also the actual deployment of the software on servers and DECO provides means to describe this as well.

In the rest of this section we will introduce the concepts of DECO gradually by adding more and more detail to an abstract system architecture.

A list of all the concepts of DECO can be found in the appendix A.

2.1 Parts of an Erlang system

2.1.1 System Context

The Erlang systems that we are going to describe normally talk to both clients and external services as shown in Figure 1. There will be one or more (possibly millions) clients accessing the system, which in turn can call upon external services to handle the requests.

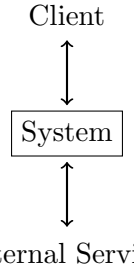


Figure 1: System architecture

2.1.2 Nodes and Clusters

This arch-typical context has the implication that if we open up the system component, it consists of a number of *clusters*, which in turn consists of a number of *nodes* as show in Figure 2, where Cluster 1 has been opened up.

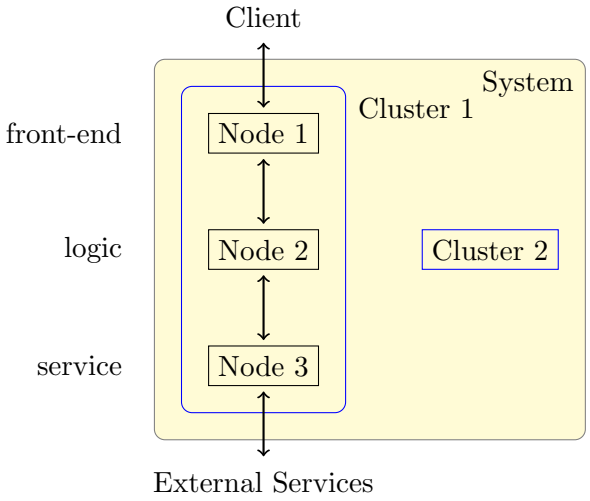


Figure 2: Internal system architecture

A node is an instance of an Erlang run-time system running on some provisioned machine. The connections in Figure 2 between the nodes signify that there is communication between the nodes in question. Exactly how nodes communicate differs depending on the system — some nodes will be connected using distributed Erlang, some using UDP, others with RabbitMQ and so on.

Since the context of most Erlang systems is as in Figure 1 the internal architecture in Figure 2 with three layers is very common.

The top nodes in a cluster are often called *front-end nodes* — one of a number of different *semantic node types* which is used to classify the nodes in a cluster. The level below are called *logic nodes* and they implement the business logic of the system. In order to do so they often require to use a number of different *services* and for that purpose a number of *service nodes* may be part of a cluster. And some of the service nodes may need to contact external services to do their job.

Note: the semantic node types are merely a term used to help talking about the overall responsibility of a node.

Note: a node may have several semantic node types, e.g, both be a front-end node and a logic node. This normally happens if the system is not that big. In that case there will be processes on the node that handles the front-end and processes that deals with the logic.

This layering is similar to what one finds in many single node Erlang applications, where there are processes responsible for dealing with the incoming requests (typically decoding the messages) before sending them on to some business logic process that may call on other processes to fulfil its task. The only real difference is that for a multi-node system these responsibilities are distributed across nodes instead of across processes.

Regardless of if one is using the three-layer architecture or some other architecture communication between nodes should respect the layering in the architecture, i.e., one should not have front-end nodes talking to service nodes. It is not illegal to do so, but it may very well lead to a spaghetti architecture or at least some confusion for those trying to understand a system using the architecture as a starting point.

The different semantic node types are represented by instances of one or more *node types*. A node

type is a descriptive term used to talk about nodes running the same Erlang release; possibly in different versions. So if you have a release R_a in version 1.0 running on one node and in version 1.1 running on another node those two nodes would be of the same node type (R_a) and we would talk about the R_a node type.

The type of nodes in a cluster and how they are connected are described in the *cluster blueprint*, which also contains all the information about how a cluster looks like. In the following the remaining parts of the cluster blueprint will be explained.

Scaling with Nodes When the load on a system becomes bigger it will no longer be enough to have just one node on each layer. E.g., for every 3 front-end nodes there needs to be one logic node and for every 2 logic nodes there needs to be 1 service node of type R_x and 2 of service node R_y.

This information is needed to ensure that the system can be scaled in an orderly fashion and it has to be stated in the cluster blueprint.

When there is more than one node on a level the requests to that level are normally distributed using a *load balancing tool* — either a software or hardware based one. The responsibility for setting up load-balancing resides with the upper of two levels in the architecture, since that allows the layer below to stay agnostic about what is going on in the layer above.

There may also be a load-balancing entity between the clients and the front-end nodes, but they are blissfully unaware of this fact.

When more than one node of a given type is required in the logic layer they are often connected in order to coordinate their behaviour. The front-end nodes which are normally stateless or do not share any data, so they work in isolation. For service nodes it varies more since the nature of the service dictates if there is a need for sharing data. Sharing data is normally the reason for connections between nodes of the same type.

