# Engineering the Cloud from Software Modules

Jan S. Rellermeyer Michael Duller Gustavo Alonso
Systems Group, Department of Computer Science
ETH Zurich, 8092 Zurich, Switzerland
{rellermeyer, michael.duller, alonso}@inf.ethz.ch

## Abstract

*Cloud computing faces many of the challenges and difficulties of distributed and parallel software. While the service interface hides the actual application from the remote user, the application developer still needs to come to terms with distributed software that needs to run on dynamic clusters and operate under a wide range of configurations. In this paper, we outline our vision of a model and runtime platform for the development, deployment, and management of software applications on the cloud. Our basic idea is to turn the notion of software module into a first class entity used for management and distribution that can be autonomously managed by the underlying software fabric of the cloud. In the paper we present our model, outline an initial implementation, and describe a first application developed using the ideas presented in the paper.*

## 1. Introduction

CLOUD COMPUTING is used to refer to many different technologies: from outsourcing of information processing to the usage of external data centers. Existing cloud services range from virtualized server environments such as AMAZON EC2 [13] to mash-up platforms for external services such as YAHOO PIPES [15]. Regardless of the specific definition used, the underlying software fabric for cloud computing is not new: cloud computing is typically implemented as a distributed or parallel application running on a virtualized cluster of computers. As such, cloud computing software suffers from the same problems that plague traditional distributed and parallel software, i.e., they are complex to design, develop, test, deploy, and manage. To make the vision of pervasive cloud computing a reality beyond simple services, powerful and sound application models are needed to simplify the development and operation of the software behind the functionality implemented in the cloud. Such models only exist for specific problems and are restricted to very narrow forms of data processing, e.g., processing of parallelizable batch jobs [2, 5] or data storage [1].

In this paper, we outline our vision of a unified application model for cloud computing software. Our model exploits conventional software modularity to define the entities that will compose the distributed application and uses an underlying runtime module management platform to hide most of the complexity of the distributed application form the programmer. The model supports the development and testing of applications on a single machine with, e.g., network communication being transparently added by the software fabric when the application is deployed to the cloud.

In this short paper, we outline the model and describe a first implementation based on OSGi [7], the dynamic module system for Java. Through examples, we show that the model is independent of the programming language and runtime. Finally, we demonstrate the feasibility of our approach and the potential for the model by presenting XTREAM, a personal stream management system for the cloud implemented using the model and ideas described in the paper.

## 2. Modularity as the Basis for Distribution

Cloud computing is still hampered by the lack of a proper methodology for developing suitable software and faces many of the challenges of distributed and parallel software. For instance, the initial challenge is to divide the application into units that can be deployed in a distributed setting. Often this is done using tiers as in conventional multi-tiered applications. Unfortunately, there are few design rules or even tools to help in the design of such systems. In reality, each tier is also a distributed application in itself (e.g., an application server), making the problem even more complicated. In particular, when the underlying problem is not trivially parallelizable, there are many possible designs and the implications of each one are hard to predict. This problem is composed by the fact that commodity middleware platforms are invasive against the application code: communication with remote entities is typically explicit and intertwined with the application code. As a consequence, debugging such applications is hard and testing tends to become both time-intensive as well as expensive.

Our vision for cloud computing software is to provide the means to build cloud applications ignoring the fact that the final deployment will be distributed. To do so, we propose to use the widely accepted notion of software module as the unit of management and distribution and let a module management platform deal with the complexity of distributed deployment, execution, and maintenance.

Modules are units of encapsulation. Separating the code into modules encourages the developer to define interfaces and thereby to shape coupling and cohesion of the code. Cohesion means that all the functions of a module should ideally be closely related so that the resulting module encapsulates common functionality. This functionality is ideally only exported to other modules through a narrow set of interfaces to avoid coupling. Coupling means that one module depends on internal implementation details of a second module. Tight coupling is not only a burden for code evolution but also for reusing modules in a different application context. It also makes distributed deployment and maintenance very complex and unmanageable for large systems.

In cloud computing, however, it is very important to be able to reuse pieces of software as services. If both high cohesion and loose coupling can be achieved, the resulting code base can be treated as a collection of independent modules communicating through service interfaces, with the communication being managed by an underlying runtime platform.

