Distributed Exception Handling: Ideas, Lessons and Issues with Recent Exception Handling Systems

Aurélien Campéas[1], Christophe Dony[1], Christelle Urtado[2], and Sylvain Vauttier[2]

[1] LIRMM, 161 rue ADA, 34392 Montpellier
[2] LGI2P, Ecole des Mines d'Alès, site EERIE, Parc scientifique G. Besse, 30035 Nîmes

Abstract. Exception handling is an important feature of the tool-set that enables the building of fault-tolerant concurrent and distributed programs. While transactional distributed systems have been studied for a long time, distributed exception handling techniques are only now evolving towards maturity, especially within asynchronous multi agents or component based systems. In this paper, we review two recent proposals for distributed exceptions handling systems (DEHS), namely SaGE and Guardian, in the light of what the Erlang programming language brings to the table : native constructs for concurrency, distributedness and exception handling across processes. We expand on the merits and possible drawbacks of these systems. We advocate the need to introduce the notion of *resumption*, an often downplayed feature of EHSs, to modern day DEHS, in order to address the problem of multi agent systems facing the "real world".

1 Introduction

Exception handling mechanism have been a feature of programming languages for about thirty years now. The complete feature set of an exception handling system (EHS) is well known and stable (chap.1.1) in the context of non-concurrent, non-distributed programs [1, 2, 3, 4]. Yet within concurrent and distributed systems, it remains an active research topic [5].

Distributed computing has been addressed in length, with relation to programming languages. On one hand, distributed systems have been built with languages without any support for it except system call level primitives for process instantiation and message passing (like C or C++). On the other hand, actor's model programming language have been devised to make distribution and concurrency high-level and first-class concepts available to the programmer (recent examples : *Erlang* [6], *JoCaml* [7], *Oz* [8]). Thus, it is no surprise that in those recent languages we find building blocks, and even potent implementations of distributed EHS. We will begin to review the most mature of them, Erlang.

N. Guelfi (Ed.): RISE 2004, LNCS 3475, pp. 82–92, 2005.

There seems to be a middle ground between those poles, around Java. Java has already its own primitives to help building multi-threaded and distributed (RMI) programs. But in Java, there is no language support to manage exception handling across thread groups and process boundaries.

Thus, in this paper, we also review two proposals around this limitation of the Java platform : the *SaGE* [9, 10] and *Guardian* [11, 12, 13] systems, which specify and implement a DEHS for asynchronous component based and multi agents [14] systems written in Java. We believe that Erlang, SaGE and Guardian present important insights for the future makers of a DEHS. The CA Action framework is not reviewed there, for we lack sufficient knowledge about it, but it should nevertheless be mentioned.

As a programming language feature, an EHS is mostly orthogonal to other language features, but can be disastrous in the presence of low-level memory management primitives, like in C++, for instance. Similarly, it remains to be seen upon which assumptions it may behave well in distributed and concurrent settings. Through this study, we uncover some of them.

Finally, we advocate a DEHS supporting the notion of resumption of a computation after an exception has been handled.

1.1 The EHS Feature Set

Before we delve in DEHS, let us remember the commonly accepted features of an EHS [15]. Firstly, EHSs have been devised in the context of functional and imperative programming languages, which all carry the notion of a program execution as a sequential series of jumps from current continuation to next normal continuation.

An EHS introduces the notion of an exceptional set of continuations to be executed in the presence of an exceptional situation arising during a computation.

The act of *signaling* is what occurs when a program encounters an exceptional situation. It can be triggered automatically, by perusal of some built-in exception-throwing primitive, or explicitly via some signalment primitive. Signaling implies the building of an *exception object* carrying run-time information about the exceptional situation, and finding the innermost exception handler matching its type.

Exception handlers (EH) allow to define, at any arbitrary point in a program, a new set of exceptional continuations, that adds up to, or overrides part of, the current set ; each exceptional continuation is usually selected using the type of the exception object that is built at signalment time ; an EH has typically dynamic extent and global scope[1].

Most programming language since the past twenty five years, from Ada [16] to Java [17] or Haskell [18], have had this kind of exception handling capabilities, which is also referred to as the "termination" model. Indeed, the research of handlers is done "destructively", that is the call stack is unwound up to the handler to be selected.

[1] Some systems offer statically (class) or lexically scoped handlers.

The possibility to resume activity after signalment exists in the "*resumption*" model, where the signalment is merely a function call. The ability to resume depends on the type of the exception being signaled (an *error* implying termination, and a *continuable error*, or mere *condition*[2], allowing resumption). Thus, a handler may have the choice between termination (then the stack will be unwound up to the handler) or a list of possible resumption points, also described in the Common Lisp terminology as *restarts* [19]. Let us define the notion of restart.