Note: sharing in this setting means that nodes of the same type has a shared data storage. E.g., they could be sharing an Mnesia table.

A *node family* is a description of how nodes of a specific node type connects to other nodes. The nodes created according to the node family description are referred to as a node family when talking about a live system. In a multi-cluster system nodes of a node family running in the same cluster are called a *node sub-family*, but more about that later. The node family description is also part of the cluster blueprint.

So after scaling within a cluster the system may look similar to the one described in Figure 3, where there are two front-end nodes, three logical nodes and two service nodes, where SB_1 is an internal service. It is only the logic nodes that form a fully connected network. Furthermore, the communication between the nodes have been abstracted to be between the node families to indicate that there is load balancing in effect.

Scaling with Clusters In some cases it is not possible to scale a system by just adding more nodes to a cluster, e.g., there might be a limit to how many nodes there can be on the logic level before the coordination between them becomes a bottleneck, or a service node has some bandwidth limitation towards the external service it accesses.

In order not to make the architecture of a cluster too complicated when running into bottlenecks like this one can add an extra cluster that follows the same cluster blueprint. After adding a cluster the distribution of incoming requests from clients will be distributed across the clusters using the load balancing tool that sits between the clients and the top nodes.

Since the clusters have to form a coherent system some inter-cluster communication is often necessary. However, one should limit the communication since it defeats the purpose of using clusters as the way to scale a system when adding more nodes to an existing cluster does not work anymore. Furthermore, it is only inter-cluster connections between nodes from the same node family that makes sense —

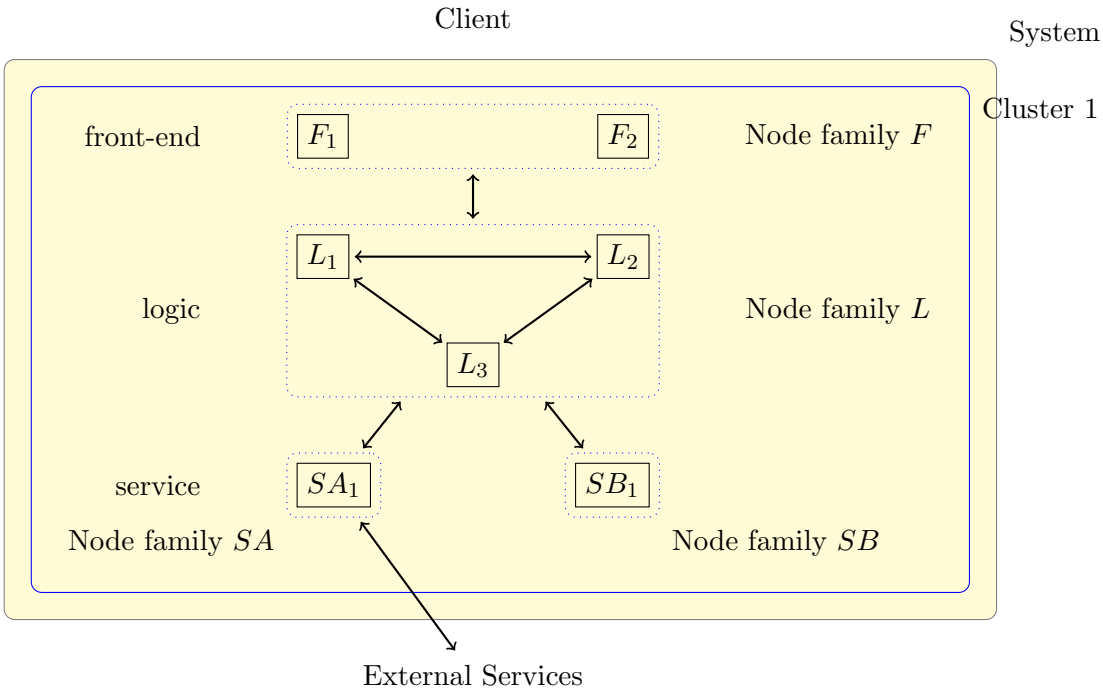


Figure 3: Internal system architecture after scaling

all other connections are forbidden for that reason. The nodes that talk inter-cluster are called *gateway nodes*. The gateway nodes tend to allocate less resources to intra-cluster duties than the non-gateway nodes in their node sub-family in order to be able to handle the inter-cluster communication.

2.1.3 System Deployment

The cluster blueprint only deals with the logical structure of a system whereas the *resource blueprint* specifies which resources that are available to deploy a system on. Resources being your own hardware or instances from one or more cloud providers [VRMCL08].

In the cluster blueprint each node type specifies what it needs to run, so that those needs can be matched with resources specified in the resource blueprint when it is time to deploy the system.

A concrete definition of these needs that fits in with the deployment from WP4 are described in D4.4[REL13c].