Designing modular applications is easier than designing distributed applications. The design principles of modules are much better understood [8, 14] and are by now common programming practice. However, structuring the code base into modules does not automatically make a well-structured system. Module systems which require explicit statement of dependencies make the design process more manageable and controllable. The application can be developed using all the standard tools available for centralized, monolithic applications. Most importantly, the application can be tested in isolation and on a single machine, enabling agile and iterative development techniques, which are generally much harder to use for distributed software. As a further advantage, many modular approaches provide means for deploying modules to machines. For instance, web applications can be deployed through management interfaces of the application server. The same techniques can be combined with autonomic decision making to let the deployment and management of the cloud software in the hands of a runtime platform that manages software modules as first class entities and the main units of deployment and distribution.

We propose to use these basic ideas as the core of a model for cloud computing. The advantage of modularity should be obvious. What remains to be shown is that modules can really be used to build distributed applications on the cloud, to define the functionality of the underlying runtime platform, and to show with an example how the idea may work in practice. With the proper software fabric underneath, our model allows programmers to focus solely on developing the functional aspects of an application in the form of software modules and treat the distribution in the cloud as an orthogonal concern. While the modules are agnostic to any distribution-specific requirements, it is the task of the fabric to support the deployment of the modules and handle the calls between distributed modules. Among the many and different module approaches, OSGi [7] for the Java language is one of the most popular and widely used systems. Hence, in the following section we discuss how to borrow the ideas of OSGi and extend them to create such a software fabric.

## 3. Universal OSGi: The Fabric for the Cloud

OSGi is an open standard for dynamic software modules in Java. Additionally, it specifies a runtime infrastructure, the so-called FRAMEWORK, to load new modules and manage the life cycle of modules at runtime. In order to do so, OSGi loads each module through a separate class loader. By making the dependencies between modules explicit, OSGi can dynamically compose the modules at runtime while at the same time preserving the type consistency within the VM. Imports from another module lead to a delegation to the class loader of the dependency. Besides these tightly coupled dependencies, OSGi facilitates the loose coupling of modules through services. Every plain Java object can be a service when it is published under a set of service interfaces. Clients can retrieve services from a central service registry by the name of the interface and, optionally, filtered by predicates over service attributes.

The OSGi model is a very advanced way of dealing with modules. However, two intrinsic features of the design prohibit the use of OSGi in the cloud. Firstly, OSGi only deals with services running on the same Java virtual machine. In this regard, OSGi is able to loosely couple a centralized application running on a single address space but has no support so far for dealing with remote services. Secondly, an implicit assumption of OSGi is that applications are written in Java and that a Java VM is available on each node. Clearly, this is not applicable in the cloud. However, a distinct advantage of loose coupling through services is that service clients have no dependencies to the implementation of a service but only on the service interface (Figure 1).

As a result, it is possible to bridge between different OSGi frameworks by providing service proxies for remote services. In order to preserve the expected behavior of a centralized application, the service proxies have to be pro-

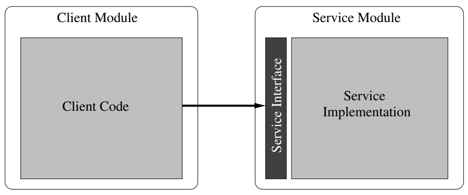

**Figure 1. Service clients are resolved against the interface and have no dependency to the service implementation.**

vided by proxy modules which exhibit the same properties as the remote module has. Furthermore, OSGi-based clients can transparently talk to services written in any language as long as the host running the service is able to understand the protocol. For a consistent behavior with regard to a local OSGi service, this involves that the service provider is able to

a) share life cycle state information and other meta data with the client

b) provide a Java interface for the service upon request

c) invoke the service upon request and send the result back to the client.

*a)* is not particularly challenging to implement and trivial for static language runtimes that do not allow dynamic code loading and unloading. *b)* only requires the service provider to provide a chunk of binary data that makes a Java interface but not to understand the interface. *c)* potentially involves Java serialization since service invocations could ship complex Java types as arguments. However, the verbs and objects used in the communication between the client and the service are solely defined by the service through the service interface. As a consequence, services can restrict themselves to simple types and a small set of derived complex types. This approach is taken by most web service protocols like SOAP [6]. If, however, it is a requirement to support the entire expressiveness of the Java type system, this does not prevent services from being written in other languages. As we have shown in [11], it is sufficient to statically generate the fragments of the Java serialization protocol required for serializing and deserializing the types involved. As a proof of concept, we implemented remotely accessible OSGi services, for instance, for embedded systems written in C or for the TinyOS [4] sensor network platform.