Restarts are a set of continuations whose goal is to provide lexical reparation and resumption capabilities to an EHS. Restarts, being closed over one of the lexical environments located between the signaling place and the innermost last EH, can be told to modify their closed-over environment and do a non-local jump to this environment in order to resume the computation, if abortion is not wanted. They are typically selected and funcalled by an exception handler as the last step in the signalment cycle. Restarts have also dynamic extent and global scope.

Resumption is successfully supported in dynamic languages (mainly Smalltalk [4, 20] and various Lisps). To our knowledge, only Common Lisp and Dylan have restarts, which give a choice of resumption points. We will argue in the last chapter in favor of the resumption model in the context of distributed and concurrent programs.

1.2 Implications of This Feature Set

The main assumptions under which EHS have been built are:

- functional or imperative programming languages (where the notion of continuation makes sense),
- automatic memory management (garbage-collected languages),
- one single control flow,
- synchronous calls (or jumps) between parts of a program,
- determinism of execution and total-orderliness of any program's trace[3].

Distribution and concurrency raise new specific and difficult issues for fault-tolerance that have up to now remained incompletely studied.

1.3 Properties of Distributed, Concurrent Systems

Distributed and concurrent systems are build from small sequential and process-like blocks ; those blocks may be executed independently and concurrently, as long as they don't interact ; they may be distributed on different places, physically speaking ; they may have to cooperate in various ways to achieve a desired outcome : then, communication happens through asynchronous message passing.

[2] "A condition is a generalisation of an error" [19].

[3] This one has to be taken with a grain of salt. Haskell has a simple exception handling mechanism that works in spite of the lazy evaluation strategy ; however it needed a pinch of cleverness to get it working.

[21] identifies three concurrency patterns:

1. disjoint concurrency, where concurrent activities share nothing and do not synchronize (in the manner of independent UNIX processes),
2. competitive concurrency, where different running activities compete for some shared resources (like in the case of transactional systems),
3. cooperative concurrency, where different active entities act collectively to reach a common outcome.

Sometimes it seems like the distinction between the problem of one system's consistency and the handling of distributed, concurrent exceptions, is blurred. We defend the view that exception handling is a low-level construct which ought to remain orthogonal, albeit compatible, with transactional systems. Both are important with relation to software reliability and fault-tolerance, but they really are independent things. From now on, we will mainly focus on cooperative concurrency, without after-thoughts about transactions. Thus, we will not cover the Coordinated Atomic Action (CAA) framework [22, 23, 24] for it mixes those two aspects. We must note however that CA Actions embody most of the idea we will discuss here.

In the context of cooperative concurrency, the questions we face are:

- how to deal with asynchronous message-passing,
- how to design signalment path and boundaries, and exception handlers scope.

Older proposals, such as Guide or Argus [25], helped raise the importance of the notion of *concertation* amongst a set of cooperating entities. As we will show, SaGE and Guardian have retained this notion.

2 An Overview of Three Systems

Erlang is a high-level programming language with built-in features to provide fault-tolerance to massively concurrent and distributed programs. It is a complete, integrated system. It has been proven in the field [6].

On the other hand, SaGE and Guardian are independent of any language. But it must be noted that they fill an important gap in the Java world and their current implementations are purely Java based.

2.1 Erlang

Erlang has been built out of the need of an efficient, robust and massively parallel programming language for the programming of world-class telecoms switches; it has proven successful with relation to these objectives.

Language Constructs. At its core, Erlang is a pure *functional* programming language (i.e closures and function calls are everything) with a simple EHS providing exception signalment and handling but no resumption.

Erlang has a notion of *process*, which is an independent, distributable sequential piece of code. Erlang processes communicate through asynchronous message passing : no shared state is ever allowed between concurrent entities.

There are two types of processes : worker and supervisor processes. A typical Erlang program is a tree of supervisor processes whose leaf nodes are worker processes. At run time, the program is unfolded from the root supervisor process (which can be seen as the "main" process) to the leaves made of worker processes. The act of creating a new process, and message passing between processes, are always explicit.

Exception Handling. Whenever an exception is signaled (typically in a worker process), it can be handled within the process, with the classical set of operators ; if it isn't handled there, then the process automatically sends a message carrying failure notification to its direct supervisor - the last step in signalment - and suicides. Depending of the signaling process's expected durability (permanent, transient or temporary), it is restarted *from scratch* by its supervisor or nothing happens. Additionally, processes can be *linked* in a way that guarantees that whenever one process crashes, all associated processes also die. Thus, one unhandled exception can lead to a whole process group crash (and possibly a reboot by a supervisor).

