Building Dependable Internet Services with E-speak

Svend Frølund, Fernando Pedone, Jim Pruyne, Aad van Moorsel
Software Technology Laboratory
HP Laboratories Palo Alto

HPL-2000-78
June, 2000

E-speak, dependability, high-availability, E-services

The process of moving business to the Internet through e-commerce solutions seems to have gained unstoppable momentum. We believe, however, that the success and pace with which business services will migrate to the Internet will be dictated by the ability to provide them with dependability guarantees. In this short paper we survey the progress we are making towards providing fault-tolerating mechanisms in e-speak, the Open Source software platform for building e-commerce applications. Among the various issues we discuss are the interceptor programming model as infrastructural enabler of dependability, high-availability features of the e-speak platform itself, and exactly-once semantics and transactional properties for services running on top of e-speak.

Building Dependable Internet Services with E-speak

Svend Frølund, Fernando Pedone, Jim Pruyne, Aad van Moorsel
Hewlett-Packard Laboratories, 1501 Page Mill Rd., Palo Alto, CA
`frolund,pedone,pruyne,aad@hpl.hp.com`

Abstract

The process of moving business to the Internet through e-commerce solutions seems to have gained unstoppable momentum. We believe, however, that the success and pace with which business services will migrate to the Internet will be dictated by the ability to provide them with dependability guarantees. In this short paper we survey the progress we are making towards providing fault-tolerating mechanisms in e-speak, the Open Source software platform for building e-commerce applications. Among the various issues we discuss are the interceptor programming model as infrastructural enabler of dependability, high-availability features of the e-speak platform itself, and exactly-once semantics and transactional properties for services running on top of e-speak.

1 E-Speak

E-speak is an Open Source effort initiated by Hewlett Packard that targets the rapid development of Internet business applications [5, 8]. It anticipates a world in which 'e-services' will interact dynamically, transparently and securely over the Internet to offer a multitude of services. This service-centric scenario is known as 'Chapter 2' of the Internet, following the current chapter of document-centric standalone e-commerce sites.

In this paper we discuss how to build dependable Chapter 2 e-services with e-speak. We restrict our attention to dependability as far as it relates to fault-tolerance.[1] Robustness against failures (accidental or malicious) is crucial for e-commerce solutions, since service breaches correspond to business (and thus money) loss.

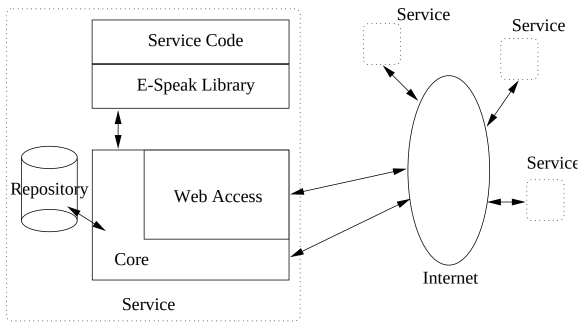

Figure 1. E-Speak Architecture

Let us first introduce e-speak to the extent relevant for dependability solutions. More complete or alternatively focused presentations can be found on the e-speak web site `http://www.e-speak.net`, or in [5], which provides a web-centric discussion. In e-speak, each service consists of an engine that provides *web access* to the service and mediates all communication through its *core* (see Figure 1). The tasks of the core include message routing, service advertisement, service discovery, local naming, and dealing with various aspects of security. To store names and service handles, the core uses a *repository*. Note that the core's advertisement and discovery functions are staples of the Chapter 2 vision behind e-speak.

Applications (that is, services) running on e-speak link with the *e-speak library* to connect to

[1]We use the terms dependability, reliability and availability liberally and interchangeable in this paper. If necessary, we introduce precise terminology for various system properties in the respective sections.

the core (the current e-speak implementation is all Java). Various programming models for services are offered, including synchronous method invocation and asynchronous XML document exchange. The engine's web access module allows browsers to interface with e-speak services [5], and to complete the interoperability picture, gateways to non-e-speak aware services can be added.