Summarized, the concept of modularization is language-independent and the OSGi model allows non-Java services to participate in cloud applications. With the Universal OSGi approach, we mean that the modularity of the OSGi standard is pervasively used for managing modules and interoperate through services, regardless of the implementation of the service and the communication protocol. For the cloud, the availability of a universal modularity framework means that it will soon be possible to construct modules in different languages, deploy them to computers, instruct the fabric to bind clients to remote services, and design cloud applications that do not explicitly have to deal with distribution. Another intrinsic feature of the cloud that perfectly matches the modular design paradigm is the *elasticity*. On-demand acquisition of computation resources together with pay-as-you-go business models enable service providers to scale their resources seamlessly with the number of users of the service. Modules are already a viable deployment unit and support on-demand relocation and replication for stateless services. The software fabric can bind service proxies to multiple redundant service implementations in the cloud and seamlessly switch from one to another service. For services containing state, replication among a varying number of replicas can as well be added as an orthogonal concern.

## 4. R-OSGi: A First Step for Modules into the Cloud

R-OSGi [10] is the conceptual extension of the OSGi model for distributed systems. The R-OSGi software fabric facilitates transparent distribution along the boundaries of modules by turning service calls into remote service calls. OSGi services from remote peers can be accessed through proxy modules, dynamically generated by the client on demand. These proxies do not only register a service stub but also show the same behavior and are synchronized with the state of the original service module. This is important for a consistent behavior between a local setting for rapid prototyping and the productive distributed environment in the cloud.

Since OSGi does not impose any restrictions on services and in fact accepts potentially every Java object to become a service, R-OSGi has to deal with the same miscellaneousness that is expressible in Java. In particular, the service interfaces of remote services can have (tightly coupled) dependencies to either classes from inside the original module or other modules. R-OSGi provisions dependencies through two different mechanisms. Intra-module dependencies are injected into the proxy module (a process which we coined TYPE INJECTION). The result of the injection of dependency classes is that the content of the proxy module is equivalent to that fraction of content of the original module that is exposed through the transitive closure of the service interface and all its intra-module dependencies. Inter-module dependencies are provisioned to the runtime before installing the proxy by creating copies of the respective modules on the client.

As discussed in the previous section, the OSGi model is generally capable of dealing with non-Java services. In particular, the service-oriented approach can be easily used to interoperate with existing web service standards as they are often found in the cloud. One limitation, however, is that the application itself (the service client) still has to be written in Java. This is arguably against the openness of the cloud. The reason for the dependency to Java is that the client of a remote service has to support the generation and loading of a dynamic proxy module.

In R-OSGi, this is implemented with dynamic byte-code engineering and wrapping the resulting proxy class into a module so that the ordinary OSGi facilities for loading and unloading modules can be used. In other language runtimes, such facilities do not exist because OSGi is not defined for those platforms. We are currently in the process of implementing an OSGi-like platform for C as part of the BARRELFISH [12] multi kernel operating system. As a preliminary result, it can be reported that implementing an OSGi-like system for a runtime which usually runs modules in different address spaces leads to a system design which is between the two extremes defined by standard OSGi and R-OSGi. Direct references to services like in OSGi are not possible since they cross address spaces. Instead, service calls have to resort to IPC, which make them similar to R-OSGi remote service calls. Besides implementing OSGi-like runtimes for other languages, it is a goal to develop a mechanism for interoperability between platforms. This is particularly challenging when different kinds of languages like imperative and functional languages are involved.

With the R-OSGi fabric, a large class of OSGi applications can be developed as modules and distribution can be added as an orthogonal concern. This POJO-like application model is particularly attractive for building applications in the cloud. In [9], we have presented a tool which enables the composition of distributed deployments by dragging and dropping modules and then deploying the resulting setting to running machines. This deployment tool even supports advanced aspects of distributed systems like load balancing and fail-over redundancy.

## 5. Use Case: XTream

### 5.1. Overview

We illustrate the suitability and potential of our approach using the XTream prototype. XTream is a personal stream management system, which targets applications that collect, process, and disseminate personal information by treating arbitrary source thereof as sources of data streams. The potential, challenges, and an initial model for personal data streams have been demonstrated in [3].