Thus, a well-written Erlang program has its functionality distributed into a tree structured hoard of independent worker processes, such as to limit the consequences of a process crash (an unhandled exception) to a subtree, or a cluster of disjoint subtrees (by means of process linkage).

Conclusions. This has proven to be a very efficient, allegedly because simple, strategy for Erlang's initial target, namely the software running massive telecoms switches. Also while the Erlang EHS offer simple primitives, it allows the making of very complex systems by way of composition of those primitives.

2.2 SaGE

SaGE is a specialized DEHS, in that it first defines a protocol, in other words a fixed set of interactions between groups of concurrent and (possibly) distributed entities, and then defines a proper EHS on this protocol, which is named "service". SaGE has been implemented for the Madkit [26] multi agents system and the Jonas J2EE [27] component-based framework.

Services. A service is basically modelled after method invocation in object oriented languages ; it associates an agent, a name, a set of formal parameters and a piece of functionality. A service is said to be provided by an agent A, invoked by an agent B, and then executed by agent A.

There are however important differences with method invocation. In a non-concurrent object-oriented program, there is a call chain of method invocations; with services, there is a tree of services invocations.

This is because:

- one agent can execute in parallel many instances of any service (mapping to an even number of service invocations by possibly different agents),
- one service execution which invokes another service execution (it is then said to be a "complex" service) is not synchronously stuck to its normal or exceptional outcome, but can err on its own and indeed invocate many other sub-services, for the sake of redundancy or merely for it is profitable to exploit the distributedness of the agents, making the worst-case execution time the maximum of the execution of the distributed sub-services instead of the sum.

Such a protocol is said to be "semi-synchronous", that is an invoking process is free to do whatever it pleases until some later time when it decides to collect the answers, which it is, contractually, bound to do. This is much like Multilisp's "futures" [28].

Exception Handling. On a call-tree degenerated into a chain (or stack), the EHS behaves quite like in a monolithic Java program. There is a new type of exception : *SageException* ; for each service, one can associate a set of handlers with any set of subtypes of SageException.

In the general case, there is an important piece of functionality available that does not belong to the EHS of non concurrent systems. We have said that one service can issue many sub-service invocations for the sake of redundancy ; when doing this, one does not want to have our service execution killed because just one (or a few amongst many) redundant sub-service execution failed and signaled an exception ; instead it is practical to craft a special function that filters the exceptions and can decide not to handle or propagate upwards, that is to merely do nothing special and go on with the computation. This function is named the *"concertation"* function [29].

Limits and Conclusion. We can model a problem with SaGE as long as it fits well into the notion of service. That is, it works well for any problem for which a functional decomposition comes in handily like, for instance, information retrieval systems.

Indeed, SaGE tries hard to put the least possible amount of ordering constraints on the execution of services, in order to benefit as much as possible of concurrency of the executions. The only constraint is that a service execution starts before any of its sub-services, and terminates only after all of them have terminated. Thus, when service execution side-effects an agent's mental state or the outer world, then suddenly every kind of interlocking and race conditions raise its head ; at this point, the programmer has to cope with the ordering and serialisation of access to the shared entities through low level constructs like locks and mutexes, which are hard to get right and may have scalability issues. To avoid these issues, one must use a transactional system along with SaGE.

While restricted in various ways, SaGE is nevertheless an efficient and expressive framework that provides a simple DEHS on top of an oo-like interaction model.

2.3 Guardian

While SaGE defines a DEHS upon a set of well defined interactions, Guardian strives to be a complete DEHS, in the sense that it can work with any protocol for which an exception handling strategy can make sense. This mighty goal seems to be attained with Guardian, but at a price ; we will see how and why soon.

Components and Primitives. A multi-agent system extended with Guardian has two main new components : a special agent named "leader", and sets of agents engaged in a collective activity, whose members are said to be "participants". It adds to the participants a set of primitives to manage the exception handling whose scope is the collective activity. Exception handling at the level of a set of participants is said to be global ; accordingly, Guardian introduces a *GlobalException* sub hierarchy to the built-in exception taxonomy, like with SaGE and its SageException.

Those primitives mimic the Java EHS model, with one addition: methods to enable and disable local *contexts*. Contexts are a generalisation of the lexical context, from the viewpoint of exception handling of the monolithic EHSs; in those, it is easy to think about the places where signaling and handling happen: all EHSs provide syntax to help the programmer map an exception handler or a signaling instruction to their lexical contours. Guardian contexts represent phases of the program, those phases being now decoupled from the block structure of the programming language. Moreover, Guardian, being built on Java, cannot benefit from any syntactic sugar ; therefore it has to make context management available to the programmer through the "{enable|disable}Context" pair of primitives.

