Primitives and Mechanisms of the Guardian Model for Exception Handling in Distributed Systems

Robert Miller and Anand Tripathi

Computer Science Department

University of Minnesota, Minneapolis, MN 55455

Acknowledgements:

- **This work was partially supported by NSF grants ANI 0087514 and ITR 0082215.**
- **Robert Miller thanks IBM for its support through the Graduate Work Study Program.**

Outline

1. **Objective**
2. **Exception Handling Models for Distributed Programs**
   - **Issues in Exception Handling for Distributed Programs**
3. **Guardian Model for Exception Handling**
   - **Guardian execution model**
   - **Notion of "contexts"**
4. **Examples of Exception Handling in Distributed Programs using the Guardian Model**

Exceptions in Distributed Programs

- **A distributed application consists of multiple processes executing asynchronously.**
- **A process may encounter (signal) an exception which may be independent of the state of other processes.**
- **Multiple exceptions may occur concurrently in different processes.**
  - **These may or may not be related to each other.**
- **For recovery, an exception in one process may need to be communicated to the other processes to be raised in them, and their exception handlers may need to perform coordinated recovery.**

Distributed Exception Handling Issues

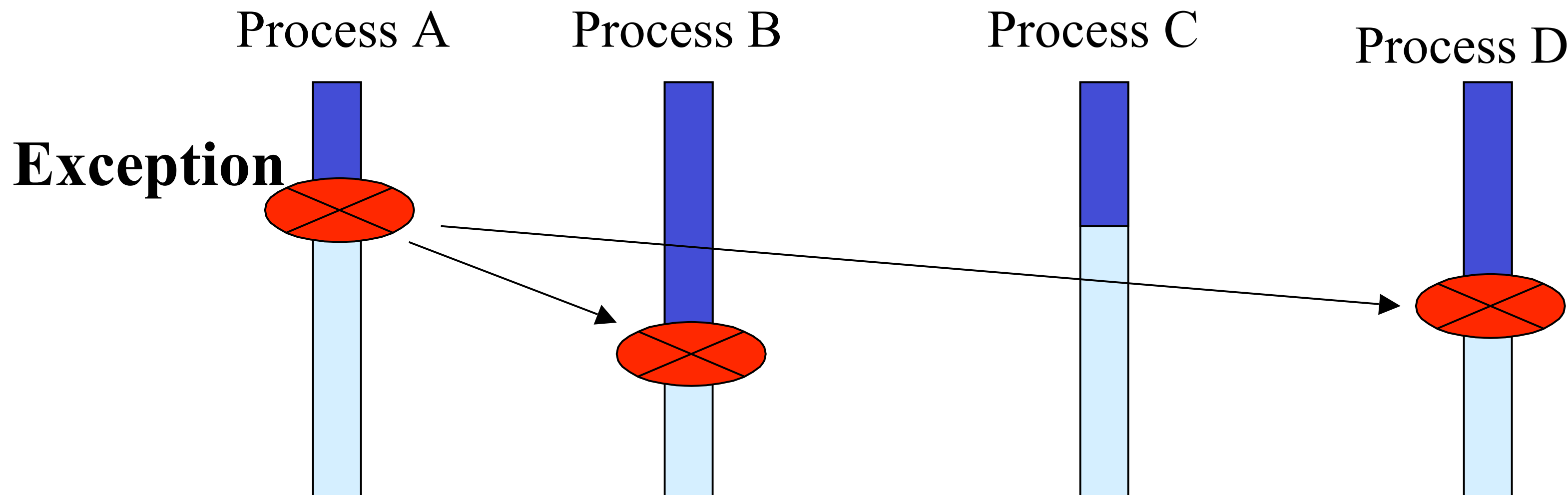
- **When an exception condition occurs in a process;**
  - **Which other processes should be informed of this exception?**
  - **Should the same exception be raised in all other processes?**
    - **Thus a process may receive an exception that is asynchronous (and unrelated) to its current execution state.**
- **When multiple exceptions occur concurrently in different processes:**
  - **Should all exceptions be resolved into one single exception?**
  - **Should all these exceptions be delivered to different processes in a fixed sequential order?**

Exception Handling in Distributed Programming

- **Exceptions in RMI based interaction models**
- **Concurrent Exception Handling Models**