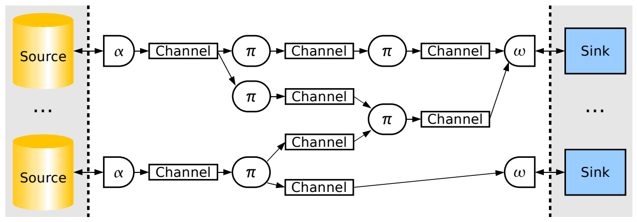

**Figure 2. Data processing model of XTream**

The application scenario for personal information processing using XTream includes a large number of people around the world that use XTream to manage and process their personal information and also exchange parts of it with their friends and colleagues. They will access their processed information from a variety of end devices like mobile phones or desktop applications and are only interested in this processed information, not how and where it has been precisely processed. Hence, XTream service providers can offer processing services to people and realize the actual processing in the cloud, making this application scenario a perfect match for a cloud application. We briefly recapitulate the XTream model, present its service-oriented design, deliver brief insight into the implementation of the prototype, and conclude how XTream can be run in the cloud easily due to its modular, service-oriented design.

Figure 2 illustrates the XTream model. In this model, applications are represented in a mesh of adapters and processing elements, called SLETS (for stream-lets), which are connected by CHANNELS. On the left, arbitrary sources like e-mail accounts, instant messaging chats, photo blogs, smart appliances in the home, or mobile phones are adapted by $\alpha$*-slets*, which bring data into the system. On the right, sinks like graphical user interfaces (both on desktops and mobile devices), digital photo frames, or actuators in the home are adapted by $\omega$*-slets*. In between, a mesh of $\pi$*-slets* and channels processes ($\pi$-slets) and buffers and forwards (channels) data.

The key challenge of personal data stream applications that XTream tackles is the dynamic nature that is inherent to these applications. The system must provide continuous operation of the applications while changes happen at runtime. On the one hand, the application mesh of a user changes over time to follow the user's personal needs and preferences. These changes include adding and removing new kinds of operators—and thus loading and unloading code—as well as the ability to move or replicate parts of the mesh to another device of the user. On the other hand, the meshes of individual users exchange data with each other and thus federate to a global mesh. Since a user only has control over his own mesh, XTream must also be able to gracefully deal with unpredicted changes from outside its own mesh like, e.g., abrupt disconnection.

## 5.2. Modular, Service-Oriented Design

To deal with these challenges, XTream is designed in a modular way and using services to provide a managed and loose coupling of components through well defined interfaces. A specific kind of slet, e.g., an $\alpha$-slet for IMAP mail servers, represents one module that can be loaded by XTream. From this slet class, multiple instances of this specific kind of slet can be created. Every instance registers a number of services. One of these services is an `Slet` service representing the slet itself and providing management methods. The other services are any number of `InputPort` and `OutputPort` services, which provide for data exchange between slets and channels.

Likewise, channel modules provide different kinds of channels (e.g., buffering/non-buffering, persistent/non-persistent) that create channel instances, which register a `Channel` service for management and `ChannelInput` and `ChannelOutput` services for data exchange.

The extensive and uncompromising SOA design of XTream facilitated its development significantly because some cumbersome issues like bookkeeping tasks are accomplished by the SOA runtime—we will illustrate this with an example in the subsequent section. Service types (service interfaces), unique service identifiers, and arbitrary service properties are sufficient to keep management information about individual system elements (slets and channels) and their wiring.

In addition to running a mesh of channels and slets, XTream also makes heavy use of modularization and service-based interaction for administrative tasks like monitoring a processing mesh and modifying it. The module boundaries of XTream have been chosen to maximize cohesion and minimize coupling where possible. Features of XTream that directly result from cohesion and coupling properties are, for example, the ability to load and unload management and monitoring modules at runtime and to extend the system with different variants of slets and channels.

## 5.3. Prototype Implementation

The design outlined above has been implemented in a prototype on top of R-OSGi and thus written in Java. Once Universal OSGi is available, the programming model of XTream will be significantly broadened as slets can then be written in any language.

Modules like the channel module, monitoring and management modules, or different kinds of slets are implemented as bundles (which is OSGi's module implementation). These bundles register and consume a number of services as outlined above. This service-based interaction between modules is realized as standard OSGi services, registered with the OSGi service registry and making heavy use

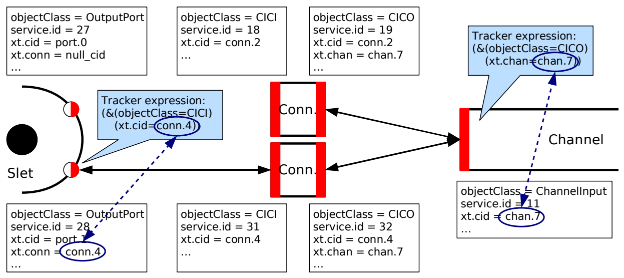

**Figure 3. Component interaction in XTream**

of service properties. Thus, a significant amount of system state, e.g., which slets are connected to which channels, is already kept directly in the SOA runtime, which facilitates not only the implementation of the prototype in terms of keeping state but also in terms of accessing this state for routine system operation. Figure 3 illustrates how state is kept and directly accessed for routine system operation with the example of the binding between slets to channels.

