Modular fault tolerance in a network-transparent language

Peter Van Roy, Raphaël Collet, Sébastien Doeraene, and Géry Debongnie
{peter.vanroy,gery.debongnie}@uclouvain.be, {sjrdoeraene,raphael.collet}@gmail.com
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium

A programming system is said to be *network transparent* if program source text executed over several nodes gives the same result as if it were executed on a single node, provided network delays are ignored and no failure occurs. The system is said to be *network aware* if programs can predict and control their physical distribution and network behavior. Formal definitions of both properties are given in [2]. Both of these properties together aim to simplify distributed programming by separating a program's functionality, in which distribution can be ignored, from its distribution behavior, which includes network performance, partial failure, and security. We have implemented a system called Mozart [9] that combines network transparency and network awareness so that partial failure may be handled in the following way. First, when there are no failures (reliable ordered communication and correct process execution) then the system combines both properties almost perfectly. To be precise, network transparency is perfect (language semantics are independent of distribution) and network awareness is almost perfect (simple cases have the same number of messages and hops as hand coding and complex cases are occasionally more complex than hand coding). Second, when failures can occur (assuming a crash-stop process model and ordered message delivery with arbitrary message delay or loss), then the system does the following: (1) the program performs no incorrect operation but individual program threads may suspend if they cannot continue correctly, and (2) the program provides a modular failure detection mechanism called *fault streams* that allows a second program to perform the appropriate tasks to handle the failures at the application level. Therefore, the source text of the original program needs no modification in the case of partial failure: it can be written assuming perfect network transparency and a second program text adjoined to it to handle failures.

We implement network transparency by combining both language and algorithm design. Our implementation is based on a language, Oz, that makes a clear distinction between three kinds of entities: stateless, single assignment, and stateful. The language has annotations to choose the distributed algorithm used to implement each language entity. The resulting language, Distributed Oz, is implemented by a set of distributed algorithms according to each kind of entity [5]. Objects are implemented by a mobile state protocol, which allows their state pointer (the right to update the object state) to move atomically between nodes [11]. Ports (FIFO communication channels) are implemented by an asynchronous message passing protocol. Values (constants) such as records, lists, procedures, and classes, are implemented by eager and lazy copying protocols. Single assignment values (which combined with the functional subset of Oz define a declarative dataflow sublanguage) are implemented with a distributed rational tree unification algorithm [6]. In the current implementation, language entities have no special support for fault tolerance. Both the language and distributed algorithms are designed for network awareness. For example, a simple client/server is implemented by a combination of ports and dataflow values. The client sends a query containing an unbound dataflow value

to a port located at the server node. The server replies by binding the dataflow value. The whole operation consists of a single round trip: one sent message (to the port) and one return message (distributed unification binding the client's dataflow value). The distributed unification algorithm is designed to make the simple cases as efficient as hand coding and the complex cases correct. In the current implementation, the complex cases occasionally need more messages than hand coding.

Distributed Oz as described above is fully implemented in the Mozart system and maintains perfect network transparency when there are no failures. Let us now motivate the system's behavior when there are partial failures. Partial failure clearly breaks network transparency because it cannot be hidden in general (given our fault model, this is a consequence of the CAP theorem [3]). For example, an object located on a given node will simply disappear if the node crashes. If we cannot maintain transparency, what is the best we can do? Many systems allow the new failure modes to show up at the points where they affect execution. For example, an RMI in Java will cause an exception to be raised at the caller if the callee's node fails. This seems to be a straightforward way to handle the problem: simply catch the exception and perform the necessary cleanup. However, this solution is not modular. If we distribute a program, then new behavior (e.g., new possible exceptions) will arise depending on how the program is physically distributed. The program will have to be modified to catch these exceptions and do the appropriate fixes. If the system is network transparent, then all possible distribution structures must be handled, so all potential distribution points have to be recognized and handled a priori. This greatly complicates the program. One way to solve the problem is to limit the possible distribution structures. For example, in Java the distribution structure is typically fixed at the time of program development. This limits how the program can later be evolved to changing requirements or changing system structures.

