Comparing Distributed Online Stream Processing Systems Considering Fault Tolerance Issues

André Leon Sampaio Gradvohl
School of Technology / University of Campinas, Brazil
Email: gradvohl@ft.unicamp.br

Hermes Senger
Department of Computer Science / Federal University of São Carlos, Brazil
Email: hermes@dc.ufscar.br

Luciana Arantes and Pierre Sens
Laboratoire d'Informatique de Paris 6 / Université Pierre et Marie Curie, France
Email: {luciana.arantes, pierre.sens}@lip6.fr

Abstract **—This paper presents an analysis of four online stream processing systems (MillWheel, S4, Spark Streaming and Storm) regarding the strategies they use for fault tolerance. We use this sort of system for processing of data streams that can come from different sources such as web sites, sensors, mobile phones or any set of devices that provide real-time high-speed data. Typically, these systems are concerned more with the throughput in data processing than on fault tolerance. However, depending on the type of application, we should consider fault tolerance as an important a feature. The work describes some of the main strategies for fault tolerance – replication components, upstream backup, checkpoint and recovery – and shows how each of the four systems uses these strategies. In the end, the paper discusses the advantages and disadvantages of the combination of the strategies for fault tolerance in these systems.**

Index Terms—**Fault-tolerance, Distributed systems, System applications, Online stream processing.**

I. INTRODUCTION

Online stream processing or event stream processing (ESP) is an initiative that is growing in recent years with social networks and other applications that require data processing in real time. Examples of such applications are network attack detection, financial analysis, spam filtering, targeted advertising, trend analysis, participatory sensing and even disaster management situations, among others [1] [2] [3]. The main feature of an application that uses event stream processing is to deal with a continuous flow of incoming data, also called events, as quickly as possible reducing this huge input data volume, for decision-making or storage of what is relevant.

In few words, an ESP system receives a data stream as input, perform some computations or transformations and produces an output. The input data rate is not under control of the system, which generally receives it at a high throughput. The amount of data coming from the input stream is usually so huge that it is impractical to store on disks. The events or input stream may come from different sources such as websites, mobile phones or specific sensors. Keeping this system on a single server is risky and may not have enough computing power to meet system's demand for processing [4]. Therefore, a distributed system will increase the processing power and the availability.

On the other hand, stream-processing applications usually runs for a long time and shall keep their state over extended periods. Thus, when running this sort of application, we expect that it might encounter problems such as failures, infrastructure updates, scheduled restarts and application updates

According to [5], buffer overflow and node failures are the two most challenging faults in distributed ESP systems. A buffer overflow happens when a node cannot allocate enough memory to buffer and incoming event. In turn, a node failure occurs when there are hardware faults on the platform that supports a processing node.

Therefore, to achieve its objectives, it is necessary to build a distributed, fault tolerant, persistent state and scalable system. The fault tolerance feature is important to keep the system working as long as possible, since losing a portion of the data stream can lead to inaccurate decisions. Maintain persistent states of the system is also required for quickly restart the system after a fault situation. In other hand, scalability is a feature that allows the growth of the system, adding or removing components for processing without impact on its performance.

The first ESP systems, created in the early 2000s, Aurora, Borealis, STREAM, TelegraphCQ, NiagaraCQ and Cougar [6]. At that time, they were centralized systems, i.e. they run on a single server, and aimed to overcome the problems to deal with stream processing by traditional databases.

It is important to mention that objectives of the databases are essentially different from ESPs. While databases are optimized for the efficient storage and querying of data, ESPs are designed to provide high performance analysis of streams with low latency. Thus, the requirements for fault tolerance and scalability differs substantially in both kind of systems.

A. Key Concepts in Event Stream Processing Systems

In ESP systems there are some common concepts described as follows. First, stream processing refers to a programming paradigm for processing continuous data streams, i.e. an infinite sequence of data items, through a topology (stream graph). Such topology is a directed acyclic graph where edges are streams and vertexes are operators, as depicted in Fig. 1. The operators do computations or transformations in input streams to produce new streams for other operators or for output [7].

A data item or an event consists of a key-value pair, which forms a tuple. A key identifies the tuple and the value is a sequence of bytes associated with a particular key. Notice that some operators need to maintain their state, which means that the operator should keep information while it processes different tuples.