So, when writing an application exploiting the capabilities of Guardian, one has to think about the specification of the participants, which are bound to use the Guardian primitives to stack up contexts, to install exception handlers on those contexts and to signal global exceptions ; and about the leader, which computes the set of exceptions to be handled in the participants in response to a set of concurrent global exceptions.

According to the authors, the gist of exception handling is the choice of the "semantically correct" global exception to be signaled by a participant, so as to allow the leader to compute a meaningful concerted exception set in return. Off course, the choice depends heavily on the type or structure of the collective activity.

The Signalment Process. Signaling is a two phases mechanism, after a first global exception is signaled:

1. the leader is alerted, tells every other participant to suspend any work and wait synchronously for instructions, and then it collects all pending global exceptions,

2. when all the participants are ready, the leader computes a set of concerted exceptions to be raised in the participants in response to the set of global exception it has received, then it sends the computed exceptions back to the participants, which can then enter their handler and go on with their (concurrent) activities.

We should note that the leader is akin to the concertation function from SaGE: concertation is the central mechanism in Guardian whereas it is an auxiliary mechanism in SaGE.

Limits and Conclusion. Guardian has been built to address many, if not all, of the situations a DEHS has to cope with. For instance, services (like in SaGE), coordinated atomic actions (CAA) and conversations can be built using the Guardian framework.

But it does not scale well : the need to freeze all participants' activity whenever a global exception is signaled makes it not usable when the participant set becomes big or if some participants have to perform some uninterruptible, high-priority or real-time service. Moreover, the system would be sensible to crashes or unavailability of the leader ; uniqueness of the leader seems like a weak point.

Furthermore, one important problem remains the complexity of the system, for it is quite hard to figure out how to use it properly.

These two points relate to the fact that Guardian is more than a DEHS framework and seems to encourage the programmer to build transactional-like systems using exception handling capabilities, since those come with automatic agent synchronisation for free. In fact, there is probably a wrong coupling between EHS semantics and execution serialisation : both should probably not come in the same package, and under the same name.

3 The Case for Resumption

There is an emerging pattern around these systems : the global activity of the cooperative entities is well handled if it is spread in a tree of contexts ; exception handling in this case is, in part due to the notion of concertation, now well understood.

But the three systems lack the ability to resume an activity *around* the signaling point. While there are many reasons for not wanting to use an EHS with resumption, even in the context of non-concurrent, non-distributed programs[4], there are nevertheless good reasons to want it : we want to show there exists a world in which a DEHS with resumption is a good thing. First, we will see why Erlang hasn't got it, then why it is potentially usefull.

3.1 Erlang Philosophy: "Fail Fast"

The designers of Erlang provided an in-depth explanation for their design choices and why there is no resumption in Erlang : they had to built software that run

[4] See the main book on fault tolerance by Anderson & Lee.

continuously for years without interruption, and allowing incremental live upgrades. To get this, they devised a simple but expressive language (thus functional), with strong grips on massive concurrency and distributedness (hence the notion of processes as part of the language) and advanced serviceability and reliability features (thus, amongst many other features, a *simple, terminal* exception handling system).

The "fail fast" strategy embodied by Erlang has its roots in the early work on high-reliability systems (such as Tandems systems). Resumption may add complexity to the code and rarely tested code paths. In the context of hundred of thousands of concurrently running processes, it is wise to keep it simple, stupid, and be just brutal : when a process or a group of process is in fault, we had better "reboot" it, in the hope that it will work. And it just works.

But it works in the context of telecoms switches, which exposes the real world to the software in a (we believe) limited and controlled manner. The cost of rebooting a subtree of the running application may be, in those circumstances, lower than the maintenance burden of more complicated (and thus potentially bug-ridden) code paths.

We argue, with no more tools than common sense, that in some cases, resumption would be of great help.

3.2 Usefullness of Resumption

There are two sides to the arguments : some non-fatal conditions may have to be signaled, and some long-running processes had better not been restarted from scratch in the presence of non-fatal conditions.

Condition Handling. A terminating DEHS can only help dealing with hard, uncorrectable errors. In practice, there are a lot of non-error conditions that are to be dealt with, or possibly just ignored. The more a system is open to "real world" entities, the more it has to deal with those conditions. When some condition can be safely ignored by some program, we need a way to say "I don't care" in no more words.

Sometimes, these conditions have to be taken seriously : someone ought to plug in this network cable again, so that we can go on with our current work set. We need a way to be notified that something is going wrong, yet ask for a reparation by a different entity, and just resume work after reparation.