The combination of a cluster blueprint and a resource blueprint is called a *system blueprint*. The cluster blueprint allows one to understand the abstract structure of a distributed Erlang system, whereas the resource blueprint explains how machines to run the nodes on can be provided. The resource blueprint is the glue that helps out with mapping the abstract system architecture to physical or virtual machines when the entire system is deployed.

For pure analysis of system architecture the cluster blueprint is enough, and for legacy systems developed without DECO in mind one should create a cluster blueprint in order to describe the system architecture.

The resource blueprint only has value when one is using a tool like Wombat (see below) to deploy on an infrastructure. As will become clearer Wombat needs to have a way to make the cluster blueprint concrete if it is used for deployment of a system.

2.2 DECO and Wombat

In WP4 we have worked on Wombat (formerly known as CCL), which is a tool for deployment and management of Erlang systems. Wombat was first introduced in D4.2[REL12b].

It was during a discussion about how Wombat should deal with the deployment of big Erlang systems at the RELEASE project meeting in Athens February 2013 that we came to the conclusion that a common vocabulary for distributed Erlang systems was needed.

The main point about DECO is to get a way to describe Erlang systems in a uniform and unambiguous manner. Since Wombat has to deal with deployment and monitoring of Erlang systems, it is the intention to have a very close mapping between the concepts of DECO and what Wombat can support. This is based on the premise that if a concept is needed in DECO to describe an Erlang system, it would be very weird if that same concept wasn't present in Wombat.

Wombat uses a configuration file to represent a system and the top level is the representation of the system blueprint, i.e., the cluster blueprint and the resource blueprint. This correspondence continues down through the abstraction layers. In this sense DECO is a high-level specification of the concepts that Wombat has to support.

3 RELEASE Contributions to Distributed Erlang

The RELEASE [pro] project aims to make Erlang/OTP's concurrency and distribution model even more attractive for the development of large-scale systems. Quoting the project's description of work, the goal of the project is

> "to scale the radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on multi-core machines"

This section focuses on two contributions the RELEASE consortium has made — the SD-Erlang library will be included in an upcoming Erlang/OTP release. More specifically, the section presents how Wombat and SD Erlang have been integrated to present developers with a full-stack solution to deploy large-scale Erlang systems.

Wombat is a cloud deployment tool for distributed Erlang applications. It aims to provide *operations professionals* with a tool belt to seamlessly deploy Erlang applications across multiple infrastructure providers. Apart from provisioning host machines Wombat also does the following: handles all the required firewall configurations, transfers Erlang releases to the host machines, and adds new nodes into the system. One could say that Wombat's mantra is *"configure once, scale (anywhere) easily"*. Once Wombat has been provided with a *system blueprint*, a system can be easily grown-up/shrunk-down by just triggering an add/delete node action on Wombat. The rationale behind the development of Wombat is to provide infrastructure management tools specialised in Erlang/OTP systems. The RELEASE consortium benefits from Wombat by getting access to a testbed backed by multiple infrastructure providers.

SD Erlang is a language extension of distributed Erlang that aims to enable distributed Erlang to scale for server applications on commodity hardware with at most 10^5 cores [REL12a]. SD Erlang reduces the number of node connections and the number of nodes in a namespace [REL13a]. For this purpose we have introduced s_groups in SD Erlang, i.e. nodes in an s_group have transitive connections only with nodes from the same s_groups, but non-transitive connections with other nodes. Each s_group has its own name space, i.e. names registered in an s_group are replicated only to the nodes from the same s_group.

This section is organised as follows. Wombat and SD Erlang overviews are presented in Sections 3.1 and 3.2 respectively. A connection between Wombat node families and SD Erlang s_groups is covered in Section 3.3. SD Erlang intergration into Wombat is discussed in Section 3.4.

3.1 Wombat Overview

In this section we discuss how Wombat can be used to orchestrate the deployment of an Erlang application. When using Wombat, the steps involved in the deployment of an application are as follows: 1) registration of one or more infrastructure providers, 2) uploading of the Erlang release of the application, 3) defining one or more node families required by the application, and 4) deployment of one or more nodes in the defined node families.

1. *Registration of infrastructure providers:* For a node to be deployed, it requires a hardware and software infrastructure where it will run. It is the responsibility of an infrastructure provider to supply Wombat with the required hardware and software substratum required for running Erlang applications. Users can register as many infrastructure providers as they have access to. An example of an infrastructure provider is Amazon EC2 [Inc08]. To register Amazon as an infrastructure provider Wombat only requires user's Amazon EC2 credentials, i.e. access and secret keys.

2. *Upload of releases:* The next step is to upload to Wombat a tarball that contains the Erlang release of the application [pro]. All the files in which the Erlang node name is hardcoded must be turned into template files. Wombat dynamically generates new node names as new nodes are added to the system. Then it is Wombat's responsibility to update these template files with the corresponding node name.