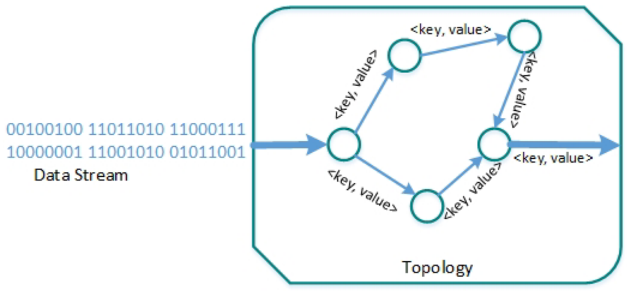

Figure 1. Generic schematic model for ESP systems.

As soon as the operators receives the tuples and just before their processing, tuples are temporarily stored in the input queues (one for each input stream). Likewise, the output queues hold the tuples generated by each operator, just before other operators receives them.

Those concepts are common to most distributed ESP systems. In the remainder of this paper, we describe some selected ESP systems regarding fault tolerance issues. Section II describes the main fault tolerance models within distributed ESP systems, along with the some fault tolerance strategies in distributed ESP systems. Section III depicts four ESP systems selected for this work. Section IV discusses the differences in the advantages and disadvantages ESPs systems regarding fault tolerance features in those systems. Finally, Section V presents some final remarks.

II. Fault Tolerance Models

The way in which a computer system can gracefully degrades in a presence of failures of some of its components is what we call a fault tolerance model. Thus, fault tolerant techniques used will depend on the tolerance model predicted for specific computing systems.

According to [8], there are two main fault models, Byzantine faults and fail-stop faults, but also a third model called fail-stutter faults.

In Byzantine fault model, failed nodes are still alive and interacting with the rest of the system. However, their output is corrupted, although failed nodes still produce valid messages. According to [9] [10], this means that other nodes in the system cannot detect the failures, neither the failed nodes.

On the other hand, in fail-stop fault model, when failures happen, a node stops producing outputs and interacting with the system. This a sort of fault is easy to detect and overcome.

The third model, called fail-stutter model, is based on an extension of the fail-stop fault model. In this case, it also includes performance faults, which occurs when a node of the system performs much worse than other ones.

A. Fault Tolerance Strategies in Distributed ESP Systems

Next, will describe some strategies to improve fault tolerance in distributed ESP systems.

1) Replication of components

This strategy, according to [11] [12], duplicates the components of systems to minimize damages is case of failures. In active replication, duplicated components are alive and receive the same input. This strategy allows the comparison of outputs and thus can identify unexpected behavior of a component. Furthermore, in the case of a fail-stop failure, backup components can take over the role of the primary component, preventing the system from collapsing. The problem with this strategy is the investment since the strategy replicates components of the system and therefore involves, at least, duplicating costs.

In passive replication, on the other hand, there are components in stand-by, which assume the role of the corrupted component only in case of failures. In this situation, the whole input must be resubmitted to reach the normal state of the system. Furthermore, it must consider the elapsed time until the normal state is established. However, this delay is not always tolerable, especially on stream processing systems.

2) Checkpoint

To implement checkpoints, we model the whole system as a series of fault-free states. Upon reaching each of these states, we do a checkpoint. In addition, a checkpoint is consistent if all events that occurred up to that checkpoint came from a previous consistent checkpoint.

To keep these checkpoints, the system must have a stable storage, i.e. a storage device that allows system state saving and recovery in case of failures, recovering from the fault-free states recorded previously. The recovering process is based on the most recent set of consistent checkpoints.

There are two classes of recovery protocols based on checkpoints: uncoordinated (asynchronous) or coordinated protocols [13]. In uncoordinated protocol, each process decides when making their own checkpoints.

However, this type of protocol is very risky because it does not guarantee a global system consistency. Additionally, it can generate the so-called domino effect, when the recovery of a process will require the recovery of another one, thus continuing indefinitely.

The coordinate protocol, on the other hand, requires that the processes organize their checkpoints in order to ensure global system consistency. However, this is a more complex protocol for creating checkpoints as it requires an exchange of messages between all processes, making it a relatively time consuming task.

3) Upstream backup

In upstream backup, upstream nodes (e. g. node *u* in the graph depicted in Fig. 2) logs the tuples in their output queues for their downstream neighbors (e. g. node *v* in Fig. 2) until the downstream neighbors finish processing these tuples [14]. In case of failure of a downstream node, upstream nodes send the tuples again.