Distributed Exception Handling Issues

- **When an exception condition occurs in a process;**
  - **Which other processes should be informed of this exception?**
  - **Should the same exception be raised in all other processes?**
    - **Thus a process may receive an exception that is asynchronous (and unrelated) to its current execution state.**



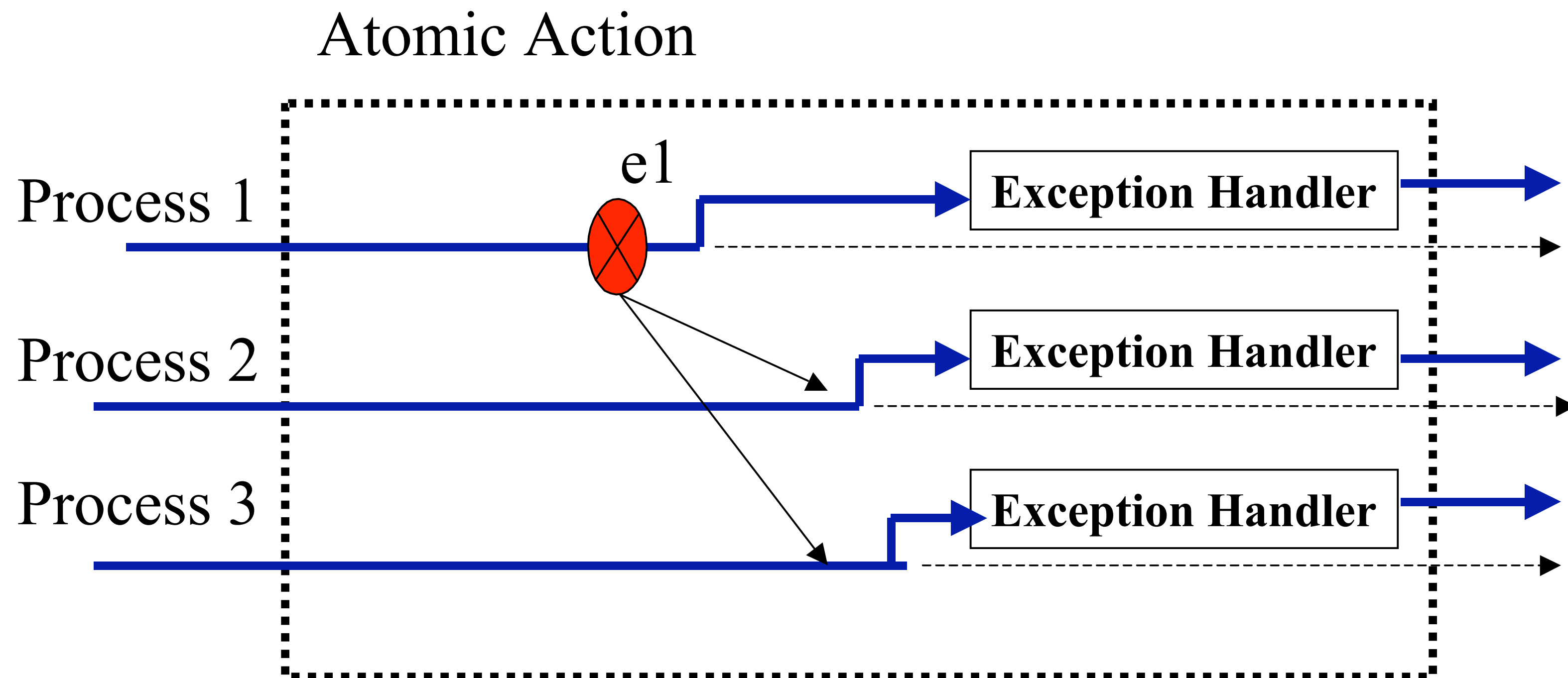
Exceptions in RMI based interaction models

- **An RMI operation may signal an exception to its invoker.**
  - **This model is adopted in Java, CORBA and other distributed programming languages and middleware.**
- **The exception occurrence is synchronous to the invoker's execution context.**
- **Exception is raised in the server and signaled to the invoker (client).**
- **Limited to synchronous interactions between two processes.**

