FLOW-BASED PROGRAMMING AS A SOLUTION FOR CLOUD COMPUTING REQUIREMENTS

Marcel R. Barros[1], Charles C. Miers[1,2], Marcos Simplício[1], Tereza C. M. B. Carvalho[1], Jan-Erik Mångs[3], Bob Melander[3] and Victor Souza[3]

[1]*Laboratory of Computer Architecture and Networks, University of São Paulo (USP), São Paulo, Brazil*

[2]*Santa Catarina State University (UDESC), Florianópolis, Santa Catarina, Brazil*

[3]*Ericsson Research – Packet Technologies, Kista, Sweden*

Keywords: Cloud computing, Flow-based programming, Functional programming.

Abstract: Cloud computing services provide a new way of deploying applications over the Internet, as well a prominent approach for achieving enhanced scalability. Usually, exploration of cloud computing resources relies on a regular programming paradigm (such as Oriented Object Programming), depending on adjustments to deal with details inherent to the cloud provider and the issues related to scalability of regular programming paradigm. This paper addresses how Flow-Based Programming (FBP), a software architecture model based on Functional Programming, can be used as a solution to the challenges involving the achievement of distributed systems requirements. Firstly, we present a review of the concepts of FBP. We analyze Live Distributed Objects, Microsoft Orleans, and Yahoo! S4 under FBP perspective, providing a comparison among these solutions based on FBP criteria. Finally, we present an analysis of how FBP could be used to provide a better way to developers create scalable applications such as cloud computing.

1 INTRODUCTION

In the last decades, Object-Oriented Programming (OOP) has been widely employed in the development of several systems for diverse purposes. This predominance is probably justified by the fact that OOP solves (or at least gives a more elegant option for avoiding) numerous of issues identified with procedural programming, such as poor maintainability when dealing with complex systems and hardly reusable code. Indeed, core concepts of OOP, such as inheritance and encapsulation, are well known for helping developers to build complex systems that are easier to maintain and extend (Morrison, 1994). Despite such advantages, many authors have identified inherent flaws in the OOP paradigm (Armstrong, 1997; Ostrowski et al., 2009a). The most notorious are related to parallel processing and distributed systems, as well as situations in which a method's call leads to side-effects such as the modification of some objects' internal states or other changes that are not visible in the methods' return value. Most of these issues arise because simple events that are natural to OOP, such as variables changing during runtime and objects triggering other objects' methods, introducing considerable complexity when dealing with parallelism and distributed systems (Armstrong, 2007a). For these reasons, OOP may lose some of its interest in highly distributed environments such as Cloud Computing (Armstrong, 2007b), in which applications can run across a large collection of physically separated computers, allowing a lower expenditure on hardware structure while providing on-demand scalability and ensuring Quality of Service (QoS).

In a deeper analysis, any system approach that aims to be efficient in distributed environments needs to take into account two aspects: state dependencies and synchrony. The design of method invocation must be such that distributed events being triggered in different locations do not turn the system inconsistent, something difficult to achieve when there are numerous dependencies between objects. If the system remains synchronous, controlling such dependencies may be feasible, but as pointed out in (Morrison, 1994), unnecessary synchronism can be harmful to scalability. This issue is particularly serious when the time spent waiting for the response of request takes longer than the processing time itself: in such scenar-

R. Barros M., C. Miers C., Simplício M., C. M. B. Carvalho T., Mångs J., Melander B. and Souza V..
FLOW-BASED PROGRAMMING AS A SOLUTION FOR CLOUD COMPUTING REQUIREMENTS.
DOI: 10.5220/0003759004320440
In *Proceedings of the 1st International Conference on Operations Research and Enterprise Systems* (ICORES-2012), pages 432-440
ISBN: 978-989-8425-97-3

ios, synchrony clearly decreases the system's performance and flexibility.

The aforementioned issues can be efficiently addressed by means of the Flow Based Programming (FBP) (Morrison, 1994) software architecture model, a black-box processes assembling model that has the use of asynchrony and development of better ways to build scalable and maintainable systems as its central goals. One key aspect of this model is that, by employing Functional Programming characteristics, it precludes the existence of states or data structures modifiable in runtime, in such a manner that every function's output depends only on its input. Therefore, scalability is not limited by synchrony issues related to data state. These appealing features lead to the adoption of Functional Programming on systems demanding high levels of performance and parallelism, such as Amazon EC2 and Facebook.