Figure 3 shows a small extract of an XTream mesh with half of an slet depicted on the very left and half of a channel on the very right of the figure. In the middle, two CONNECTORS are shown. These are not present in the processing model shown in Figure 2 and are only part of the implementation model, on which we do not elaborate in detail in this paper. For the sake of understanding how component interaction has been implemented, it is enough to understand that connectors are simply another level of indirection between slets and channels and provided by a corresponding connector bundle.

The bars in the figure represent services registered by the respective components. On the left, the slet registered two `OutputPort` services, the connectors registered one `CICI` and one `CICO` (Channel Input Connector Input/Output) service each, and the channel a `ChannelInput` service. Depicted in boxes next to the services are their respective service properties. These contain a unique ID for each service and, in the case of ports and connectors, to which connector or channel they are connected. For example, the binding of the lower port of the slet to the lower connector is recorded in the port's service property `xt.conn`, which is set to the unique ID `conn.4` of the connector. When this port wants to call the connector it is connected to (e.g., to send data to it), the connector service can be easily fetched using its unique ID recorded in the port's service properties. Instead of fetching the connector service every time it is called, we use the OSGi service tracker that automatically tracks the connector service for the port. The tracker expression used is depicted in the legend box. Likewise, the channel can easily access all connector services that are connected to it using a simple expression. Note that the tracker expressions are always of constant length and can be constructed solely from the component's own, local service properties, which is indicated by the dashed arrows.

The component interaction presented above is only one example how service-oriented design can even facilitate the implementation of a system. Besides its own services, the XTream prototype also uses a number of predefined OSGi services, e.g., for logging or keeping state persistent.

### 5.4. XTream and the Cloud Made Easy

XTream defines an application model for building applications on data streams. In terms of XTream's application domain of personal information processing, the ability to run and be managed autonomously in the cloud is an important property of the system to enable everyday users to define, run, and access their personal information processing applications without bothering about the *wheres*, *hows*, *whens*, and further challenges of deploying and running a distributed application. The way how XTream achieves this is by assimilating the modular paradigm into its application model. Users of the system never have to deal with the implementation of channels. Neither do they have to worry about how to connect different slets. Instead, XTream provides a set of prefactored source connectors and operators which can be reused in new applications. The entire system is nevertheless fully configurable. Custom slets can be added at any time and seamlessly integrated into applications because of common interfaces defined through the slet model.

## 6. Conclusions

The example of XTream shows that a clean modular design supported by services can be sufficient for designing cloud applications independent of the fact that the resulting deployment will be a distributed system. For XTream, this can be achieved by the R-OSGi software fabric, which extends ordinary Java OSGi applications to run in distributed settings. Our vision is to generalize this approach and turn it into a generic application model for building cloud applications.

## Acknowledgements

Part of this work has been funded by Microsoft as part of the Innovation Cluster for Embedded Software (ICES) initiative.

## References

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, 2004.

[3] M. Duller, R. Tamosevicius, G. Alonso, and D. Kossmann. XTream: Personal Data Streams. In *SIGMOD*, 2007.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[6] H. F. Nielsen, N. Mendelsohn, J. J. Moreau, M. Gudgin, and M. Hadley. SOAP version 1.2 part 1: Messaging framework. W3C recommendation, W3C, June 2003.

[7] OSGi Alliance. *OSGi Service Platform, Core Specification Release 4, Version 4.1*, 2007.

[8] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12), 1972.

[9] J. S. Rellermeyer, G. Alonso, and T. Roscoe. Building, Deploying, and Monitoring Distributed Applications with Eclipse and R-OSGi. In *ETX '07: Fifth Eclipse Technology Exchange Workshop*, 2007.

[10] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, 2007.

[11] J. S. Rellermeyer, M. Duller, K. Gilmer, D. Maragkos, D. Papageorgiou, and G. Alonso. The Software Fabric for the Internet of Things. In *Proceedings of the International Conference on the Internet of Things*, 2008.

[12] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[13] A. W. Services. Amazon Elastic Compute Cloud. `aws.amazon.com/ec2`.

[14] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[15] Yahoo! Inc. Yahoo Pipes. `http://pipes.yahoo.com/`.