CatchAndRetry: Extending Exceptions to Handle Distributed System Failures and Recovery

Emre Kıcıman
Microsoft Research

Benjamin Livshits
Microsoft Research

Madanlal Musuvathi
Microsoft Research

ABSTRACT

In this paper, we present CATCHANDRETRY, an extension of the traditional exception mechanism to provide language-level support for common recovery techniques in distributed systems. We motivate and justify our design by analyzing several cases studies taken from the context of Facebook. CATCHANDRETRY is a language mechanism that is general enough to apply to multiple tiers of a distributed application; throughout this paper, we illustrate CATCHANDRETRY with examples of its use within both a large-scale distributed server-side application running in a data center as well as a JavaScript clients-side application running within a web browser.

1. INTRODUCTION

In today's programming languages, exceptions provide developers with a mechanism to raise and propagate signals that some error or other special condition has occurred. Traditionally, these mechanisms have not included support for reacting to and recovering from such errors and other conditions, because such recovery is usually application-specific. We believe, however, that a set of general-purpose recovery mechanisms exists for common classes of errors in distributed systems. In particular, large-scale Internet services, as well as other distributed systems, must contend with a variety of common failure and error conditions, including:

- **Data errors:** maintaining data freshness and consistency in a distributed system can be expensive or even impossible to preserve in some circumstances [4]. Such data errors may include reading stale values or reading inconsistent data from multiple sources [2, 14].
- **Availability errors:** whether because of network partitions, hardware failures, performance stutters, or software bugs, code within a distributed system must be prepared to handle the problems that occur when a remote node is unavailable or slow to respond.

Of course, data errors are not completely independent of availability errors. Some data errors, such as missing data, occur when part of the storage system are unavailable.

PLOS '09, October 11, 2009, Big Sky, Montana, USA.

These classes of errors in Internet services and many other distributed systems are often resolved with a layered approach: 1) the initial design of the system incorporates node and data replication to minimize the impact of these errors; 2) a supporting infrastructure layer is tasked with detecting and repairing the underlying faults as they occur (e.g., by rebooting, re-imaging failed nodes [7, 13]; and 3) application-specific code is responsible for maintaining the user-visible functionality and performance of the system.

The first contribution of this paper is its identification of a small number of building-block mechanisms that are used to implement common application-level recovery techniques for availability and data errors. These mechanisms explored in this paper include:

- **Re-execution of failed operations**, with optional variation in parameters.
- **Scheduling control** over when, where, and whether failed operations are re-executed.

Applications can combine these mechanisms to implement the most common recovery techniques that are in use today, such as optimistically continuing and resolving conflicts later; re-trying on user action; continuing with a graceful degradation of functionality; scheduling an operation to occur once a failed dependency has been recovered, or others [1, 6]. Today, without existing implementations of these mechanisms, we find that distributed systems (re-)create their own implementations of these mechanisms [8, 11, 12].

The second contribution of this paper is a language-level mechanism that we call CATCHANDRETRY, a building block for recovery techniques; in addition to providing the ability to retry, CATCHANDRETRY includes an extension of the traditional exception objects to include explicit references to the failed code.The result is an easy-to-use language mechanism that can greatly simplify the writing of recovery code. In the rest of this paper, we describe CATCHANDRETRY, and motivate and justify our design by analyzing several cases studies taken from the context of Facebook.

As a language mechanism, CATCHANDRETRY is general enough to apply any modern language with support for structured exceptions. We illustrate CATCHANDRETRY with examples of its use within both a large-scale distributed server-side application running in a data center written in C# or Java as well as a JavaScript clients-side application running within a web browser. CATCHANDRETRY is similar to the retry mechanisms proposed in other languages, such as Eiffel, Ruby, and others [3, 9]. However, our approach addresses deficiencies in these techniques, such as concisely specifying the number of retries to avoid the danger of infinite loops, handling interleaved retries of multiple types of exceptions, parameterized retries, and providing control over scheduling of retries. All these extensions have been motivated by common error handling patterns, some of which are shown in Section 3.

```
try {
    Console.WriteLine("reading file...");
    // code to read a file from the network
    ...
} catch( StalenessException e ) {
    // wait 5 seconds before retrying
    Thread.Sleep(TimeSpan.FromSeconds(5));
    Console.WriteLine("about to retry...");
    retry; // retry once
    Console.WriteLine("retried the read");
} fail {
    Console.WriteLine("retry failed");
}
```

Figure 1: Basic `retry`.