This paper addresses the manner how FBP can be used for overcoming many of the most common challenges faced by distributed systems. More specifically, we show that FBP is a powerful tool for developing applications adapted to the context of Cloud Computing, since both have many goals in common. This study should allow developers to understand how the FBP model can help them to take advantage of cloud characteristics such as high availability and scalability.

Section 2 discusses the FBP model and the motivations behind its design. These concepts are necessary to provide the base knowledge of FBP in order to identify its relations with cloud computing. Section 3 relates three distributed systems implementations using FBP concepts, highlighting how the adoption of these concepts enabled solution for those systems. Section 4 generalizes the application of FBP concepts in cloud computing environment. Finally, we present the related works, considerations and future work.

2 FLOW-BASED PROGRAMMING DEFINITION AND MOTIVATION

FBP is a software architecture model based on black-box processes linked by dynamic connections. Structured objects, called Information Packages or IPs (Morrison, 1994), flow between processes (modules), configuring an asynchronous communication designed by 'send' and 'receive' commands. Those black-boxes can be designed to accept different IPs. Actually, even the structure of the received IP can be used as a criteria for performing some processing, resembling to Erlang's pattern matching (Armstrong, 2007b). Another key concept of FBP, and also found in functional languages such as Erlang, is the concern for side-effects avoidance, a major issue in distributed systems (Morrison, 2005). This model shares some similarities with the works usually referred to as "dataflow architectures", but some important differences exist especially because the latter is mathematically oriented, dealing with numeric values traveling through a network, while FBP works with messages carrying structured objects (Morrison, 1994).

It is important to highlight that FBP is not a programming paradigm, such as OOP and Functional Programming. FBP is a collection of many Functional Programing practices applied to generic software systems. A system organized with FBP principles does not need to be developed in a programming language that supports FBP, i.e., FBP model can encapsulate existing modules in a system. The FBP behavior is restrict to the external side of the encapsulation, so if a system uses a FBP to encapsulate large modules, it will be further from ideal FBP behavior. Therefore, developing systems with a programming language that supports FBP will provide the finnest granularity, i.e. a full FBP scenario.

Figure 1 exemplifies the above concepts, showing how structured objects containing the data to be processed flow between modules *A, B and C*. These streams of data do not possess any information about the internal functioning of the modules, differing from method invocation in OOP where control parameters can be sent. This absence of control data flowing between modules endows FBP with the *loose coupling* property, ensuring a reduced dependency between modules (Steinseifer, 2009).

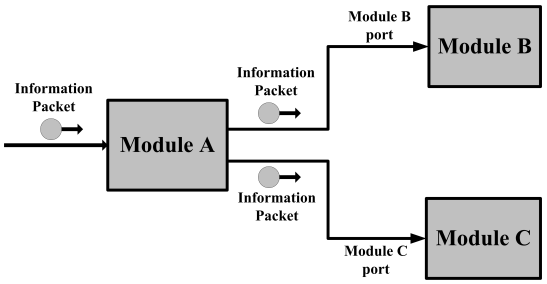

Figure 1: General outline of FBP.

The FBP model originated from the perception that synchronous communication, used by most program designers as the only way to obtain predictable behavior from the code, usually imposes unnecessary constraints upon the application, thus compromising its performance (Morrison, 1994). In fact, the only aspect required for producing predictable code is that

dependent operations must occur in the right order. When we build an application by connecting independent process, we can ensure the previous condition simply by making the correct connections between the chosen modules.

In this environment free of side-effects and unnecessary synchronism, program designers can build their code by assembling reusable modules and avoid most sources of complexity appearing in existing techniques for building distributed applications. In other words, programmers do not have to insert complex codes and synchronism to deal with recurrent unexpected behaviors, but can rather focus on what the current module should do. FBP can thus help programmers to build modules with different sizes, giving them freedom to choose the desired granularity of these modules.

3 RELATED IMPLEMENTATIONS

Even though FBP's main asset is to improve scalability and parallelism, its concepts can be used to develop wide range of systems. In the following, we describe three systems focused on such requirements – Live Distributed Objects (LDO) (Ostrowski, 2008), Microsoft Orleans (Bykov et al., 2010), and Yahoo S4 (Neumeyer et al., 2010) – and then we discuss how they use FBP principles in their conception. These solutions were chosen due their visibility, but other systems based on FBP exist.