In what follows, we provide e-speak with fault-tolerance in three steps. First, we introduce basic 'infrastructural' support for dependability through interceptors and reliable messaging. Then we make the core and repository of the platform dependable. Finally, we provide protocols for service dependability. We note that not all of the infrastructural and platform aspects are part of the Open Source, but we will make that clear in the text. The service-level dependability solutions are mostly based on recent research results, some of them accompanied by prototypes on different platforms than e-speak. In other words, we discuss both implemented solutions and research results in this paper, together representing our vision on what constitutes dependable Internet services, and how to build them. This vision can best be characterized as focused on end-to-end, service-level, dependability solutions, built in an (as much as possible) application-transparent manner.

2 Infrastructure Support for Dependability

Some of e-speak's features provide inherent support for dependability: the dynamic discovery of services helps to facilitate fail-over mechanisms, and the use of a repository helps to create service state persistence. As discussed in the following sections, e-speak's infrastructure includes two more aspects important for dependability: an interceptor programming model and reliable persistent messaging.

2.1 Interceptor Programming Model

One of the primary design concerns behind our work is transparency to application code. Separating dependability features from the application code is beneficial for various reasons, one being

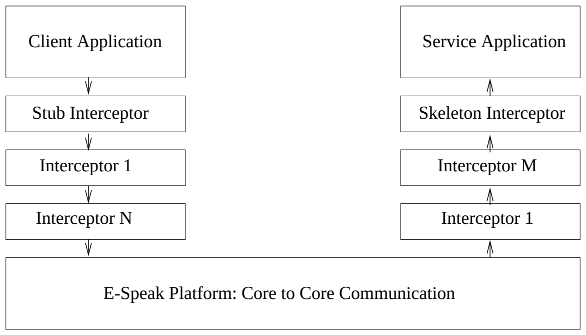

Figure 2. Logical Representation Interceptors

that properties can be guaranteed across applications, without modifying the application code. An important building block in our approach to providing this separation is the addition to e-speak of an interceptor programming model [7].

An interceptor logically resides between the application and the e-speak middleware, see Figure 2. At this level, the interceptor is able to inspect and alter all interactions between the application and the middleware. This permits one to modify the interactions between application components, such as to introduce the platform and service-level dependability protocols described in Section 3 and Section 4. Obviously, the use of interceptors is not limited to reliability, but can be useful for all sorts of QoS properties, as well as functional features. In addition to the hooks for interceptors, we also built an operational management platform to allow dynamic insertion and removal of these interceptors to running services (this is not part of the Open Source release.)

Interceptor control is part of the e-speak library. When a service registers with a core, an interceptor control service for it is also created. The control interface is invoked by the e-speak dispatch code on each remote method invocation, and is responsible for insuring that the proper interceptor modules are called. Similarly, on the client side, interceptor control is invoked by the stub object generated by the IDL compiler.

The critical run-time interceptor objects are the 'request' object, the 'control' object, and the actual interceptor implementation object. The request object stores a reified representation of each remote invocation, facilitating inspection, alteration or forwarding of the call. The control object controls the use of interceptors, such as installing and removing them. In addition, interceptors can be executed in series and the control object does record keeping on the chain. Creating a new interceptor involves subclassing an abstract class containing methods `initialize` and `invoke`. When a new interceptor is installed, the control object calls its initialize method. The control object starts execution of interceptors by calling `invoke` on the appropriate interceptor, and the interceptor calls `invokeNext` (with the possibly modified request object) to give control back to the control object for passing down the chain of interceptors. In [7] one can find further details, and also some examples of the use of interceptors. In this paper, we will point out for individual cases how we use interceptors to provide dependability to services in a transparent manner.

It should be noted that interceptors are widely used in middleware systems, such as CORBA and DCOM. The e-speak interceptor programming model is slightly different, and in some ways more practical to use. In particular, e-speak interceptors do not rely on upcalls, which makes it much easier to insert logic that would otherwise use multiple upcalls. We refer to [4] for more details.

2.2 Persistent Reliable Messaging

To facilitate communication-level dependability properties, e-speak incorporates persistent and reliable messaging. At the time of writing it is not clear what parts will be in a future Open Source release.