3. *Definition of node families:* This is the most crucial step. It consists of defining a hierarchical overlay that contains description of multiple node families required by the to-be-deployed application. A node family is a logical grouping of nodes with an identical initial behaviour that run the same Erlang release. An Erlang system may consist of multiple releases; in this case, a Wombat user will need to define, at least as many node families as there are releases in the system. For a new node to join the system the node requires initial configuration information, which must be provided by a node that already belongs to that system, i.e. a bootstrap node. Wombat supports the definition of bootstrapping strategies at two levels: 1) within a node family, i.e. intra-node family bootstrap strategy, and 2) between node families, i.e. inter-node family bootstrap strategy. A bootstrap strategy has two parts: 1) a *bootstrap node selection strategy* which specifies how a node is marked as a bootstrap node among all the available nodes in a particular node family, and 2) an action to be performed by the joining node taking into consideration the bootstrap node information. Currently, Wombat supports the following *bootstrap node selection strategies*: random node and specific node selection strategies. The difference between these strategies is that in the random node selection strategy a node is picked randomly from a particular node family, whereas in the specific node selection strategy a concrete node is given by the Wombat user. A node family also contains the application-specific firewall configuration information required to enable the communication between nodes within and across node families.

4. *Node deployment:* To deploy nodes a Wombat user needs only to specify the number of nodes and the node family these nodes belong to. Nodes can be dynamically added to, removed from, the system depending on the needs of the application.

An example of how Wombat can be used to deploy a distributed Erlang application can be found in D64 [REL13d], which explains how Wombat has been used to deploy a real-world Erlang application on Amazon EC2.

3.2 SD Erlang Overview

In SD Erlang we distinguish two types of nodes: free nodes and s_group nodes [REL13b]. Free nodes in SD Erlang belong to no s_group and behave identically to distributed Erlang nodes. Similarly to distributed Erlang nodes SD Erlang free nodes can be either normal or hidden. Free normal nodes have transitive connections and share their namespace with other free normal nodes. A free hidden node does not share its connections and namespace with other nodes. An s_group node belongs to at least one s_group. S_group nodes have transitive connections with the nodes from the same s_group and non-transitive connections with other nodes. To find nodes with which a node shares connections and name spaces `s_group:own_nodes()` function is used. Function `nodes(connected)` returns a list of all connected nodes. Figure 4 shows types of connections between different types of nodes in SD Erlang. Here, nodes N1, N2, N3, and N4 are free normal nodes; H5 and H6 are free hidden nodes; S7, S8, S9, S10, S11, and S12 are s_group nodes. A solid line represents a transitive connection and a dotted line represents a non-transitive connection. Nodes S7, S8, S9, and S10 belong to s_group G1; nodes S9, S10, S11, and S12 belong to s_group G2.

A node can join an s_group at its launch or dynamically. The difference is only in the time when a node joins s_groups whereas the node behaviour in s_groups is identical. When a node joins s_groups at its launch `-config` flag is used (Listing 1). An example of an s_group configuration file is presented in Listing 2. When a node joins an s_group dynamically `s_group:new_s_group/2` and `s_group:add_nodes/2` functions are used (Listing 3).

Listing 1: Starting s_group group1 statically

```
Terminal1$ erl -name node1@glasgow.ac.uk -config s_group_config
Terminal2$ erl -name node2@glasgow.ac.uk -config s_group_config
Terminal3$ erl -name node3@glasgow.ac.uk -config s_group_config
Terminal4$ erl -name node4@glasgow.ac.uk -config s_group_config
```

Listing 2: File s_group_config.config

```
{kernel,
  [{s_groups,
    [{group1, normal, ['node1@glasgow.ac.uk', 'node2@glasgow.ac.uk',
      'node3@glasgow.ac.uk', 'node4@glasgow.ac.uk']}
    ]}]}].
```

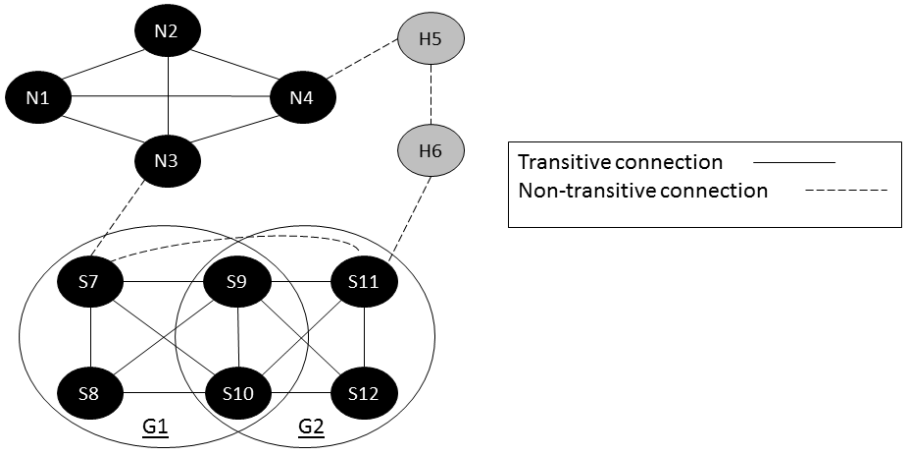


Figure 4: Types of Connections between Different Types of Nodes in SD Erlang

Listing 3: Starting s_group group1 dynamically

```
Terminal1$ erl -name node1@glasgow.ac.uk
Terminal2$ erl -name node2@glasgow.ac.uk
Terminal3$ erl -name node3@glasgow.ac.uk
Terminal4$ erl -name node4@glasgow.ac.uk
Terminal4$ s_group:new_s_group(group1, ['node1@glasgow.ac.uk', 'node2@glasgow.ac.uk',
'node3@glasgow.ac.uk', 'node4@glasgow.ac.uk']).
$
```

Details of SD Erlang and its functions can be found in [REL13a, REL13b].