3.1 Live Distributed Objects (LDO)

Live Distributed Objects (LDO) is a platform that aims to allow the development of distributed systems that can propagate decisions in a transparent manner (Ostrowski, 2008; Ostrowski et al., 2007). LDO relies on the concept of Live Objects, which are object replicas distributed among network components that may reside in different physical locations. These replicas coordinate themselves via multicast messages, thus maintaining a well defined and consistent state. The multicast layer that gives support to this communication is the *Quicksilver Scalable Multicast* (QSM) (Ostrowski et al., 2008), which is said to be highly optimized and to deliver improved scalability.

LDO implements an hierarchical structure of distributed protocols, using a set of mathematical operations for performing consistent transformations on data and to define atomic descriptions of protocols' states. It also employs algorithms that are responsible for locking methods and the election of leader nodes in this hierarchy. The resulting architecture is based on two main elements: the Membership Service (MS), which is basically a local coordination service; and the Delegation Authority (DA), a local service that works together with the MS and is responsible for keeping the leaders' information, thus defining a nested hierarchy.

The main motivation for using MS instead of centralized coordination is that the latter leads to a bottleneck, preventing the creation of scalable systems; meanwhile, MS can maintain local state information and repair the local data structures, informing local members and structure leaders nearby about its condition. Each Live Object, can only communicate with other architecture levels through the leader's information contained in its DA. The recursive architecture of LDO is exemplified in Figure 2, in which the arrows indicate communication between two connected structures (each level of the hierarchy has a pair DA/MS).

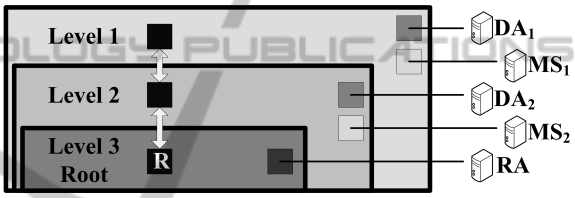

Figure 2: LDO hierarchical stack of structures. Adapted from (Ostrowski et al., 2009a).

Finally, it is interesting to notice that LDO's authors claim that the compiler should be the one responsible for dealing with physical aspects such as error handling, timeouts, network topology and construction of scalable structures (Ostrowski et al., 2009b). This concept highlights the notion that the development of network applications have many peculiarities, especially because, when dealing with distributed system, program designers still have to care about network details that long ago became transparent in other types of implementations.

3.2 Microsoft Orleans

Microsoft Orleans's a software framework for building client and cloud applications (Bykov et al., 2010). It aims to simplify programming distributed applications restricting concurrency.

Orleans is built around a unit of computation called *grain*, defined as an encapsulated sequential process with a private state that communicate exclusively by sending messages. The parallelism of the system is achieved via the production of multiple replicas (activations) of a grain. In this manner, despite being internally sequential, multiple messages

addressed to a specific grain can be processed simultaneously by two different activations. Activations can reside in different physical locations, enabling locality awareness in a manner that is completely transparent to the programmer.

Orleans message exchange is asynchronous and based on the *promise* concept. Basically, when an activation receives a message, it returns a promise that it will fulfill that request, and schedule the processing. When the processing is concluded, or a failure occurs, the promise is filled with a state, which can assume either "fulfilled" or "broken" value. Based on how the promise was filled, the requester can make decisions just like in traditional *try and catch* blocks.

This system based on promises is a solution for achieving asynchronous communication without losing control of the processing advancement, while allowing the information about failures during procedures to be gathered and creating the possibility of handling exceptions in a highly distributed environment.

Another aspect that reflects Orleans' concern with error handling is the adoption of a *LightWeight Optimistic Distributed Transactions* (LWODT) system. In LWODT, transactions are procedures that gather all activations included in a processing and ensure that the computation occurs with *non-strict* ACID (Atomicity, Consistency, Isolation and Durability) properties (Bykov et al., 2010), i.e., instead of using complex heuristic procedures to guarantee a consistent processing, Orleans detects when a processing become inconsistent and rolls back. The resulting system is optimistic because it takes advantage of the fact that most processes complete successfully. Thus, since a strict heuristic system would be unnecessary and inefficient in this context, Orleans can explore trade-offs between consistency and scalability. Figure 3 shows an overview of Orleans' message exchange and LWODT system.