Persistent messaging is implemented using a persistent store, and reliable messaging is based on a three-way messaging protocol. Together, these two messaging services make it possible to assure exactly-once message delivery. If clients want this property, they can query for a service provider that offers persistent and reliable messaging. This way of identifying and possibly negotiating dependability properties is of great interest for multi-party service-level dependability guarantees we discuss in Section 4.

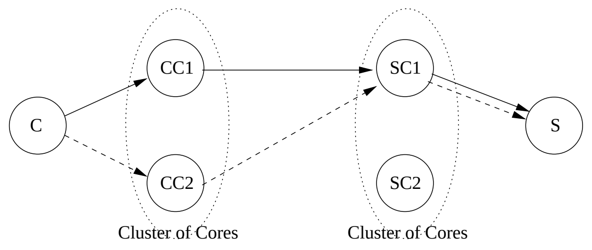

Figure 3. Client-Side Core Fail-Over

3 Platform Dependability

Having the infrastructural tools in place, the next step in providing dependable Internet services through e-speak is to harden the platform itself. In particular, the e-speak core and the associated repository should be made fault-tolerant. To achieve this, we use basic and widely accepted fault-tolerance techniques: database clusters and process groups.

The core repository contains data about services registered with the core, stored in a database. If one takes the core perspective, the repository contains its recovery state, and it is therefore extra important to make the repository reliable. To do this, we use a database clustering solution, such as HP ServiceGuard and Microsoft Wolfpack. Such clusters have multiple CPUs, which share a disk. If a database crashes, its failure is diagnosed as such, and a database is restarted on one of the two CPUs. It should be noted that in basic database clustering the recovery time is strongly dependent on the time to replay the recovery log, which may take minutes. When we discuss three-tier service-level dependability in Section 4.3, we introduce a novel alternative to alleviate this recovery time bottleneck [6].

To improve the availability of the core, we use process groups. In a group of cores, each core handles its clients, and in case of a core crash one of the other cores takes over the jobs of the crashed core. This type of fault tolerance is typical for software processes such as web servers, and improves both reliability and performance.

Let us first consider core failures at the client side, illustrated in Figure 3 (the initial communication is the solid line, and the fail-over path is

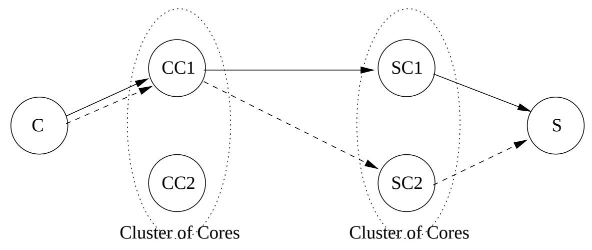

Figure 4. Service-Side Core Fail-Over

dashed). In the grouping solution, when a client connects with a core, it gets information about a list of core names (*CC1, CC2*), of which it selects one. Say the client connects with *CC1*. Since it knows which cores belong to the group, if core *CC1* fails, the client can reconnect to another core (this time core *CC2*). All state information needed by core *CC2* is available in the repository. To implement this solution, we can make use of the interceptor framework for failure detection and retries. The details of such an implementation are currently under investigation.

Consider now service-side core failures. Following the scenario depicted in Figure 4, the client rebinds to *SC2* when *SC1* fails. When the service registers with the different cores in the group, a unique name is created at each core, and using the standard e-speak connection set-up fail-over is established. Note that the repository is assumed to be shared among the cores. We therefore introduce a light-weight registration operation that only creates a handle to the service, but does not advertise the service multiple times, nor store redundant service information. Failure detection as well as rebinding may be implemented through a client-side interceptor.

4 Dependable Services

The ultimate objective of our dependability work is to guarantee dependability properties for the actual services running on e-speak. In this section, we introduce several solutions for service-level dependability. We discuss first recovery of a single service, then end-to-end dependability for three-tier applications, and finally dependability for service-level 'transactions' spanning multiple parties (a Chapter 2 scenario).

4.1 Service Recovery

There is a base mechanism for recovering services from crash failures. It uses e-speak's web access persistent queue as reliable storage for the service state. A recovery manager controls the restart of the service if a failure is detected.