We propose a different way to handle partial failure in a network-transparent language. Our approach does not require changing any existing code, but just the addition of new code to handle failures. This modular approach is called *fault streams* and it is implemented in Mozart 1.4.0 [2,9]. The approach is simple: every language entity has an associated fault stream, which gives the entity's successive fault states. We define a *stream* as a list value with a single assignment tail, which allows it to be extended monotonically. There is a language operation to make visible the fault stream of any language entity; if the fault stream is not visible then it has no overhead. Any operation that cannot be completed because of a partial failure will simply suspend (i.e., wait until the problem goes away), and a failure notification will appear on the fault stream. Another program module is then free to read the fault stream and act accordingly.

In the Mozart 1.4.0 implementation with the fault model given above, three fault states are recognized: *ok* (no fault), *tempFail* (temporary failure), and *permFail* (permanent failure). This combines a perfect failure detector (*permFail*) and an eventually perfect failure detector (*tempFail* corresponds to suspect and *ok* corresponds to resume) [4]. Permanent failures are difficult to implement; currently they are only detected between processes on single machines and on local area networks in certain circumstances. We therefore allow the perfect failure detector to be omitted at any given node, but the eventually perfect failure detector must always be implemented. We note that temporary failures take the place of time outs. A temporary failure is supposed to be detected quickly, unlike a time out which approximates infinity. This allows the application to make the appropriate decision in a timely manner. Fault streams can

be seen as a generalization of Erlang's linked processes [6]. In Erlang, any two processes can be linked so that if one process dies, the second receives a message reporting the failure. Our fault streams generalize this in three ways: (1) we handle an ordered sequence of fault states, (2) we handle temporary failures, and (3) we handle failures at the level of language entities and not at the level of processes. Note that Erlang processes are fine-grained; they contain just a single thread executing a recursive function and a mailbox to handle incoming messages.

We are extending this work by investigating how to program large-scale systems in Distributed Oz. We have implemented a transactional peer-to-peer storage system, Beernet, in Distributed Oz [8]. This system uses a uniform consensus algorithm to implement atomic commit. Our experience with this system has exposed another difficulty for network transparency, namely resource management in the face of failure, which shows up in all long-lived distributed systems subject to partial failure. A large-scale system that lives long will have resource leaks because of partial failure. We are currently investigating formalizations of software rejuvenation to solve this problem [7]. This can be seen as an application of the ideas of Recovery Oriented Computing to distributed programming language design and semantics [10]. The ultimate goal is to build efficient large-scale distributed systems in the most network-transparent way possible.

References

[1] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. Ph.D. dissertation, Royal Institute of Technology, Stockholm, Dec. 2003.

[2] Raphaël Collet. *The Limits of Network Transparency in a Distributed Programming Language*. Ph.D. dissertation, Department of Computing Science and Engineering, Université catholique de Louvain, Louvain-la-Neuve, Dec. 2007. Available at pldc.info.ucl.ac.be.

[3] Seth Gilbert and Nancy Lynch. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. ACM SIGACT News 33(2), June 2002, pp. 51-59.

[4] Rachid Guerraoui and Luís Rodrigues. "Introduction to Reliable Distributed Programming". Springer-Verlag, 2006.

[5] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. *Programming Languages for Distributed Applications*. Journal of New Generation Computing 16(3), May 1998, pp. 223-261. Available at pldc.info.ucl.ac.be.

[6] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. *Efficient Logic Variables for Distributed Computing*. ACM Transactions on Programming Languages and Systems (TOPLAS), May 1999, pp. 569-626. Available at pldc.info.ucl.ac.be.

[7] Yves Jaradin and Peter Van Roy. *A Formal Model of Software Rejuvenation for Long-Lived Large-Scale Applications*. 2011. Draft available from the authors.

[8] Boris Mejías and Peter Van Roy. *Beernet: Building Self-Managing Decentralized Systems with Replicated Transactional Storage*. IJARAS: International Journal of Adaptive, Resilient, and Autonomic Systems 1(3), Jul.-Sep. 2010, pp. 1-24.

[9] Mozart Programming System version 1.4.0, www.mozart-oz.org, July 2008.

[10] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kıcıman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Computer Science Technical Report UCL/CSD-02-1175, UC Berkeley, March 15, 2002.

[11] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. *Mobile Objects in Distributed Oz*. ACM Transactions on Programming Languages and Systems (TOPLAS), Sep. 1997, pp. 804-851. Available at pldc.info.ucl.ac.be.