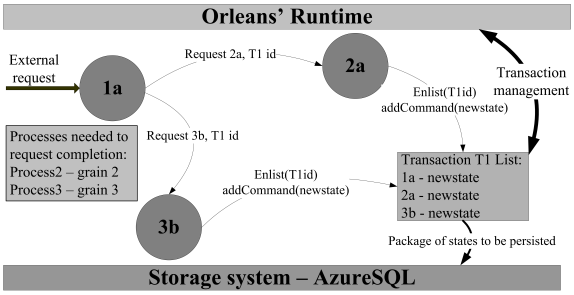

Figure 3: Overview of Orleans' message exchange (Bykov et al., 2010).

Figure 3 exemplifies an external request that reaches an activation – represented by a circle – *a* of *grain 1*, denoted as 1*a*. This external request involves processing data in grains 1, 2 and 3. In order to perform this processing in atomically, grain 1 perform two actions. Initially, it enlists itself in a empty transaction. Secondly, it sends an asynchronous message that contains a request for the activation of grains 2 and 3 and also the transaction reference. Using this reference, grains 2 and 3 enlist themselves to the transaction that becomes complete, containing all grains required to processing completion. If Orleans runtime detects no inconsistencies between the new states of grains, it sends a pack of data to be persisted using a AzureSQL storage system.

3.3 Yahoo! S4

The Simple Scalable Streaming System (S4) is a Distributed Stream Computing Platform developed by Yahoo!. It was conceived for solving problems relative to search applications that use data mining and machine learning algorithms such as Yahoo!, Bing and Google. Additionally, in order to render the most relevant ads in search pages, S4 was designed to be capable of processing thousands of queries per second (Neumeyer et al., 2010).

The computation units in the S4 platform are called Processing Elements (PEs), which are basically black-box processes with strictly private states. PEs are defined by the types of events it consumes, by when these elements should be consumed and by what kind of processing will be performed.

S4 considers that stream processing systems can be simplified by the absence of central or specialized nodes. For this reason, it adopts a symmetric approach, avoiding issues related to bottlenecks on central decision units. This approach allows the creation of large systems with high maintainability, which behave as pluggable and auto-configurable platforms.

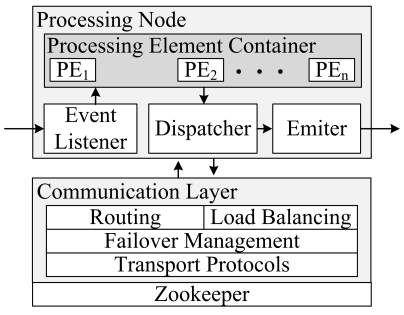

Figure 4: Yahoo! S4 architecture view.

Figure 4 shows the S4 architecture. This architecture is basically composed by Processing Nodes over a Communication Layer, thus hiding communication details from developers. Processing nodes receive the

uncontrolled data flow that passes through a chain of PEs. After that, one of the following must occur: either a message is sent requesting the activation of another processing node or results are published by the Emitter. Transport, failure treatment and load balance issues are all managed by the lower layer, abstracting these details from the programming model. Hence, programmers need to deal basically with the configuration of the PE chains' behavior.

It is important to highlight that S4 was designed for the context of processing user queries. In this context, the processing system has no control over the flow of data, leading to an architecture that degrades gracefully when discarding excessive data is necessary. In addition to this, S4 works with a lossy failover, according to which nodes are sent to a standby server after they fail, losing their memory state during this handoff. The regeneration process after failures can only be performed by the input stream, i.e. by data coming from in ports. This is not acceptable in some applications (Neumeyer et al., 2010), for which other mechanisms would be needed.

3.4 Comparisons and Comments

Despite their differences, the aforementioned solutions have many characteristics deeply related to FBP. An interesting issue that can be noticed almost immediately is that FBP treats the concept of *state* differently than Orleans and LDO do. Live Objects and Grains have well defined states, which are persisted after processing, while FBP modules work much more like functions, depending only on input data to define their behavior. This model endows FBP with the *loose coupling* property, which leads to a low dependency between linked modules. This different treatment, however, does not imply the absence of loose coupling property on the aforementioned works, since this concern is present in the definition of isolated computation units. Indeed, these units have private states that can only be indirectly accessed via asynchronous messages, which works similarly to FBP's black-box processes.