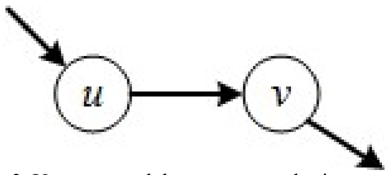

Figure 2. Upstream and downstream nodes in a topology.

This approach has some drawbacks, however. For instance, the queue length may not be large enough to handle all the tuples stored between failure and recovery times. Additionally, it is important to mention that upstream backup takes a long time to recover, because the system should wait a new node to rebuild serially the failed node's state by sending its data again.

4) Recovery

There are three types of recovery according to [14]. On precise recovery, it can completely mask a failure by sending all the tuples that that downstream nodes did not receive. On the other hand, on rollback recovery strategy, the output produced after a failure is not necessarily the same as the output of an execution without failure, since on this type of recovery some duplicate tuples may be sent again. Finally, on gap recovery it drops off some old data to reduce recovery time and runtime overheads.

The recovery method chosen determines the latency of the recovery operation. Therefore, the application should consider which method uses to reduce the latency of recovery to acceptable levels [15].

III. Event Stream Processing Systems

We selected four distributed ESP systems to analyze how they treat fault tolerance issues. The criterion for selection was strictly the usage, i.e., we choose systems that are currently in production on enterprises that deal with event stream processing. Among the selected are the following: MillWheel [16], S4 [17] [18], Spark Streaming [19] and Storm [20] that are described in next sections.

A. MillWheel

MillWheel is widely used at Google. It is a framework, which implements a programming model for streaming and low-latency systems [16]. MillWheel allows users specify a topology, where tuples (triples) are composed of key, value and timestamp data. Unlike other ESP systems, MillWheel includes a timestamp in tuples, which is a time reference generated by the event which outputs that tuple.

Based on the input stream, each node in the topology does a computation and outputs tuples to another node. Thus, as soon as data arrives in a node, it invokes these computations or transformations. The computations may occur in parallel, which makes them faster, although more susceptible to system faults. One of the Millwheel features is that users can add or remove computations dynamically, i.e., they can add or remove nodes in the topology during its execution, without having to restart the whole system.

1) Fault tolerance strategy on MillWheel

MillWheel provides persistent states of operators by a replicated high available data storage systems, such as BigTable (a distributed storage system) or Spanner (a distributed database). These systems ensure data integrity and are transparent to users.

MillWheel also deals with fault tolerance at the framework level, ensuring that each operator receives its tuple. For this, it is necessary that the application use the abstractions of state and communication provided by the framework itself.

Therefore, upon receiving a tuple, the operator checks if it is duplicate or not. If it is duplicate, then it discards the tuple; otherwise, it processes the tuple according to the user code. After that, the state of that node is committed into a storage system and the sender node receives acknowledge.

B. S4

In S4, a stream is a set of elements composed of tuples of key and attributes (or values) and we call these tuples as events. Yahoo! proposed the S4 system architecture as a set of processing elements (PE) that consumes or process exactly those tuples identified by a particular key. There is a special type of PE, called keyless PE, that consumes all events associated with a tuple, regardless their key. The association between the outputs of some PEs and the entry of other PEs also forms a dataflow graph or a topology [18].

The PEs are logically hosted in Processing Nodes (PN), which are responsible for listening to events, perform operations when an event arrives and dispatch new events for other nodes in the topology.

1) Fault tolerance strategy on S4

ZooKeeper [21] [22] is the subsystem responsible for detecting failures in S4 nodes. A fixed number of tasks or partitions defines a S4 clusters. Usually, we define more nodes than partitions and hence we got stand-by nodes.

Upon detecting a node failure, the ZooKeeper notifies the stand-by nodes. These stand-by nodes compete for the partition and, once determined the assigned node other nodes are notified that the task was assigned. Then the new active node sends a message the other active nodes requesting to forward messages to it.

ZooKeeper considers that a node has failed when there is no response from him after a timeout. The node itself defines this time when connects to ZooKeeper and is at least twice the heartbeat signal frequency specified by the ZooKeeper configuration.