4.2 Three-Tier Solutions I: Exactly-Once Transactions

We discuss protocols to provide exactly-once semantics in three-tier applications, given that crash failures can occur in both services and databases. Many current (Chapter 1) e-commerce applications follow a three-tier structure. In this setting, front-end clients are thin, for example browsers and applets. Middle-tier application servers are stateless, examples include traditional web servers as well as application servers, such as BEA WebLogic or Microsoft COM+. The back-end is typically one or more standard database systems, such as Oracle 8i.

It is very challenging to provide end-to-end reliability in three-tier systems without violating their nice properties (scalability and manageability). Fault tolerance efforts, such as in CORBA, do not solve the end-to-end problem, but instead deal with two separate issues (client-server and server-database, respectively) that do not add up [1]. We define the concept of an *e-transaction* as a practical and desirable end-to-end reliability guarantee for three-tier applications [2]. The 'e' in e-transactions stands for exactly-once, and reflects the fact that clients want their requests to be processed with exactly-once semantics. Moreover, with e-transactions, clients get transactional guarantees, even though they are not part of the 'real' server-side transaction. Roughly speaking, if a client submits a request 'within' an e-transaction (and does not crash), the client will eventually receive a reply, and this reply is the result of a server-side transaction that has committed. Note that since the notion of exactly-once in this section relates to an end-to-end property, this notion is more encompassing than that of exactly-once messaging in Section 2.2.

We have also defined various protocols that implement e-transactions in practical settings. We assume off-the-shelf databases that run the XA interface. We assume that clients may not recover, and that clients cannot accurately detect server-side failures. Finally, we maintain the stateless nature of middle-tier servers as much as possible. With a single database, middle-tier servers are completely stateless. With multiple back-end databases, middle-tier servers communicate with each other to coordinate their actions, but they do not store recovery information on their local disk.

In all our e-transaction protocols, some basic issues on the client side as well as the service/database side must be addressed. We will briefly outline these basic ideas behind our solutions. In our protocols, it is the client's task to diagnose service failures, and reissue requests. The service, on the other hand, must be able to determine the outcome of a transaction for which the client resubmits a request. If a client suspects a failure, the client can not know whether the service failed before or after the transaction was committed against the database, and hence the service (in collaboration with the database) has mechanisms to determine this. If the transaction was committed, the result of the first transaction attempt is returned. Otherwise a new transaction is executed. Figure 5 displays a possible design for an implementation based on interceptors. We note that our current implementation is not in e-speak, but in CORBA.

4.3 Three-Tier Solutions II: Fast Database Fail-Over

Typically, the high-availability bottleneck in three-tier systems is the database. The reason is that database failures normally result in log-based recovery: to restart, the database has to establish a consistent pre-failure state from a persistent transaction log. As we remarked when discussing fault-tolerance of the repository in Section 3, this can easily take minutes. Here we discuss a promising new approach, called Pronto [6], to expedite database fail-over. We note that the applicability of Pronto is not limited to three-tier systems, but extends to any set-up in which databases need to be highly available.

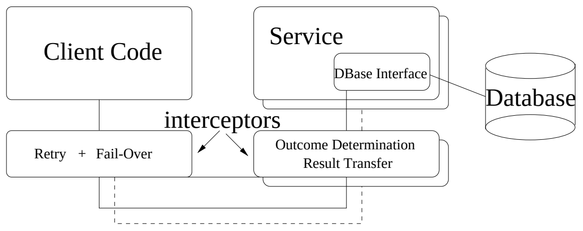

Figure 5. Interceptor-Based E-Transaction Implementation

Pronto uses multiple off-the-shelf database systems to mask the log-based recovery of one database and continue processing transactions against other databases when failures occur. The Pronto protocol is a hybrid approach that has elements of both active replication and primary-backup. Pronto deals with database non-determinism by having a single (primary) database execute transactions in a non-deterministic manner. But rather than checkpoint the resulting state to backups, Pronto sends the transaction itself to the backups along with ordering information that allows the backups to make the same non-deterministic choices as the primary. By shipping transactions instead of transaction logs, we can have heterogeneous databases with different log formats, and prevent the contamination that may result if the data in one database becomes corrupt. Furthermore, the Pronto algorithm does not rely on perfect failure detection, an important property to make database fault-tolerance work in an Internet environment.