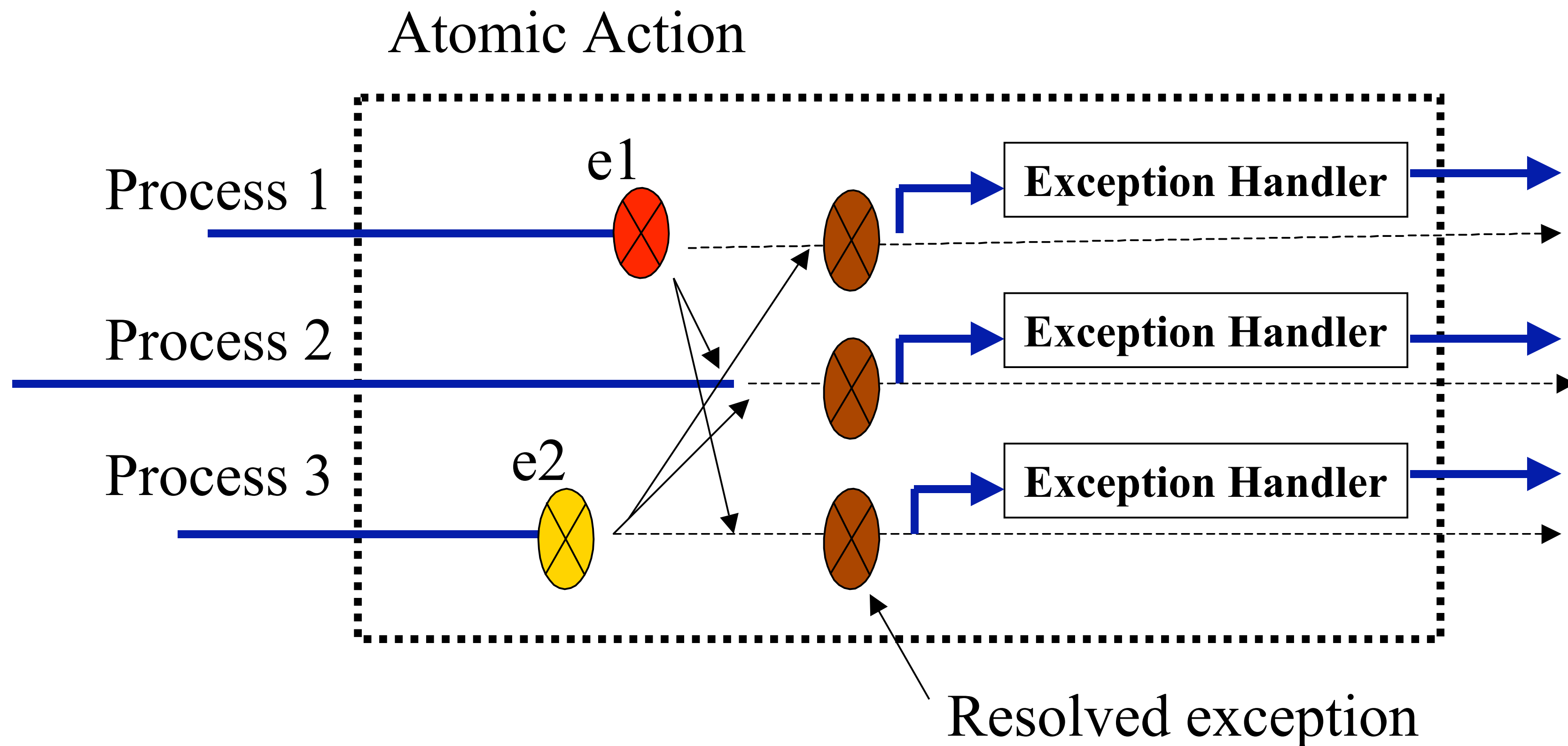
Concurrent Exception Handling Models

- **Primarily developed for conversation and transaction based interactions.**
  - **Campbell and Randell [TSE 1986]**
  - **Coordinated Atomic Actions by Xu, Romanovsky, Randell [TSE 2000]**
  - **Open Multithreaded Transactions by Kienzle, Romanovksy, Strohmeier [2001]**
- **Concept of exception resolution based on tree based hierarchy of exceptions.**
- **All processes in an atomic action handle the same exception.**