2. OVERVIEW OF CATCHANDRETRY

In building large-scale Internet services, such as search engines, web-mail services, and social networking sites, developers must contend with the challenges of data and availability errors. In this section, we describe an approach where data and availability errors are signalled via exceptions, and handled with language features for operation retries and their fine-grained scheduling.

Throughout this paper, our examples assume a single-threaded model similar to that of JavaScript, and omit the synchronization complications that would be needed in a multi-threaded system. We discuss the implications of multi-threading in Section 4.

2.1 Data and Availability Exceptions

The first element of our approach is for system and application code to raise an exception when a data error, such as staleness or inconsistency is detected; or when a remote node providing some service to the running code is unavailable or too slow in responding to a request.

Today, some exceptions, such as SQL exceptions on database queries and I/O exceptions on network errors, are already raised to signal data and availability errors. We propose expanding the use of these exceptions to include a broader set of problems, including data errors and availability error conditions that may or may not be real problems, depending on application semantics and requirements.

Our approach for *recovering* from these failures is orthogonal to the mechanisms used to *detect* potential data and availability errors. For the purposes of this paper, we assume that there is some clear mechanism, such an application-specified policy, or a strong systems indications, such as a network partition, that indicate data and availability error conditions.

2.2 Basic Retry

Once data and availability errors in a program are signaled via the use of exceptions, we propose the use of a new language-level feature, the `retry` keyword to re-execute the `try` block in which the exception was raised, as shown in Figure 1.

The `retry` keyword is only valid within a `catch` block and will attempt to re-execute the `try` block. While we shall use the C# syntax for illustration, similar syntax would be valid in other languages such as Java and JavaScript. An optional integer parameter specifies how many times to retry the block: e.g., `retry 3;`. When retry is called without an argument, `retry 1;` is assumed. Limiting the number of allowed retries prevents the accidental infinite loops that can occur in other languages with exception retries, such as Ruby [3].

2.3 When Retries Fail

Despite repeated attempts to retry a particular `try` block, there are situations when all of these attempts will ultimately fail. To process this scenario, each `retry` block can have an optional `fail` block that is processed when the specified number of retries fail. An example is shown in Figure 1. Note that `fail` "swallows" the exception; if we wish to re-throw it, we can just put `throw;` without an argument in the `fail` block.

In a way, `fail` blocks are similar to `finally` blocks. The difference is that they only execute if all retry attempts fail for a particular `catch` clause; `fail` for a block without a `retry` is syntactically invalid. Note that in the presence of multiple `retry` keywords within a `catch` block, a `fail` block will execute after the first one that is hit at runtime. Just like with regular exception semantics, another `catch` (or a `finally`) clause can be provided after the current `catch` to deal with the situation when a different type of exception is thrown.

2.4 Parametrized Retry

Albert Einstein defined insanity to be "doing the same thing over and over again and expecting different results". In a distributed system, retrying an operation *can* often lead to different results, such as when a fault was only transient, or the system is non-deterministic. Other times, however, it is useful to be able to explicitly retry a parameterized variant of the original failed operation. For this purpose, we support a parameterized version of the `try` keyword:

```
int x = 10, y = 100, z = 200;
try(x,y) {
    z = x + y;
} catch(Exception ex) {
    // retry twice with x set to 5
    retry(x=5), 2;
}
```

The parameterized `try` lists the variables in scope that can be modified when retrying an operation. When retrying an operation, new values can be specified using the C# 4.0 syntax for optional arguments. If no new value is specified then the original value is used. A `retry` cannot change the value of any variable that was not explicitly listed in the `try` parameter list.

Note that retries do not attempt to provide transactional semantics. In other words, a `retry` will re-execute the portion of the `try` block that had already run before the exception was thrown, which may lead to undesirable side-effects, as discussed in Section 5.