The definition of the computation units with private state shall be the first step of every distributed system design. Knowing the behavior of units and their communication capabilities allow developers to address their concerns about what type of control will be needed to maintain desired characteristics such as error handling, scalability and replication.

Regarding the modeling of the control system, LDO and S4 defend the adoption of a fully distributed control unit, eliminating the need of specialized nodes. Orleans works this issue in detail, noting that despite being a highly distributed environment, it needs a central unit to keep information for its distributed transactions; this central unit uses distributed and shared hash tables allowing grains' activations to coordinate themselves. In fact, as pointed in FBP, distribution is a natural consequence of eliminating unnecessary points of synchronism. To achieve a synchronous system, it is necessary to keep a lot of information about related computation units' states, leading to a central control unit that possess all this information. Thus, after designing isolated computation units, the goal must be the creation of a highly asynchronous system, which will eventually lead to distributed coordination.

The way that computation units are linked and organized is discussed in (Ostrowski et al., 2009b) and (Bykov et al., 2010). Both works adopt a hierarchical and nested architecture. Orleans' concept of *grain domains* can even be used to address security issues, considering that it is possible to restrict messages traffic based on relationships between grains. Grains are only capable of communicating with other grains that are on the same domain or with explicitly declared external grains. This grain reference concept is directly related to the port references adopted in FBP.

The three solutions presented, even if indirectly, aim to enable an easier and more intuitive experience when working with distributed systems. This goal is closely related to that of creating systems with high maintainability, as noticed by the careful treatment given to error handling on these works. Such concerns are a natural consequence of the increasing need for highly available systems, capable of providing Internet services in the same way as electricity services.

As aforementioned, the usage of FBP concepts does not allow only to achieve better distributed characteristics but also replication, error handling, and scalability. These characteristics are basic requirements for cloud computing systems. As a system closely related to Microsoft Windows Azure (Li, 2009), Orleans is already directly addressed to cloud computing environment (MSDN, 2011). S4 and LDO can also be related to cloud computing, since most of the issues addressed by their developers perfectly match important cloud computing requirements. Achieving a solid fault-tolerant system in a distributed environment is challenging due to the absence of a central control entity that can analyze a complete view of system's state and take the appropriate decision. Specifically, Orleans transaction system offers a very interesting solution to this issue, which is detailed in (Bykov et al., 2010). Table 1 summarizes the relations between the discussed solutions.

Table 1: Characteristics of implementations using FBP.

Solution	Main goal	Programming model	Processing unit	Architecture model	Communication organization	Error handling
LDO (Ostrowski, 2008)	scalability	assembly of existing objects	Live Object	Hierarchical and nested DA/MS pairs	QSM	local reboot by MS
Microsoft Orleans (Bykov et al., 2010)	scalability & mobility	grains specification	grain activation	Hierarchical and nested grains domains	asynchronous messages	LWODT
Yahoo! S4 (Neumeyer et al., 2010)	scalability	PEs specification	Processing Elements	Processing nodes over communication layer	uncontrolled flow through PEs	lossy failover

4 THE ROLE OF FBP IN A CLOUD COMPUTING

Cloud computing refers not only to the applications delivered as services over the Internet, but also to the platform/hardware and system software in the data centers that provide those services (Armbrust et al., 2009). The infrastructure that lies under this term brings advantages to end-users and developers, as well as to the cloud providers that maintains this infrastructure.

The main promise of Cloud Computing is to provide end-users with applications having an unforeseen availability. As a result, the access to Internet services for cloud users should achieve the status of an essential service, most like the access to electricity and sanitation. Meanwhile, developers with innovative ideas would no longer need to invest a large capital outlay for acquiring and operating the hardware required by their services.

Cloud providers could take advantage of the growth in the use of web services for providing low-cost infrastructure for developers, thus generating larger profits from their installations. However, many challenges are linked to this structure, including how applications must be developed and how the runtime must work to obtain the expected result. Basically, the main cloud requirements for deploying cloud applications are (Rimal et al., 2009; Gonzalez et al., 2011):

1. *Availability*: services should be available all time;
2. *Locality and replication*: availability and performance should not depend on users' locations;
3. *Maintainability*: developers must be allowed to perform major updates and fixes with low effort;
4. *Scalability*: Quality of Service (QoS) needs to be assured even under high demand;
5. *Security*: security-related issues such as data protection against unauthorized access need to be dealt with;
6. *Data loss avoidance*: data should be processed and persisted in a safe manner, preventing data losses.