3.3 Node Families and S_groups

Wombat Node Families (NFs) and SD Erlang s_groups may look similar because both of them group Erlang nodes. However, NFs and s_groups have different purposes, and are used in different context, as illustrated in figure 5.

A grouping in s_groups means that nodes in the same s_group have transitive connections between each other and non-transitive connections with other nodes. Nodes from the same s_group have a common namespace. *A grouping in NF* means that all nodes started in the same NF have the same initial configuration pattern. Wombat provides initial connectivity information required by an application, but it is up to the application to decide which communication mechanism to use. By a communication mechanism we mean a set of rules that enable Erlang nodes to exchange information. The mechanisms differ in the type and the number of connections between nodes. Examples of communication mechanisms are distributed Erlang, SD Erlang, and TCP. As the application evolves Wombat does not prevent nodes from establishing further connections.

3.4 SD Erlang integration into Wombat

The integration of SD Erlang into Wombat can be materialised as a set of Wombat built-in actions that can be carried out at bootstrapping time. The election of one action or another will have implications on the number of s_groups that will be created and the connectivity between nodes.

In Wombat we distinguish two types of bootstrap strategies: intra-NF and inter-NF bootstrap strategies. Inter-NF and intra-NF bootstrap strategies define initial connections of a new nodes with other nodes from the same NF and remote NFs respectively. In Figures 6, 7, and 8 we present six intra-NF and twelve inter-NF bootstrap strategies related to SD Erlang. Wombat does not limit the number

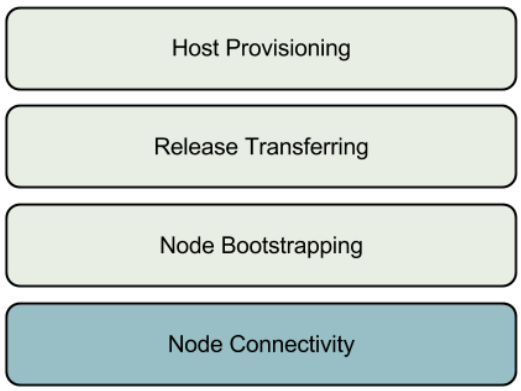
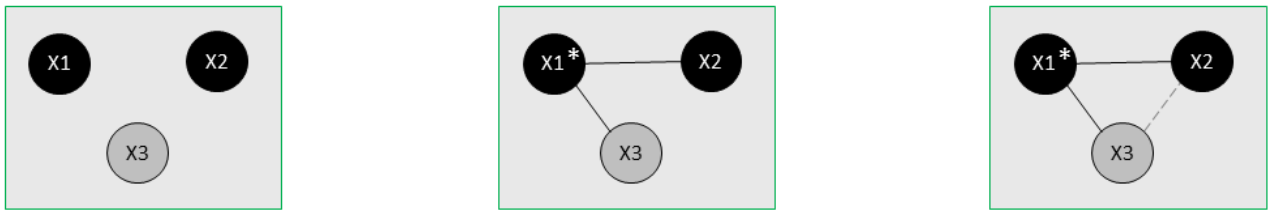


Figure 5: Wombat (yellow) and SD Erlang (green) Responsibilities



(a) (b) (c)

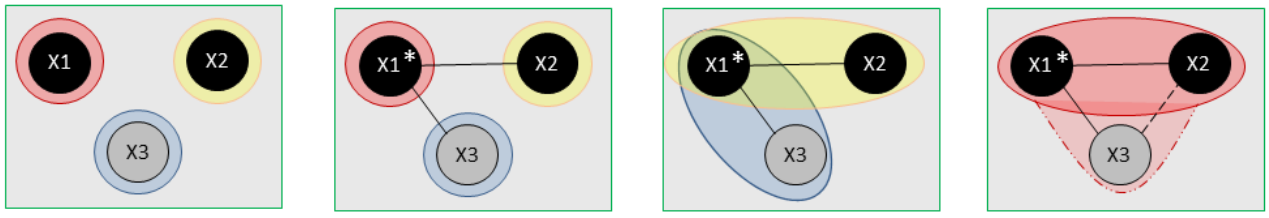
Figure 6: Intra-NF Bootstrap Strategies with no S_groups

of bootstrap strategies, and more strategies will be introduced as the tool develops. In the figures a rectangle represents an NF, a black circle represents a running NF node, and a grey circle represents a newly started node that joins the NF. S_groups are shown as coloured ovals that group together one or more nodes; different colours refer to different s_groups. Intra-NF and inter-NF bootstrap nodes are marked by * and # respectively. To simplify the description of the strategies all examples presented below use a specific-node bootstrap strategy introduced in Section 3.1.