2.5 Recovery From Multiple Exceptions

It is common in a distributed system to have multiple failures and applications should robustly handle all of them. CATCHANDRETRY allows the application to separate the logic required for handling multiple failure modes. Consider the example below:

```
try {
   ReadData();
} catch(NetworkConnectivityException e) {
    Thread.Sleep(TimeSpan.FromMilliseconds(5));
    retry 5;
} fail {
    Console.Writeline("network failure");
} catch(StaleDataException e) {
    RefreshData();
    retry 10;
}
```

When reading data from a network, the application recovers from network failures by retrying after sleeping for a small period of time. Simultaneously, the application recovers from stale data reads by retrying after refreshing the data. At runtime, it is possible for the application to receive network-failure exceptions while recovering from stale data reads. CATCHANDRETRY automatically dispatches to the right `catch` clause in this case. Note that if exceptions are thrown by the `fail` block or the code before or after the

```
A a;
try(a){
    T;
} catch(Exception1 ex1){
    P1;
    retry 3;
    Q1;
} fail {
    F1;
} catch(Exception2 ex2){
    P2;
    retry(a=a2);
    Q2;
} fail {
    F2;
} finally{
    R;
}
```

Figure 2: Example for translation.

`retry` keyword, it will not be handled by the `catch` clauses and will need to be handled separately. In other words, the sequence of `catch`/`retry`/`fail`/`finally` clauses applies to the `try` block.

2.6 Scheduling Control Over Retries

While the ability to immediately retry a failed operation can often be useful, there are also many times when we know that a retry will not succeed until some additional condition is met. Perhaps, an operation will not succeed until a remote node has completed rebooting, or until a cached data item has been refreshed from its master copy.

To support these cases, we extend the traditional `catch` block to accept an explicit reference to a function object representing the retriable operation:

```
try {
  // do something
} catch(AvailabilityException e, RetryFunction r) {
   // allow the application to schedule the
   // try block asynchronously
   scheduleForLater(r);
}
```

The function object `r` behaves as an anonymous function capturing the functionality of the `try` block as well as a closure over its scope, as defined by the position of the `try` block. Once the `catch` block has an explicit handle to the function representing the (retryable) `try` block, it may pass it around, store it in an application-level data structures, and execute the function whenever is desired.

2.7 Implementation

These new constructs for retrying the `catch` block, scheduling retries, and handling retry failure are effectively convenient syntactic sugar that makes many common cases easier to program, as Section 3 shows. It is possible to implement them using regular language syntax, as we show below.

Figure 3 shows the translation of a `try-catch-retry-fail-finally` block shown in Figure 2. We use a local delegate to capture the `try` block `T`. Delegates are a feature of C# that effectively allows the local closure to be properly captured in the same lexical scope as the block `T`. We change the occurrences of variables `a` and `b` with `_a` and `_b` when we convert `T` into a delegate so that we can pass these in as parameters. In Java, anonymous classes can be used for similar effect.

Note that as Figure 3 shows, we keep a set of counters per single `try` block. It is possible to alternate among different kinds of exceptions without resetting retry counters; every time an exception is caught, a retry counter will be incremented. Doing so allows us to

```
// rename argument and adjust returns
bool _T(A _a=a) {T/a/_a/;}
int retryCount1 = retryCount2 = 0;
A _arg = a;
delegate void Continuation();
Continuation _cont = null;

while (true) {
   try {
      bool shouldReturn = _T(_arg);
      if (shouldReturn) {
          _cont = null;
          return;
      }
   } catch(Exception1 ex1){
      P1;
      if(retryCount1 < 3){
          retryCount1++;
          _arg = a;
          _cont = ( () => Q1 );
          continue;
      } else{
          _cont = null;
          F1;
      }
   } catch(Exception2 ex2){
      P2;
      if(retryCount2 < 1){
          retryCount2++;
          _arg = a2;
          _cont = ( () => Q2 );
          continue;
      } else{
          _cont = null;
          F2;
      }
   } finally {
      if(_cont) _cont();
      R;
   }
   break;
}
```

Figure 3: De-sugaring of the Figure 2 example in C#. We are using meta-syntax as needed (for example, `T` is actually a block of code).

place an upper limit on the number of times retry code will be run. Also note that we generally allow for code *after* a `retry` block, as indicated by `Q1` and `Q2` in our example, which can be used for local cleanup, for instance. However, since these block may never be reached, using them adds unpredictability in practice. Enforcing that a `retry` keyword be the last element of a `catch` block could be a good stylistic recommendation.

We also replace occurrences of `return` keyword within the `try` block to return to the surrounding function. To signal returns, we make the underlying delegate `_T` return a boolean value, which is handled by the caller. Note that `finally` "trumps" `return`: before returning from the function, the `finally` block will be called in C#. Once the `try` block has been implemented as a delegate function, passing an explicit handle to this function into a `catch` block is trivial. Note that an alternative, perhaps more efficient implementation for Java bytecode would use JSRs, local Java subroutines [5]; in this case, we would not have to address `returns` specially. The translation uses per-`retry` counters to capture whether we should continue the retrying process or just bail out of processing this `try` block. Depending on which `catch` block succeeds, if any, we set up and execute a continuation represented by delegate `_cont`.