Exception Handling in Conversations



Exception Handling in Conversations



Limitations of Current Approaches

- **RMI based exception model is primarily limited to synchronous interactions between a pair of processes.**
- **Models for multiple processes are limited to programs structured using conversations and transactions.**
  - **Many applications do not conform to such structuring: parallel computing, distributed simulations, agent-based programs, monitoring systems.**
  - **Many applications require approaches other than resolution for handling concurrent exceptions.**
  - **In some cases an exception signaled in a process may need to be raised as different exceptions in other processes.**

Basic Problems

- **An exception may be signaled asynchronously in a process. The process may not be in the right context to handle the exception.**
- **Multiple exceptions may be raised concurrently in a distributed system. The current model using resolution are applicable only to conversations. A general model is lacking.**
- **Exception handler in distributed processes may need to coordinate with each other for recovery.**

For exception handling in distributed systems, each affected process must invoke the semantically correct handler.

Guardian Abstraction

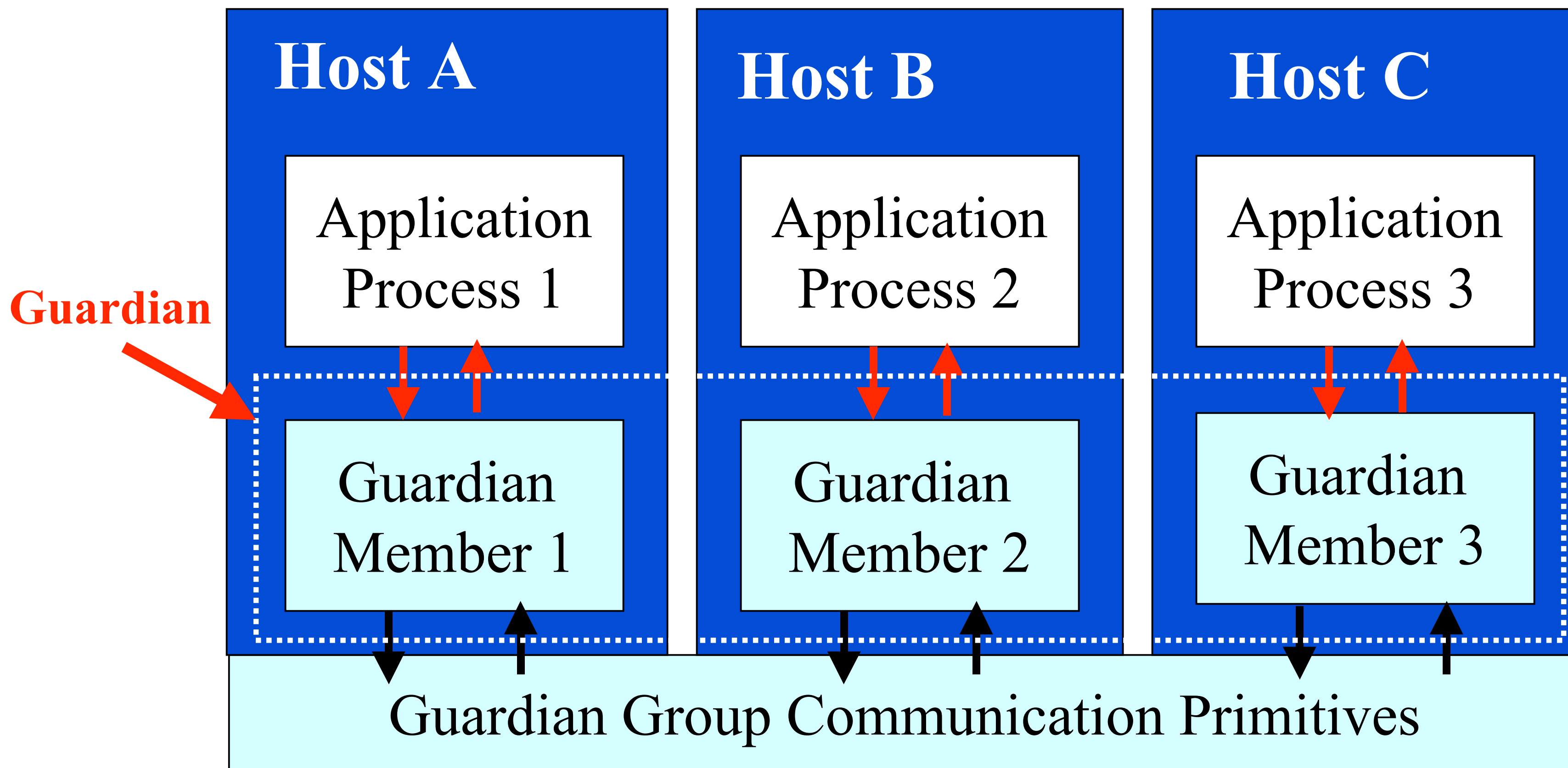
- **A guardian represents a centralized abstraction associated with a distributed application.**
- **It provides a set of primitives for global exception handling.**
- **It encapsulates the programmer-defined rules for coordinating the exception handling actions of different processes in its distributed application.**
- **It determines how an exception in one process should be signaled to other processes or how to handle concurrent exceptions.**