Figure 6 shows intra-NF bootstrap strategies for free nodes, i.e. when a new node is started in an NF initially it neither creates nor joins any s_groups in its NF. Figure 6(a) shows that new node $X3$ does not establish connections with other NF nodes. Strategies in Figures 6(b) and 6(c) are similar in that when new node $X3$ is started it establishes a connection with the bootstrap node $X1$. The difference is only in the type of nodes, i.e. in Figure 6(b) the nodes are free hidden, so they establish non-transitive connections, and in Figure 6(c) the nodes are free normal so, the connections are transitive.

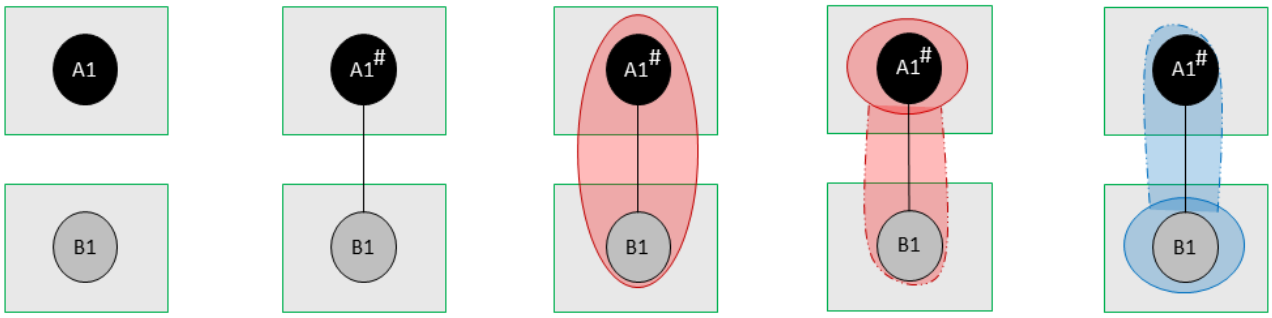
Figure 7 shows intra-NF bootstrap strategies for s_group nodes, i.e. when a new node is started in an NF initially it either creates or joins an s_group in its NF. In Figure 7(a) every new node creates its own s_group but neither joins other s_groups nor connects to the nodes from the same NF. In Figure 7(b) new nodes create their own s_groups and connect to the bootstrap node. In Figure 7(c) a new node creates its own s_group and adds the bootstrap node in it. In Figure 7(d) the first node in the NF $X1$ creates an s_group and every new node joins that s_group.

Figure 8 presents inter-NF bootstrap strategies. To simplify the description and focus only on the inter-NF strategies in Figure 8 the NFs can be replaced by any of the intra-NF bootstrap strategies presented in Figures 6 and 7. In Figure 8(a) nodes from NFs A and B do not connect to each other. In Figure 8(b) every new node from NF B connects to the inter-NF bootstrap node from NF A. In Figure 8(c) every new node from NF B creates an s_group and adds the inter-NF bootstrap node from NF A in it. In Figure 8(d) every new node from NF B is added to an s_group of the inter-NF bootstrap node from NF A. In Figure 8(e) the inter-NF bootstrap node from NF A is added to an s_group of every new node from NF B.



(a) (b) (c) (d)

Figure 7: Intra-NF Bootstrap Strategies with S_groups



(a) (b) (c) (d) (e)

Figure 8: Inter-NF Bootstrap Strategies

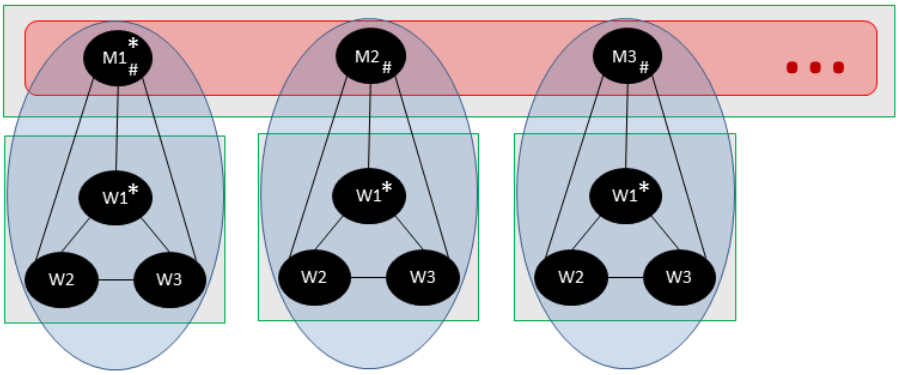


Figure 9: NF and S_group Configuration in Orbit

4 Case Study: Orbit

In this section we discuss an example of configuring Wombat to run SD Erlang application Orbit in clouds. The purpose of Orbit application benchmarking is to compare performances of distributed Erlang and SD Erlang applications. We use Wombat to simplify the process of node deployment on clouds and automate node grouping in s_groups for different runs.