Resilience of Long-Running Processes. Very often, smallish conditions will grow, if unhandled, into proper errors. That is because they are causally linked : a non-erroneous low batteries notification shortly precedes a fatal power outage, if nothing is done in time. So we have to correct the small glitches in order to prevent the bigger breakages. This is even more needed in distributed and concurrent systems facing the real world, that is, a lot of unexpected albeit not yet deadly events.

To sum up, whenever a robot, in a factory, experiences an exceptional and locally unsolvable situation, we don't think that destroying the robot and replacing it is a good way to handle the problem.

4 Conclusion

Resumption is a useful feature that prevent, where needed, a lot of ad-hockery. We may need it in the future DEHS. It would be an interesting experiment to bring them to Erlang, or to extend Common Lisp with primitives to deal, *à la* Erlang, with issues of concurrency and distributedness.

References

1. Weinreb, D.L.: Signalling and handling conditions. Technical report, Symbolics, Inc., Cambridge, MA (1983)
2. Koenig, A.R., Stroustrup, B.: Exception handling for C++. In: Proceedings "C++ at Work" Conference. (1989)
3. Pitman, K.: Exceptional situations in lisp. In: EUROPAL'90. (1990)
4. Dony, C.: A fully object-oriented exception handling system : Rationale and smalltalk implementation. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001)
5. Romanovsky, A., Dony, C., Knudsen, J.L., A.Tripathi: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001)
6. Armstrong, J.: Making Reliable Distributed Systems in the presence of Software Errors. PhD thesis (2003)
7. Sylvain Conchon, F.L.F.: Jocaml : mobile agents for objective caml. (1999)
8. Peter Van Roy, S.H.: Mozart, a programming system for agents applications. (1999)
9. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: A proposition for exception handling in multi-agent systems. In: SELMAS'03 International Worskshop proceedings. (2003)
10. Frédéric Souchon, Sylvain Vauttier, C.U.C.D.: Fiabilité des applications multi-agents : le système de gestion d'exception sage. (2004)
11. Tripathi, A., Miller, R.: Exception handling in agent-oriented systems. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001)
12. Miller, R., Tripathi, A.: Primitives and mechanisms of the guardian model for exception handling in distributed systems. In: Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference) proceedings. (2003)
13. Miller, R.: The Guardian Model for Exception Handling in Distributed Systems. PhD thesis (2003)
14. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. (Addison-Wesley Pub Co; 1st edition (February 25, 1999))
15. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Communications of the ACM **18** (1975) 683–696
16. J. Ichbiah, J.G.P. Barnes, J.H.B.K.B.O.R.B.W.: Rationale for the design of the ada programming language. In: ACM Sigplan Notices. Volume 14(6B). (1979)
17. Sun Microsystems Mountain View, Calif.: Java 2 Platform, Standard Edition (J2SE). (2004) http://java.sun.com/j2se.

18. Jones, S.P.: Tackling the awkward squad : monadic input/output, concurrency, exceptions and foreign-language calls in haskell. (2002)
19. Pitman, K.: Condition handling in the lisp language family. In: Advances in Exception Handling Techniques. (2001)
20. Dony, C.: Exception handling and object-oriented programming : towards a synthesis. ACM SIGPLAN Notices **25** (1990) 322–330 *OOPSLA/ECOOP '90 Proceedings*, N. Meyrowitz (editor).
21. Romanovsky, A.B., Kienzle, J.: Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In: Advances in Exception Handling Techniques. (2001) 147–164
22. Romanovsky, A.B.: Conversations of objects. Computer Languages **21** (1995) 147–163
23. Wu, B.R.A.R.C.R.C.R.S.Z., Xu, J.: From recovery blocks to concurrent atomic actions. In: Predictably Dependable Computing Systems. ESPRIT Basic Research Series (1995) 87–101
24. Romanovksy, A., Kienzle, J.: Action-oriented exception handling in cooperative and competitive object-oriented systems. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001) Also available as Technical Report (EPFL-DI No 00/346).
25. Liskov, B.: Distributed programming in argus. In: Communications of the ACM, vol. 31, n°3. (1988) 300–312
26. Jacques Ferber, O.G.: Madkit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems. (1997)
27. : Java open application server (JOnAS) 4.1 : a J2EE platform (2004) http://www.objectweb.org/jonas/current/doc/JOnASWP.html.
28. Halstead, R., Loaiza, J.: Exception handling in multilisp. In: 1985 Int'l. Conf. on Parallel Processing. (1985) 822–830
29. Issarny, V.: Concurrent exception handling. In: Advances in Exception Handling Techniques. (2001)