Guardian Abstraction…

- **Its function is to use application-defined rules to direct a process to appropriate exception handlers.**
- **It separates the global exception handling and recovery policies from the handling of local exception that can be handled internally by a process itself.**

- **It is implemented as a replicated object, using a group of processes, one member in this group corresponding to each application process.**
- **It assumes the timed-asynchronous execution model.**

Guardian Abstraction…



Guardian Model

It consist of the following elements:

1. **The concept of exception contexts and process contexts.**
2. **A set of guardian primitives that are invoked by the application processes.**
3. **Application-defined global exception handling rules.**

Global Exceptions

- **A distributed application system can define a number global exceptions, which would be raised through its guardian.**
- **Additionally, there are certain system defined global exceptions:**
  - **Membership exceptions:**
    - **Participant join and leave events**
  - **Environment exceptions:**
    - **Resource failures**
    - **Deadlocks**

Contexts

- **A context is a programmer-defined symbolic name associated with an execution phase of the process.**
- **Its purpose is to determine which processes should participate in a recovery action and which of their handlers should be executed.**
- **A context list is associated with each process.**
- **When an exception is raised in a process, a target context is specified which indicates the context in which the receiving process should handle the exception.**

Guardian Primitives

1. **gthrow (GlobalException, ParticipantList)**
   - **An application process invokes this guardian function to raise a global exception in the *specified set of processes*.**
2. **checkExceptionStatus( )**
   - **A process calls this function to check if there is any global exception for it that is pending to be raised in it.**
   - **This function raises such a pending exception, otherwise it returns without any effect.**
3. **propagate( )**
   - **This function is called by a process in an handler to determine if it has the right context to handle its current global exception. If so, the function returns false, otherwise true**

Guardian Primitives

4. **enableContext (Context, Exception List)**
   - **Using this function a process informs the guardian that it is entering a new context and it has handlers for the exceptions in the given list.**
   - **The guardian maintains for each process an ordered list of its current contexts, and for each context in the list it contains the exceptions that can be handled by the process.**
5. **removeContext( )**
   - **The most recent context is deleted by the guardian.**

Process Identifiers for Recovery Rules

- **For the purpose of recovery actions, processes (participants) of a distributed application are identified using their contexts.**

  For example, if the current context list of a process is

  C1 → C2 → C3

  Then such a process is identified as C1/C2/C3

- **A set of participants can be identified by a regular expression using contexts. For example:**

  ***/C2 matches all processes whose current context is C2.**

Recovery Rules

- **Input is one or more exceptions.**
- **Output is a list of exceptions for each process that needs to participate in recovery.**
- **Two kinds of rules:**
  1. **A set of rules to deal with occurrence of a single exception (sequential rules).**
  2. **A set of rules to deal with the situation when multiple exceptions occur concurrently (concurrent rules).**

1. **Priority table that associates an exception type with priority level**

Recovery Rules ...

- **Concurrent rules first. After all rules execute, output is one Output Sequential List (OSL) of exceptions**
- **For each exception in OSL, sequential rules applied. Output of rule appends exception to be signaled in process to Output Exception List (OEL) of process**
- **Process may have multiple exceptions on its OEL**
- **Exceptions in OEL are in priority order**

Structure of Concurrent Exception Handling

- **Input: ES -- a set of concurrent exceptions**
- **Output: OSL (Output Sequential List)**

1. **The set of exceptions is divided into subsets. Each subset corresponds to a priority level.**
2. **Exception resolutions rules are applied to each priority level, and the resulting exceptions are put in the OSL.**
3. **The exceptions in OSL are then sequentially processed by the previous rules.**