It comes with no surprise that all these requirements have been considered in the works discussed on Section 3. What is notable, however, is that accomplishing these requirements can be seen as consequence of using FBP concepts.

Isolation is the main responsible for providing availability in FBP. FBP modules can be easily rebooted in case of failure, since the programming model is designed to avoid side-effects between modules, as discussed in depth in (Armstrong, 2007b); in addition, these characteristics greatly simplify debugging processes, an important aspect in highly available systems. Indeed, the model adopted by Microsoft's Orleans for consistency- and error-handling shows that the isolation of grains activations allowed the use of distributed transactions, a traditional concept that became difficult to implement using technologies such as OOP due to the complex interdependencies between objects.

FBP's isolated modules also support the development of simple replication techniques, as the consistency between multiple instances of a module precludes the need for complex control mechanisms (Ostrowski et al., 2009b; Bykov et al., 2010). The existence of consistent distributed instances of modules, together with location-aware mechanisms for content replication, allows the provision of content with the same QoS level for millions of users spread across the globe. The management of multiple module instances in a highly distributed environment is achieved by means of well defined *ports*, which not only allows the different instances to be referenced (Morrison, 1994), but can also be used for improving the system's location awareness. For example, a system having ports with well defined locations can keep information about the amount of data traveling between these points and, thus, it becomes natural to build algorithms to relocate or replicate high demanded modules for improving performance and ensuring QoS.

Regarding maintainability, the adoption of a distributed control model for the system can become a problem, since a possible modification in the system must be propagated through modules with no central communication point. On the other hand, FBP view of black-box modules linked by dynamic connections simplifies the maintenance task. A new version of any module can be developed and the only intervention to the system is updating the ports' addresses so they refer to the new version. For example, in Microsoft's Orleans model, this would be equivalent to modifying an activation ID written on the distributed hash table.

Scalability is one major concern in the context of cloud computing, and receives especial attention on all implementations discussed in section 3. Here, we discuss this requirement under two aspects. The first refers to the avoidance of unnecessary synchronism, along with the adoption of distributed control. The second is related to the system's granularity, which also impacts on system's performance.

Spreading an application across the globe using the cloud structure faces some essential challenges. Streams of data may take tens of milliseconds to be delivered, which is likely to introduce significant impacts on the performance of delay-sensitive applications (Ostrowski et al., 2007) and, similarly, impair processes based on a series of synchronous requests and responses. In this scenario, asynchrony assumes thus a major role, attenuating these effects by maximizing the system's throughput while minimizing the accumulation of transport delays. Coupled with asynchrony, the adoption of distributed control can reduce the traffic of control data, since structures are controlled locally (Morrison, 2003). At the same time, local coordination can simplifies eventual needs for synchronous control signals, avoiding the creation of new sources of delay. Therefore, the widely adopted "request & response" model for developing web applications, despite its simplicity, has inherent limitations.

Granularity refers to the size of the system's modules, i.e., the amount of data that defines a module state. For example, a system that processes large graphs, composed by thousands of nodes, can be designed in a variety of granularities: developers may adopt a fine granularity model, making each module store the state of one node, or a coarse granularity model in which hundreds of nodes are encapsulated in the same node (Bykov et al., 2010). However, the former approach could become a major concern when developing a distributed application. Still in the large graph example, if some process needs to access tens of the system's nodes, the fine granularity approach would lead to high overhead due to the need of assembling data from several different modules; in comparison, a coarse-grained distribution with adequate division among nodes would not face this problem.However, an application with another kind of data may lead to a different scenario. For example, we can imagine a social network in which a module is associated to a single visitor profile to a large group of them. In this case, the fine granularity approach is likely to fit better, since a coarse-grained distribution would result in internal searches when trying to locate the data from some specific profile, and also lead to unnecessary replication of profiles if there is a high demand for a single user's data.