When the cluster brings a new node, it has no state. Thus, if there has been a previous checkpoint, the user code should retrieve the state when messages arrive to that node and when the PEs are instantiated. Checkpointing is uncoordinated and asynchronous to minimize latency and reduce state loss. In addition, recovery is lazy, which means that occurs only when necessary.

C. Spark Streaming

Despite other ESP systems, Spark Streaming works on a series of deterministic tasks (batch computations) on small time intervals [19]. For this work, the Spark Streaming system defines a data structure called Resilient Distributed Dataset (RDD), which holds the data in memory and can retrieve it without replication by analyzing the lineage graph operations used to build it.

The RDD is an immutable, deterministically re-computable, distributed and fault tolerant dataset. More specifically, an RDD is a read-only collection of records partitioned and replicated across the worker nodes in a cluster. These worker nodes are processes that can store and process RDDs in memory.

The Spark Streaming architecture comprises a master node, which schedules tasks to compute new RDD partitions; worker nodes to get the data, stores the input data and perform the tasks; and the client library, used to send data into the system.

1) Fault tolerance strategy on Spark Streaming

Spark Streaming provides two types of strategies for fault tolerance: replication and upstream backup. Spark Streaming handles fault tolerance by periodically writing metadata information into a Hadoop Distributed File System (HDFS) [23] directory. Therefore, when a worker node fails, another worker node can restart and continue processing from the last checkpoint. The fact that RDD models all data assures it. Thus, any computation will lead to the same result.

On the other hand, for online data sources, received input data should be replicated in memory between nodes of the cluster. Then, if a worker node fails, the other worker node can still process the input data.

D. Storm

The Storm system, created by Twitter, also works with the concept of topology as MillWheel and S4. In a Storm topology, there are spouts and bolts, which forms a topology. A spout is a source of a stream and a bolt does a computation or transformation. A bolt receives a stream and may even produce an output stream.

Three kind of nodes comprises a Storm cluster: a master node, called Nimbus; a coordinator node, managed by ZooKeeper; and one or more workers embedded in nodes called supervisors. The Nimbus is responsible for distributing code around the cluster, assign tasks to machines, and faults monitoring. A worker node, in turn, processes the work assigned by the master node; and ZooKeeper coordinates the relation between Nimbus and worker nodes.

1) Fault tolerance strategy on Storm

In Storm, the spouts – sources of streams – keep the messages at their output queues until the bolts acknowledge them. If there is an acknowledgement, the spout drops off the message from the queue. Otherwise, the spout sends the messages again.

To handle the node failures, Nimbus – the master node – listens for worker nodes heartbeats, which are sent periodically. If the heartbeats do not come, Nimbus assumes that the node is down and move the workers to another node.

Nimbus and supervisor nodes are stateless and they destruct themselves when and unexpected situation occurs. Therefore, if an abnormal situation occurs with those nodes, they restart themselves transparently to the worker processes. However, other machines could not host worker processes if they fail and Nimbus is dead.

IV. Discussion

Tab. 1 summarizes the differences between the selected ESP systems regarding the supported programming languages, the strategies used for fault tolerance and the subsystem responsible for dealing with failures.

Both MillWheel and Storm work with virtually any programming language, which means that, with less effort, programmers can adapt their legacy software to work with these ESPs. S4 and Spark Streaming, in turn, work on Java, despite the fact that Spark Streaming also supports computations programming in Scala and Python.

In addition, MillWheel and Spark Streaming do not depend on third-part subsystems to address the fault tolerance. In this aspect, they both deal with fault tolerance issues within the framework itself. S4 and Storm, in turn, are strongly dependent on ZooKeeper to tolerate failures; and Storm is even dependent on Nimbus. In both cases – S4 and Storm – user should manage the operator state, which is responsible to specify how and when it should recover those states.

V. Conclusions

TABLE I.
ESP SYSTEMS COMPARED.

ESP System	Programming languages supported	Fault tolerance strategy	Subsystem to handle failures
MillWheel	Virtually any programming language	Uncoordinated periodic checkpoint; Upstream backup	None, the system handles failures itself.
S4	Java	Uncoordinated periodic checkpoint;	ZooKeeper.
Spark Streaming	Java, Scala and Python	Coordinated periodic checkpoint; Replication; Parallel recovery	None, the system handles failures itself.
Storm	Virtually any programming language	Upstream backup; No checkpoints	Nimbus, ZooKeeper.