Structure of Single Exception Handling

- **Input: Exception E (Only one exception to be handled)**
- **Output: OEL (Output Exception List), for each process it contains an exception to be signaled.**

```
when signaled exception is E do {
    Let P_L be the list of participants satisfying identifier
    expression P_E;
    Let P_S be the subset of P_L satisfying predicate S;
    for each p in P_S do {
        OEL.insert (p, E_p(C_p) );
        // Exception E_p with target context C_p;
    }
}
```

Example of Guardian Application

- **Conversation Based Exception Handling**
- **Fault-Tolerant Barrier Synchronization**
- **Primary-Backup System**

Conversations

- **Conversations is well known exception handling model**
- **Guardian can simplify implementation of conversations**
  - **Conversation entry and exit are barriers**
  - **Contexts denote conversation name, similar to barriers**
- **Guardian can enhance conversations**
  - **System exceptions**
  - **Situations that can not resolve concurrent exceptions into one exception**

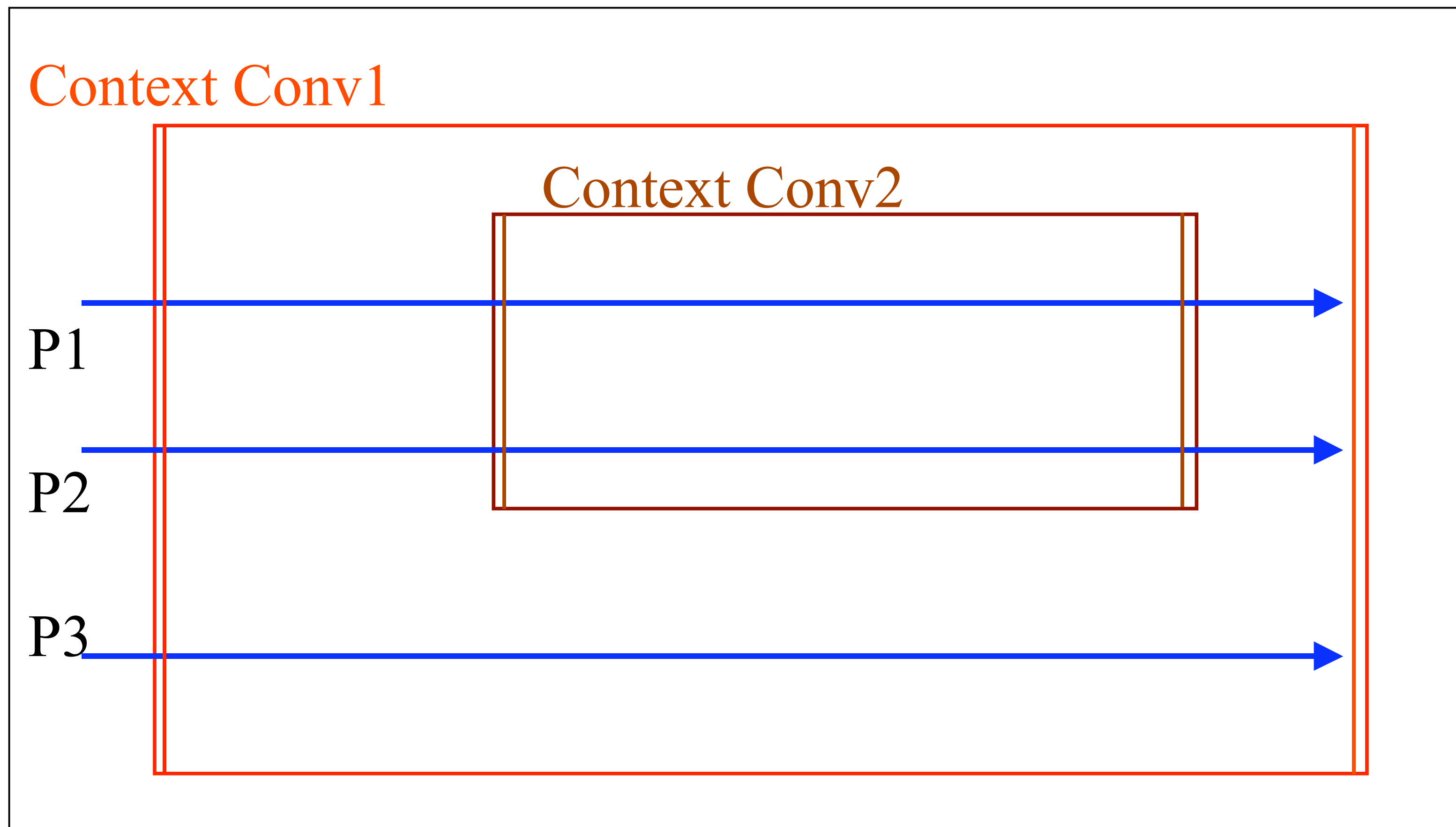
Simple Conversation Program

```
myG.enableContext("Conv", GlobalException);
try {
    myG.enableInterrupts();
     b.barrier(myG);
    myG.disableInterrupts();
    doWork();
    if (error) myG.gthrow(new GlobalException('Error');
    myG.enableInterrupts();
     b.barrier(myG);
     return;
} catch(GlobalException ge) {
    doCleanup();
}
```