4.4 Multi-Party Transactional Guarantees: X-Ability

With Chapter 2 of the Internet, an online service may be implemented by a collection of e-services. Thus, the notion of end-to-end reliability now potentially spans a collection of independently owned and operated e-services. This further increases the complexity of providing dependability guarantees compared to three-tier systems. We can no longer define end-to-end reliability in terms of how these e-services are implemented, for

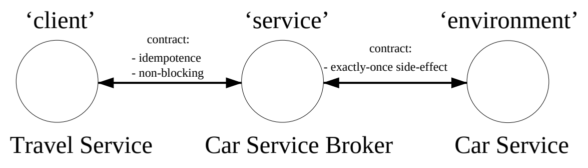

Figure 6. Contracts Between E-Services for X-Ability

example, whether they store their state in a back-end database. Just like we advertise and compose service functionality in terms of attributes and interfaces, we also need to specify and prove reliability properties in terms of abstract, per-service robustness characteristics. We need a way to define reliability contracts between services, and reason about the end-to-end reliability in terms of these contracts.

The notion of x-ability is a way to define such reliability contracts [3]. An x-able service has two contracts: one with its clients and one with its environment (e.g., other services). The client-side contract is based on idempotent, non-blocking request processing. The contract with the environment is based on exactly-once side-effect. Figure 6 displays part of a fictive multi-party interaction involving the booking of a trip, with the x-ability contracts between the various partners. With x-ability, we can reason about reliability correctness of a service as a local property. We can examine if the service satisfies its contracts. With x-ability, end-to-end reliability is compositional.

We note that our current x-ability work is of an abstract and theoretical nature, and that many implementation issues still need to be resolved. Concerning such implementation, it seems natural to use interceptors, guided by deployment management to establish contracts between partners.

5 Conclusion

The overview of the e-speak dependability features in this paper distinguishes infrastructural solutions, platform solutions, and service solutions, respectively. Together, they establish service-level dependability guarantees, which we implement as transparent as possible in relation to the services.

From a research perspective, the dependability challenge in Chapter 2 of the Internet lies in service-level dependability. Most existing solutions do not consider dependability from an end-to-end perspective in three-tier systems, let alone for dynamically created, multi-party, composed e-services. Our work on x-ability is a first step, but many more dependability issues need to be addressed at the level of concatenated, multi-party services. A rich, and critical, area of dependability research thus remains.

References

[1] S. Frølund and R. Guerraoui, "CORBA fault-tolerance: Why it does not add up," in *Proceedings of the IEEE Workshop on Future Trends of Distributed Systems*, December 1999.

[2] S. Frølund and R. Guerraoui, "Implementing e-transactions with asynchronous replication," in *The International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8)*, New York, New York, USA, June 2000.

[3] S. Frølund and R. Guerraoui, "X-ability: A theory of replication," in *Proceedings of Principles of Distributed Computing*, Portland, Oregon, USA, July 2000.

[4] S. Frølund and J. Pruyne, "In support of generic proxies," in *Workshop on Reflective Middleware (in conjunction with Middleware 2000)*, New York, New York, USA, April 2000. Position paper.

[5] S. Graupner, W. Kim, D. Lenkov, and A. Sahai, "E-speak–an enabling infrastructure for web-based e-services," in *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, l'Aquila, Italy, July 31–August 6 2000.

[6] F. Pedone and S. Frølund, "Pronto: A fast fail-over protocol for off-the-shelf commercial databases," Technical report, Hewlett-Packard Laboratories, April 2000. Submitted for publications.

[7] J. Pruyne, "Enabling QoS via interception in middleware," Technical Report HPL-2000-29, Hewlett-Packard Laboratories, February 2000.

[8] `http://www.e-speak.net` E-Speak Open Source.

Please refer to references within the above documents for a more extensive survey of relevant literature.