Applications that uses ESP systems will determine the type of fault that the system should support. In some applications, where the state of the operator is not vital to the operations on stream graph, the system can tolerate a failure of an operator (a vertex in the stream graph) simply instantiating the failed operator. In this case, systems that deal with stateless and replicated operators, e.g. S4 and Storm, can be useful.

Furthermore, applications where the state of each operator should be kept, robust mechanisms for fault tolerance and maintenance of states should be strong enough to ensure that the application remains integrate and coherent. In this case, MillWheel and Spark Streaming have interesting features.

However, we are particularly interested in applications of participatory sense in emergencies, i.e., given an emergency such as an earthquake or collapse of a building, victims who are able to use their mobile devices could ask for help, communicate with emergency teams or even informing the situation around them, all through social networks like Twitter and Facebook.

This communication could make the most effective displacement of emergency teams, directing them to the most critical *locus*. In this case, we would need an ESP system that maintains the state of its operators but at the same time, could keep the high throughput of data processing.

Therefore, a system that combines fast failure detection with operator state maintenance even in the event of failure would be ideal.

ACKNOWLEDGEMENT

The first author gratefully acknowledges Coordination for the Improvement of Higher Education Personnel (CAPES), Brazil, project number 11785/13-6, for the financial support to this project and the Laboratoire d'Informatique de Paris 6 (LIP6) for hosting him during his postdoctoral studies.

REFERENCES

[1] N. R. Adam, J. Eledath, S. Mehrotra e N. Venkatasubramanian, “Social media alert and response to threats to citizens (SMART-C),” in 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012.

[2] N. R. Adam, B. Shafiq e R. Staffin, “Spatial Computing and Social Media in the Context of Disaster Management,” IEEE Intelligent Systems, vol. 27, nº 6, pp. 90-96, Nov 2012.

[3] M. K. Boulos, B. Resch, D. N. Crowley, J. G. Breslin, G. Sohn, R. Burtner, W. A. Pike, E. Jezierski e K.-Y. S. Chuang, “Crowdsourcing, citizen sensing and sensor web technologies for public and environmental health surveillance and crisis management: trends, OGC standards and application examples,” International Journal of Health Geographics, vol. 10, nº 1, p. 67, 2011.

[4] A. Martin, C. Fetzer e A. Brito, “Active Replication at (Almost) No Cost,” in 30th IEEE Symposium on Reliable Distributed Systems (SRDS), 2011.

[5] W. Hummer, C. Inzinger, P. Leitner, B. Satzger e S. Dustdar, “Deriving a Unified Fault Taxonomy for Event-based Systems,” in Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, New York, NY, USA, 2012.

[6] V. M. Gulisano, “StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine,” PhD. Thesis. Faculdad de Informática, Universidad Politécnica de Madrid, 2012.

[7] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soule e K.-L. Wu, “IBM Streams Processing Language: Analyzing Big Data in motion,” IBM Journal of Research and Development, vol. 57, nº 3/4, pp. 7:1-7:11, 2013.

[8] M. Treaster, “A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems,” ACM Computing Research Repository (CoRR), vol. 501002, pp. 1-11, 2005.

[9] P. Costa, M. Pasin, A. N. Bessani e M. Correia, “Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes,” in IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), 2011.

[10] P. Costa, M. Pasin, A. N. Bessani e M. Correia, “On the Performance of Byzantine Fault-Tolerant MapReduce,” IEEE Transactions on Dependable and Secure Computing, vol. 10, nº 5, pp. 301-313, 2013.

[11] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki e P. Pietzuch, “Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management,” in SIGMOD International Conference on Management of Data, New York, 2013.

[12] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki e P. Pietzuch, “Scalable and Fault-tolerant Stateful Stream Processing,” in OpenAccess Series in Informatics, 2013.

[13] D. Goswami e S. Sahu, “An Efficient Protocol for Checkpoint-Based Failure Recovery in Distributed Systems,” Distributed Computing and Internet Technology, pp. 135-144, 1, 2005.

[14] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker e Z. Stan, “High-availability algorithms for distributed stream processing,” in International Conference on Data Engineering, 2005.

[15] A. Brito, S. Weigert, M. Subkraut, C. Fetzer e P. Felber, “Handling Crash and Software Faults Efficiently in Distributed Event Stream Processing,” in 2010 Third International Conference on Dependability (DEPEND), 2010.