Orbit [APR+12] is an example of distributed hash table applications which are one of the typical class of distributed Erlang applications. The Orbit's goal is to find the least subset X_{orb} such that $X_{orb} \subseteq X = [x \mid x \in \mathbb{N}, x \leq N_1]$ and $f1, f2, ..., fn : X \to X$ where $f1, f2, ..., fn$ are generators and X_{orb} is closed under all generators. The idea is that every node keeps a part of the table, and while the application is running, processes are spawned between nodes to fill the table. Thus, the larger N_1 and the more complex generators are used the larger computation and storage resources the benchmark requires.

To implement Orbit Erlang nodes can be grouped in a number of ways. In Figure 9 we present one of many possible Wombat configurations. We define two types of NFs: one master NF and a number worker NFs, i.e. one worker NF for every node from the master NF. Both master and worker NFs have the same intra-node bootstrap strategy. The first node creates an s_group and then the remaining NF nodes join this s_group (Figure 7(d)) using random node selection strategy. After a new node completes its intra-NF bootstraping the node makes sure that the inter-NF bootstrap node defined by the spesific node selection strategy is also in the same s_group (Figure 8(e)).

5 Conclusion and Future Work

While DECO has already helped in the internal ESL discussions regarding how to implement features in Wombat the next step is to present it to the Erlang community or a sub-set of it in order to ensure the general applicability of the model.

DECO has already helped us understand how Wombat and SD-Erlang fit together and it is through integrations like this that the concepts of DECO will be validated and enhanced.

Inside ESL we will be training our staff in the concepts of DECO— mainly through using Wombat on the big projects.

One thing that is missing from DECO at the moment is how to handle *node failures* in a manner similar to how Erlang handles *process failures*. Wombat suffers from this as well. Right now Wombat will use the cluster blueprint's specifications to restart a node if it looses connection with it. Unfortunately, the case where the node had just lost network connectivity, but was still running has not been addressed. This problem is closely related to *network splits* which are notoriously difficult to deal with in a generic manner. It would be very interesting to investigate this problem and try to identify patterns in architectures that are easier to deal with than others, but this is considered to be outside the scope of the RELEASE project scope.

Another thing that needs work in the future is how to visualise a cluster blueprint. Apart from the squares-and-circles diagrams in the Erlang/OTP Design Principles User's Guide[otp] there is only one attempt in the Erlang community to come up with a way to describe Erlang systems, which was done by Kresten Krab Thorup in his keynote for the 2012 Erlang Workshop[Tho12] (unfortunately the slides are not publicly available). This is clearly the next step and an important one for mainstream adoption of Erlang and DECO.

6 Change Log

Version	**Date**	**Comments**
0.1	13/07/2013	First Version
0.2	1/9/2013	Revised version

A Definition of terms

This section is a list of all the DECO terms in no particular order. The definitions strive to be more elaborate and precise than in the narrative text above.

release software bundle that conforms to the Erlang/OTP release criteria. A release consists of a number of loosely coupled Erlang applications. It is characterised by a version number, can be installed, upgraded, downgraded and removed as a whole. All items in a release have a version number and are organised according to a predefined directory structure which includes the Erlang run-time system. Some releases do not conform to this structure and will use a single ERTS installation, linking to its executable.

node a node is an instance of an Erlang run-time system, running a given release on a provisioned machine. An Erlang node is said to be alive when it is running distributed Erlang.

node type This is a purely descriptive term that allows an architect to talk about how the system works. In pure Erlang/OTP terms all releases are releases, but they perform different tasks in a system and that is what the node type is used for. A node type is associated to one or more releases with different version numbers.

node family a set of nodes of the same type with a description of if and how the nodes in the family connect together. There can be different connection types inter-cluster and intra-cluster. Nodes in the same cluster is referred to as a *node sub-family*.

cluster blueprint a description of which kinds of nodes are contained by a cluster and how they connect to each other.

cluster is an instantiation of a cluster blueprint and consists of one or more nodes logically grouped together. These nodes are not necessarily connected to each other. The main reason for clustering is scalability. By adding more clusters, one gets a system to scale.

system blueprint describes how one can build an Erlang system. Uses a cluster blueprint and a resources blueprint.

system an Erlang system consists of one or more clusters. The system deals with requests from its clients and uses zero or more external services to do its job.

resources blueprint describes which resources that are available to deploy a system on. It can be private hardware or cloud instances from multiple cloud provides.

provisioned machine a machine that is capable of running one or more Erlang nodes. When deploying a resource from the resources blueprint, the provisioned machine is started and provisioned according to the specification in the resources blueprint.