3. MOTIVATING CASE STUDIES

This section presents some case studies that further motivate our

```
TimestampingMap<Person, Set< Person » friends;
TimestampingMap <Person, Message> statusUpdates;
struct Message {string msg; DateTime time};

List<Message> getStatusUpdate(
    Person person, TimeSpan staleness)
{
  List<Message> result = new List<Message>();
  Set<Person> f = friends.Get(person, staleness);
  foreach (Person p in f) {
    result.Add(statusUpdate.Get(p, staleness));
  }
  return result;
}
```

Figure 4: Status update example.

design. These case studies are taken from the context of Facebook, a widely used distributed system. For each study, we start by describing the example scenario in question and then showing a naïve implementation of it. We then proceed to refine the example over several iterations to add sophisticated exception recovery.

3.1 Facebook Status Updates

This example captures the problem of getting status updates from one's friends on Facebook.

Take 1: The initial implementation is being very conservative about staleness. All low-level calls take staleness as an argument, as shown in Figure 4. *Any* staleness violation will cause the function to throw an exception.

Take 2: In this variant, we allow individual people's status to be stale. We simply choose to skip these outdated status messages when creating a set of status updates:

```
foreach (Person p in f) {
  try {
    result.Add(statusUpdate.Get(p, staleness));
  } catch(StalenessException ex){
    ;
  }
}
```

Take 3: In the case that the *list* of friends is too stale to be used, we force a retry to re-fetch the list instead of throwing an exception to the surrounding application logic:

```
for(int i=0;i<=10;i++)
  try {
    List<Person> f = friends.Get(person, staleness);
  } catch(StalenessException ex){
    continue;
  }
  break;
}
```

The same logic can be succinctly expressed with `retry` syntax:

```
  try {
    List<Person> f = friends.Get(person, staleness);
  } catch (StalenessException ex){
    retry 10;
  }
```

Note that CATCHANDRETRY is a syntactic mechanism — we are not trying to enforce any sort of transactional semantics here. It is up to the application to undo whatever side-effects might have happened before the exception is thrown within `the` try block. Note that above, the call to `friends.Get` is idempotent, so there is no danger of duplication. Anecdotally, experience with Facebook shows that duplication failures are common: when posting a link, for example, Facebook will often show an error message and then reposting the link will result in two similar messages being posted.

```
void SendEventInvite(Person me,
  AddressFilter<string,bool> isValid, string message,
  TimeStamp staleness)
{
  foreach(Person friend in friends.Get(me, staleness)){
    string address = friend.Get("address", staleness);
    if(isValid (friend)){
      sendMessage(friend, message, "HTTPS");
    }
  }
}
```

Figure 5: Event invitation routine.

Take 4: We can catch and retry on a signal at a later point. The interesting observation is that the code above is exactly what the client code in a browser needs to execute as well. However, the amount of state maintained within a browser session is generally considerably smaller. This will lead to a situation where staleness exceptions are more common.

One option is to force an update by going to the Facebook server every time such an exception is caught. A more efficient approach, however, is to schedule an update for later execution. In the case of Facebook, a "keep-alive" request is sent to the server every several seconds. Together with such a ping, an information request can be also batched:

```
try {
  List<Person> f = friends.Get(person, staleness);
} catch (StalenessException ex){
  Aspects.registerAspect(Facebook.PingServer,
    new Task(delegate(){
      friends.Refresh(person); // refresh from server
  })).waitForCompletion();
  // just retry
  Console.WriteLine("Retryng after update...");
  retry;
}
```

This code inserts a client-side *aspect* to add operation

```
friends.Refresh(person);
```

for the current value of `person` on next `Facebook.PingServer` request. Here we assume that the machinery for batching requests is implemented by the AJAX client.

3.2 Organizing a Facebook Event

Suppose we want to send a Facebook event invitation to a list of friends. Let's further assume that we want to send an invitation to only those of our friends who live in Redmond, WA. Note that some of friends' address information might be unavailable on the current machine or not even provided by friends.

Take 1: The basic logic of sending an event is shown in Figure 5.