[16] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom e S. Whittle, “MillWheel: Fault-Tolerant Stream Processing at Internet Scale,” in Very Large Data Bases, 2013.

[17] L. Neumeyer, B. Robbins, A. Nair e A. Kesari., “S4: Distributed Stream Computing Platform,” in IEEE International Conference on Data Mining Workshops (ICDMW), 2010.

[18] J. Chauhan, S. A. Chowdhury e D. Makaroff, “Performance Evaluation of Yahoo! S4: A First Look,” in Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 2012.

[19] M. Zaharia, T. Das, H. Li, S. Shenker e I. Stoica, “Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters,” in 4th USENIX conference on Hot Topics in Cloud Computing, Farminton, Pennsylvania, 2013.

[20] T. Chardonnens, P. Cudre-Mauroux, M. Grund e B. Perroud, “Big data analytics on high Velocity streams: A case study,” in IEEE International Conference on Big Data, 2013.

[21] P. Hunt, M. Konar, F. P. Junqueira e B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2010.

[22] S. Skeirik, R. B. Bobba e J. Meseguer, “Formal Analysis of Fault-tolerant Group Key Management Using ZooKeeper,” in 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2013.

[23] K. Shvachko, K. Hairong, S. Radia e R. Chansler, “The Hadoop Distributed File System,” in IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010.

André Leon S. Gradvohl received the bachelor’s degree in Computer Science from Federal University of Ceará, Brazil in 1997; master in sciences degree from Technological Institute of Aeronautics, Brazil in 2000; and the Ph.D. in Electric Engineering from University of Campinas, Brazil in 2005. He has been an assistant professor in computer science at the School of Technology at University of Campinas, Brazil since 2010. Currently, he is doing post-doctoral studies in distributed online stream processing systems at Laboratoire d'Informatique de Paris 6 (LIP6), France. The Coordination for the Improvement of Higher Education Personnel (CAPES) in Brazil currently sponsors his researches.

Hermes Senger obtained the BS degree in Computer Science from the State University of São Paulo (UNESP), Brazil, in 1989. He obtained both the MS and Ph.D. degrees in Electrical Engineering from the University of São Paulo (USP), Brazil, in 1996 and 2002, respectively. In 2009 he joined the Federal University of São Carlos (UFSCar), Brazil, where he teaches Computer Science at the undergraduate, masters, and doctorate levels. He has published more than 40 papers in international and national journals and conferences with selective editorial policy. His research interests include high performance computing, parallel and distributed computing systems and applications, resource scheduling, and scalability analysis. He is a member of the Brazilian Computer Society (SBC), IEEE, and ACM.

Luciana Arantes received her Ph.D. degree from the University Pierre et Marie Curie (Paris 6), Paris, France in 2000 and her masters degree from the Polytechnic Scholl of University of São Paulo, Brazil, in 1996. She got the undergraduate degree from the University of Campinas, Brazil, in 1986. Since 2001, she is associated professor at Université Pierre et Marie Curie (UPMC) and member of Regal project-team, a cooperation between the Laboratoire d'Informatique de Paris 6 (LIP6) and the French Research Center INRIA. Her research focuses on adapting distributed algorithms to large-scale, heterogeneous, dynamic, and self-organizing environments, such as grid, peer-to peer systems, cloud computing or mobile networks. She has published 7 articles in journals, 55 articles in international conferences, 13 articles in French conferences and 2 book chapters. Dr. Luciana Arantes has been member of some conference committees (ICPADS, LADC, GCP) and reviewer of some journals (JPDS, Computer Journal, FGCS).

Pierre Sens received his Ph. D. in Computer Science in 1994, and the “Habilitation à diriger des recherché” in 2000 from Paris 6 University, France. Currently, he is a full Professor at Université Pierre et Marie Curie. His research interests include distributed systems and algorithms, peer-to-peer file systems, fault tolerance, grid and cloud computing. Pierre Sens is heading the Regal group which is a joint research team between LIP6 and INRIA. He was member of the Program Committee of 25 conferences (ICDCS, IPDPS, OPODIS, Europar, SSS…). Overall, he has published over 100 papers in journals (JPDC, PPL, JOS, SPE..) and conferences (DSN, SRDS, EDCC, ICPP, OPODIS, EuroPar,…) .