Hence, we can analyze granularity from two different perspectives. First, granularity can be seen as a trade-off between maintainability and performance (Morrison, 1994). This vision is based on the fact that, if a sequential process is divided, it will add API calls for passing data through the slices, delaying the whole process. On the other hand, smaller modules increase maintainability, allowing a more focused work on specific parts of the process, and can improve modules' reuse. Second, considering a parallel processing context, granularity can be described as a trade-off between the level of parallelism and state locality needed for efficient computations (Bykov et al., 2010). This statement can be better understood considering that, when a fine-grained distribution is adopted, a higher level of parallelism is achieved due to the possibility of using more specialized processes; in the case of Microsoft's Orleans, for example, this concept is related to the ability to create new activations that supply the need for some specific piece of information.

Unlike the other requirements, security is not considered in depth by the implementations discussed in section 3. This also applies to the security-related aspect of data loss avoidance, albeit Microsoft's Orleans developers claims that, with the addition of a distributed storage system, this solution is adequate for highly secure and efficient persistence of data. Even though the relationship between FBP concepts and security in cloud computing remains a subject for further study, concepts such as isolation and the capability to create subsystems allows the establishment of boundaries to flows of data, thus facilitating the application of security principles (Mather et al., 2009; Kaufman, 2009; CSA, 2011).

4.1 FBP and MapReduce

MapReduce is a programming model, as well as an associated implementation for processing and generating large data sets. It is considered the standard

solution for processing large amounts of data distributed across thousands of machines and can thus be seen as a native cloud computing solution. As FBP, MapReduce was inspired on functional programming, more specifically on *map* and *reduce* functions that are present on Lisp and many other functional languages (Dean and Ghemawat, 2008).

As a consequence of its functional behaviour, MapReduce achieves many of the characteristics aforementioned for FBP, such as side-effects avoidance and high scalability. Even though these two models may look as suitable alternatives for developers when designing a distributed system, we believe that they can actually be used together as powerful tools to achieve maintainable code. Basically, MapReduce proposes the use of two independent sets of workers. The first group consumes the data segments tagging and organizing them by similar terms based on some comparison criteria, then the second set of workers consumes those similar terms and produces an unified result containing information about that specific group (e.g., the number of terms in this group). This apparently simple approach fulfills some cloud computing requirements such as fault tolerance and scalability: in case of failure, a worker can be reset and its task can be rescheduled and relocated to others workers; furthermore, more workers can be spawned rapidly to increase processing capabilities.

In essence, both FBP and MapReduce are similar in terms of data treatment. Both models define private states for improved data consistency, and consider that asynchronous communication is a natural way to achieve scalability. While the FBP model works on modules' isolation and on the need of asynchronous communication, discussing in depth how this communication can be organized, MapReduce proposes a system for splitting, processing and reassembling streams of data in a consistent and scalable manner, using isolation and asynchronous communication as simple tools. Thus, concepts adopted by the FBP model – such as the definition of ports, asynchronous data flow between modules and a distributed control unit – work very well MapReduce main idea. Indeed, both approaches describe scalable communication, distributed processing, practices for consistency of data and locality awareness through ports definition.

In conclusion, FBP can be seen as a collection of programming practices necessary for enabling the use of efficient distributed processing models such as MapReduce.

5 RELATED WORK

As aforementioned, there are works closely related to FBP written under the term 'dataflow architectures'. The difference between the two terms can be more easily understood through the LDO example. LDO is considered a dataflow architecture work, since its main objective is to establish algorithms for data consistency and atomicity; for this reason, the major part of Ostrowski's work is focused on the mathematical treatment of data. In contrast, FBP focus on setting a generic programming model, which can take advantage of several data consistency algorithms.

In (Steinseifer, 2009), Steinseifer discusses FBP in detail, clarifying the main concepts and providing performance information about JavaFBP implementation. Moreover, Larus (Gupta and Larus, 2010) highlight some motivations for the adoption of Cloud Computing, highlighting some of its main requirements and discussing prevailing challenges like concurrency, parallelism, application partitioning, high availability and distribution. Nonetheless, to the best of our knowledge, the advantages of adopting FBP model in the context cloud computing for overcoming inherent limitations of other programming paradigms have not been previously explored in the literature.

6 CONSIDERATIONS AND FUTURE WORK

Albeit the research on Cloud Computing is still in its infancy, many crucial and challenging requirements that must be accomplished by such solutions have already been defined in the literature. In this context, FBP presents itself as a powerful tool for developing systems and services capable of meeting these challenges, allowing simple solutions for distributed systems' classic problems and overcoming several limitations of more conventional programming paradigms. This comes from the fact that FPB relies on a set of concepts that aim at improved scalability and avoidance of side-effects, which are essential in highly distributed systems. This conclusion is supported by the study of platforms especially focused on cloud applications, such as LDO, Microsoft's Orleans and Yahoo!'s S4 solutions.