Take 2: Because a friend address lookup may fail, we wrap it in a `try` block:

```
try {
  string address = friend.Get("address", staleness);
  if(isValid (friend)){
      sendMessage(friend, message, "HTTPS");
  }
} catch (AvailablityException ex, 1) {
  friend.Refresh("address");
}
```

This will try to re-execute the operation to get the address of a friend after the `Refresh` is done. Note that CATCHANDRETRY will also restore the state of the stack to make sure that when the try block is scheduled to be re-run, the value of variable `friend`

```
void SendEventInvite(Person me,
    AddressFilter<string,bool> isValid, string message,
    TimeStamp staleness)
{
  foreach(Person friend in friends.Get(me, staleness)){
    try {
        string address = friend.Get("address", staleness);
        if(isValid (friend)){
        string protocol = "HTTPS";
        try(protocol) {
            sendMessage(friend, message, protocol);
        } catch(AvailablityException ex, 5) {
            friend.Refresh("address");
            int timeout = 1;
            if(ex.Timeout != -1){
                timeout = ex.Timeout;
            }
            Thread.Sleep(TimeSpan.FromSeconds(timeout));
        } fail(AvailablityException ex) {
            Console.WriteLine("5 retries failed");
        } finally {
            Console.WriteLine("Done");
        }
        }
    } catch(AvailablityException ex, 1){
        friend.Refresh("address");
        int timeout = 1;
        if(ex.Timeout != -1) timeout = ex.Timeout;
        Thread.Sleep(TimeSpan.FromSeconds(timeout));
    } catch(AvailablityException ex){
        Console.WriteLine("5 retries failed");
    } finally {
        Console.WriteLine("Done");
    }
  }
  // need to wait for all tasks to finish!
}
```

Figure 6: Retrying `SendMessage`.

is whatever it was at the time the exception occurred. In this case, we chose not to block subsequent iterations of the `foreach` loop: lookup of addresses of other friends can proceed in parallel.

Take 3: Retrying `sendMessage` is shown in Figure 6.

Take 4: We can retry the logic of the `try` block with a different set of values. For instance, assuming a `SendMessage` request fails using HTTPS, we can try switching to HTTP instead by wrapping `SendMessage` in the following way:

```
string protocol="HTTP";
try(protocol){
  SendMessage(friend, message, protocol);
} catch(AvailablityException ex){
  retry(protocol="HTTP");
  Thread.Sleep(TimeSpan.FromSeconds(5));
}
```

The call to will try to re-send the message with `HTTP` as the value of the `protocol` `try` block parameter.

4. DISCUSSION

While there are a small number of common approaches to recovering from exceptions in a distributed system. These patterns are applicable in many scenarios and, interestingly enough, similar mechanisms apply across distributed system tiers, from the client to the server tier. However, today's programming languages provide little support for these recovery patterns.

Our case studies demonstrate the feasibility and usefullnes of providing programming language support for retrying operations after failure, modifying key parameters for execution, and controlling the scheduling of these retries. While our case studies primarily use a staleness parameter to detect data errors, we can easily generalize to detect and raise exceptions to a broader set of staleness,consistency and availability issues. Similarly, while our case studies are written under the simplified model of a single-threaded or turn-based concurrency [10] execution model, CATCH-ANDRETRY is applicable to multi-threaded systems as well. However, scheduling a retry operation to execute in a parallel thread does require the developer to ensure that the operation take explicit action to ensure thread-safety and avoid time-of-check-time-of-use (TOCTOU) problems.

5. REFERENCES

[1] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP*, pages 205–220, 2007.

[3] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008.

[4] A. Fox and E. A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of HotOS*, page 174, 1999.

[5] S. Freund. The costs and benefits of Java bytecode subroutines. In *Proceedings of Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[6] J. Hamilton. On designing and deploying internet-scale services. In *Proceedings of LISA*, pages 1–12, 2007.

[7] M. Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, 2007.

[8] P. Kaczmarek, B. Krefft, and H. Krawczyk. Coordinated exception handling in J2EE applications. In *International Conference on Computational Science (1)*, pages 904–907, 2006.

[9] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[10] M. S. Miller, D. E. Tribble, and J. Shapiro. Concurrency among strangers. *Trustworthy Global Computing*, pages 195–229, 2005.

[11] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of USITS*, 2003.

[12] D. Oppenheimer and D. A. Patterson. Architecture, operation, and dependability of large-scale Internet services: three case studies. In *Architecture Specification and Implementation*, Mar. 2002.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[14] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of OSDI*, pages 21–21, 2000.