Nested Conversations

- **Conversations allow nesting**
- **Exception first signaled to nested conversation. If can not be handled, then propagated to next enclosing conversation.**
- **With guardian, each conversation is uniquely named so signaling exception only to nested conversation uses target context of the conversation**
- **If exception is propagated (resignaled by conversation exception handler), then target context is next context in context list**

Nested Conversations



Lookahead Conversations

- **Lookahead significant conversation enhancement**
  - **Nested conversations do not need to be synchronized, assumes conversation succeeds**
  - **If exception occurs, back out to most recent conversation that contains all the processes**
- **Easy with guardian**
  - **Barriers only at outermost conversation**
  - **Check for exceptions at each lookahead entry or exit**
  - **Guardian rules use context resolution. Similar to exception resolution but uses contexts.**

Conclusions

- **We outlined the basic issues related to exception handling in distributed systems.**
- **We have presented here a model that allows a programmer to specify global exception handling policies in a distributed program.**
- **This model uses “contexts” to determine which processes should participate in recovery, and which handler should be used by a participating process.**
- **This model has more flexibility in handling concurrent exceptions in distributed system as compared to other existing models based on conversations or transactions.**
  - **It is more basic, and hence it can be used to implement other models.**

Simple Primary-Backup Example

- **Race conditions and asynchrony**
- **A primary-backup system with multiple processes.**
- **When a process starts, it needs to determine if primary exists. If not, then it becomes the primary; otherwise it becomes the backup.**
- **When the backup joins or restarted, the primary should know which are the available backup processes.**
- **If the primary fails, the backup with the smallest index becomes the primary.**
- **A failed primary becomes backup on restart.**

Structure of a Participant Process

```
Guardian g;
ExceptionList el = PrimaryFailed, PrimaryExists;
g.enableContext("Main", el);   //Main context starts here
role = Primary;
while (True)
  try {
    g.checkExceptionStatus();
    if (role == Primary) { primaryService( );
    } else backupService( );
  } catch(PrimaryExists) { if (g.propagate( )) throw;  role = Backup;
  } catch(PrimaryFailed) { if (g.propagate( )) throw; role = Primary;
  }
}
```

Guardian Rule

```
// Rule 1
if  (signaled exception  == Join) then {
        let p be the participant satisfying Pe = */Primary;
        OEL.insert(p, BackupJoined(*/Primary) )
        let p =  Join.signaler;
        if (guardian.groupCount > 1) then
           OEL.insert(p, PrimaryExists(*/Main) );
}
```