bootstrapping strategy is a description of how a new node gets to know existing nodes in a cluster and how the existing nodes get to know about the new node. Depending on the cluster blueprint the new node then connects to the existing nodes and the existing nodes that need to take the new node into account will do so.

client is an external program that uses an Erlang system. A client could be a web browser, a mobile application or a node on another machine. Clients handled by a particular system could range from one to millions.

front-end node deals with connections to clients. Does the initial decoding of client requests before sending them on to the logic nodes. Encodes replies and messages from the logic nodes before sending them to the clients. Normally built to handle a large amount of client connections such as web sockets, it would keep the socket connection open and encode / decode all JSON requests which would be forwarded to logic nodes as Erlang terms. A front-end node is usually stateless and CPU bound. Any user session data could be duplicated in the logic nodes, so if a front-end node terminates, existing clients would establish a new connection towards another front-end node without having to recreate a new session.

logic node implements the business logic of the system. Takes decoded client requests and processes them. Normal processing typically involves converting a request into a request to a service node and then later sending the reply from the service node to the front-end node that did the original request. A logic node may keep state information about the transactions/interactions/sessions taking place with the client. Often logical nodes of the same type will share data (using Mnesia, Riak or another distributed database) for fault tolerance purposes. Dividing front-end and logic nodes gives the advantage that if a front-end node carrying hundreds of thousands of connected users crashes, the session would not be lost. All they would have to do is reconnect to another front-end node.

data sharing strategies Nodes of the same type can have different data sharing strategies. Choice of these strategies will affect the scalability of the system, with the share-nothing approach givingt close to linear scalability and the share-everything approach giving almost no scalability at all.

share-nothing is when logic nodes of the same type share nothing. If we implemented an instant messaging server with a share nothing architecture, the session data and the messages going through the system would be lost if the node were to terminate. A client would have to log in again, create a new session and resend the message.

share-something is where some, but not all of the data is duplicated across nodes. So in our instant messaging example, all session data could be duplicated across logic nodes of the same type, but instant messages themselves wouldn't. If a logic node crashes, user requests would be redirected to another node where a copy of the session data is stored. Any messages going through the node which crashed are lost.

share-everything Is where all data is duplicated for redundancy purposes. If a node crashes, a redundant node takes over and sessions or messages are not lost.

external service an external provider of some service where there is a well-defined protocol defined for interacting with that service.

service node a node that either provides access to an external service or provides a stand-alone service that other nodes can make use of. External service nodes are proxy and transport.

proxy simple bridge to an external service. IP connectivity.

transport stateful interaction with an external service. Has logic and IP connectivity.

local service performs a certain service that is done on its own node in order not to overload other nodes in the system. Has no connectivity to external services.

node connections nodes can be connected in a number of different ways:

distributed Erlang the built-in TCP based node connectivity.

protocol plain TCP, UDP, SSL and so on.

pub-sub messaging solutions such as AMQP, 0MQ or XMPP that provides a reliable messaging backbone based on a publish-subscribe paradigm.

load balancing if there is a group of nodes that should be load balanced, the users of that group have to know which load balancing algorithm to use in relation to the available number of worker nodes. Inside an Erlang system there will be no classical load balancing component, that will only be used in front of the front-end nodes. Possible load balancing strategies are:

random just pick a random worker node.

round-robin pick the worker nodes in sequence.

persistent hashing normally done with 2-levels to avoid serious re-hashing when failed nodes recover.

consistent hashing preferred over persistent hashing in most cases since the number of keys that needs to be remapped is smaller on average.

References

[APR+12] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos F. Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for erlang/otp. In Torben Hoffman and John Hughes, editors, *Erlang Workshop*, pages 33–42. ACM, 2012.

[Inc08] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008.

[LMTV11] Peter F. Linington, Zoran Milosevic, Akira Tanaka, and Antonio Vallecillo. *Building Enterprise Systems with ODP: An Introduction to Open Distributed Processing*. Chapman & Hall/CRC, 1st edition, 2011.

[otp] Otp design principles user's guide. `http://www.erlang.org/doc/design_principles/users_guide.html`.

[pro] RELEASE project. Release homepage. `http://www.release-project.eu/`.

[REL12a] RELEASE Project. Deliverable D3.1: Scalable Reliable SD Erlang Design, June 2012.

[REL12b] RELEASE Project. Deliverable D4.2: Homogeneous Cluster Infrastructure, July 2012.

[REL13a] RELEASE Project. Deliverable 3.2: Scalable SD Erlang Computation Model, March 2013.

[REL13b] RELEASE Project. Deliverable D3.3: Scalable SD Erlang Reliability Model, September 2013.

[REL13c] RELEASE Project. Deliverable D4.4: Capability-driven Deployment, May 2013.

[REL13d] RELEASE Project. Deliverable D6.4: Homogeneous Deployment of a Load-testing Tool, May 2013.

[Tho12] Kresten Krab Thorup, September 2012.

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2008.