It is notable that FBP concepts can be implemented in a variety of ways, being able even to encapsutate others programming paradigms, organizing an heterogen scenarios in a solid model. We argue that FBP as a solution to distributed systems requirements, due to its generality and absence of inconsistencies while dealing with parallel processing.

We believe that a deeper study is needed in order to address FBP security concerns in a cloud computing environment and how FBP can assist in solving problems related to distributed coordination and asynchrony. Distributed environments with asynchronous message passing shows a completely different scenario to developers, while some security vulnerabilities will not be applicable anymore, several others shall arise.

A deeper study of how granularity affects the FBP under a distributed scenario is needed in order to support applications' design. As exposed in Section 4, granularity control interferes deeply on systems' performance. It is necessary to clarify where and when a certain granularity level should be adopted, enabling a more accurate distributed applications design.

ACKNOWLEDGEMENTS

This work was supported by the Research and Development Centre, Ericsson Telecomunicações S.A., Brazil.

REFERENCES

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. (2009). Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley.

Armstrong, J. (1997). The development of erlang. In *Proc. 2nd ACM SIGPLAN ICFP*, ICFP '97, pages 196–203, NY, USA. ACM.

Armstrong, J. (2007a). A history of erlang. In *Proc. 3rd ACM SIGPLAN HOPL*, HOPL III, pages 6–1–6–26, New York, NY, USA. ACM.

Armstrong, J. (2007b). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 1 edition.

Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., and Thelin, J. (2010). Orleans: A framework for cloud computing.

CSA (2011). Security guidance for critical areas focus in cloud computing.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. In *OSDI '04*, pages 137–150.

Gonzalez, N., Miers, C., Redígolo, F., Simplício, M., Carvalho, T., Naslund, M., and Pourzandi, M. (2011). A taxonomy model for cloud computing services. In *Proceedings of CLOSER 2011*, Noordwijkerhout, The Netherlands. INSTICC.

Gupta, R. and Larus, J. (2010). Programming clouds. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg.

Kaufman, L. M. (2009). Data security in the world of cloud computing. *IEEE Security and Privacy*, 7(4):61–64.

Li, H. (2009). *Introducing Windows Azure*. Apress, Berkely, CA, USA.

Mather, T., Kumaraswamy, S., and Latif, S. (2009). *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance*. O'Reilly Media, 1 edition.

Morrison, J. P. (1994). *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1st edition.

Morrison, J. P. (2003). Asynchronous component-based programming.

Morrison, J. P. (2005). Patterns in flow-based programming.

MSDN (2011). Orleans: A framework for scalable Client+Cloud computing.

Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Data Mining Workshops, International Conference on*, pages 170–177, CA, USA. IEEE Computer Society.

Ostrowski, K., Birman, K., and Dolev, D. (2007). Live distributed objects: Enabling the active web. *IEEE IC*, 11:72–78.

Ostrowski, K., Birman, K., and Dolev, D. (2008). Quicksilver scalable multicast (QSM). In *Network Computing and Applications*, pages 9–18, CA, USA. IEEE CS.

Ostrowski, K., Birman, K., and Dolev, D. (2009a). Distributed data flow language for multi-party protocols. In *Proc. 5th Workshop on Programming Languages and Operating Systems*, PLOS '09, page 7:1–7:5, New York, NY, USA. ACM.

Ostrowski, K., Birman, K., Dolev, D., and Sakoda, C. (2009b). Implementing reliable event streams in large systems via distributed data flows and recursive delegation. In *Proc. 3rd ACM - DEBS*, page 15:1–15:14, New York, NY, USA. ACM.

Ostrowski, K. J. (2008). *Live distributed objects*. PhD thesis, Cornell University, Ithaca, NY, USA.

Rimal, B. P., Choi, E., and Lumb, I. (2009). A taxonomy and survey of cloud computing systems. In *Networked Computing and Advanced Information Management, International Conference on*, pages 44–51, Los Alamitos, CA, USA. IEEE Computer Society.

Steinseifer, S. (2009). *Evaluation and Extension of an Implementation of Flow-Based Programming*. PhD thesis, Fachhochschule Gieÿen-Friedberg.