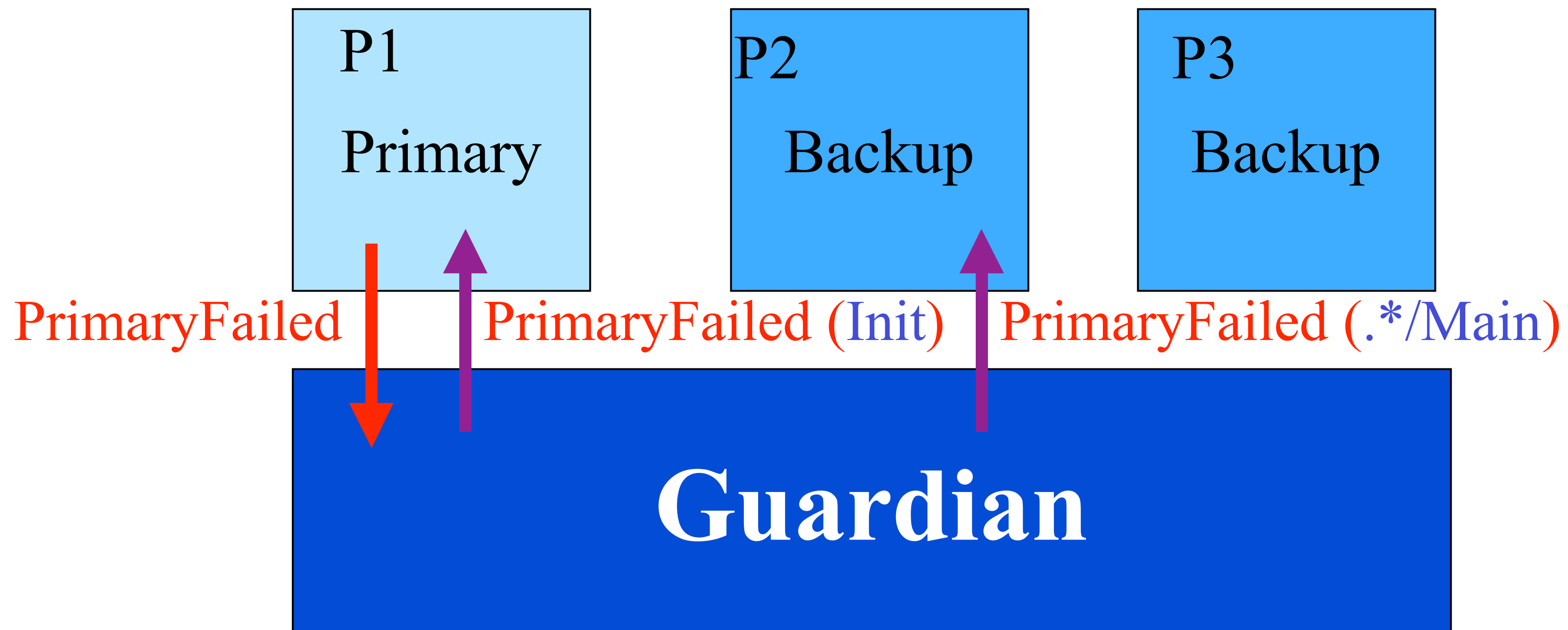
Primary Server Procedure

```
primaryService() {
  ExceptionList el = BackupFailed, BackJoined,  PrimaryServiceFailure;
  g.enableContext("Primary", el);
  Boolean backupAvailable = True;
  try { while(True)
     try {    g.checkExceptionStatus( );
              processRequest();
              if (backupAvailable) sendUpdate( );
              sendReplyToClient( );  }
    catch(PrimaryServiceFailure) {  g.gthrow(PrimaryFailed); }
    catch(BackupFailed) {  backupAvailable = False;}
    catch(BackupJoined) {  backupAvailable = True; }
  } finally {   g.removeContext( ); }
}
```

Guardian Rule

```
// Rule 2
if  (signaled exception  == PrimaryFailed) then {
  let p be the participant satisfying Pe= */Primary;
  OEL.insert(p, PrimaryFailed(Init) );
  let PL be the list of participants satisfying Pe = */Backup {
        let p =  participant with smallest index in PL;
        OEL.insert(p, PrimaryFailed(*/Main) );
     }
}
```

Primary-Backup Example



Structure of a Participant Process

```
Guardian g;
ExceptionList el = PrimaryFailed, PrimaryExists;
g.enableContext(“Main”, el);   //Main context starts here
role = Primary;
while (True)
  try {
    g.checkExceptionStatus();
    if (role == Primary) { primaryService( );
    } else backupService( );
  } catch(PrimaryExists e) { if (g.propagate()) throw e; role = Backup;
  } catch(PrimaryFailed e) { if (g.propagate()) throw e; role = Primary;
  }
 }
```

Guardian Rule

```
// Rule 3
if  (signaled exception  == BackupFailed) then {
        let p = P_s = P_L satisfying P_e = */Primary;
         OEL.insert(p, BackupFailed(*/Primary) ;
        let p =BackupFailed.signaler;
        OEL.insert(p, BackupFailed(Init) );
}
```