Fault in the Future

Ivan Lanese
Computer Science Department
University of Bologna/INRIA
Italy

Joint work with Gianluigi Zavattaro and Einar Broch Johnsen

From COORDINATION 2011

Error handling

- Unexpected, dangerous events frequently happen
  - Alien invasions not so frequent
- Unexpected events in distributed systems more frequent:
  - Client or server crash
  - Lost messages
  - Values outside the desired range
  - …
- Error handling techniques are needed for programming reliable applications in an unreliable environment
  - Errors should not cause the crash of the whole application

Our approach

- We consider the ABS language
  - Asynchronous method calls
  - Results returned using futures
- We apply to it techniques for error handling inspired by web services (e.g., WS-BPEL language)
  - Activities may fail
  - Failures are notified to interacting services
  - Failures are managed by dedicated handlers
  - Past activities may be undone

Compensations

- Perfectly undoing a past activity is not always possible
  - Sending of an e-mail
- Sometimes not even desirable
  - If you undo an hotel reservation, the hotel may want to keep part of the payment
- A compensation is a piece of code for (partially) undoing a previously terminated activity
  - Leads to a state which is not necessarily a past one
  - But it is consistent (e.g., the invariants hold)

Our approach: desired features

- We want mechanisms to notify faults
  - From callee to caller
  - From caller to callee
- We want mechanisms to compensate past method executions
- These mechanisms are needed to manage distributed errors, i.e. errors involving more than one object

Motivating scenario

- Hotel booking for the ENVISAGE meeting
- Many available hotels
- We assume that each hotel offers a method for online booking
- Possible errors
  - Booking may fail: e.g., no rooms are available (or just the server may be down)
  - Booking may be annulled: e.g., trip canceled for health reasons
    - » One should get back the money as far as possible

ABS: what our approach affects

- S ::= … (standard o-o constructs)
  - f := e!m(e_1,…e_n) (asynchronous invocation)
  - x := f.get (read future)
  - await g do {s} (await)
  - return e (return)

- g is a guard including:
  - Boolean expressions
  - Checks for future availability: ?f

Booking without error handling

- Bool bookHotel(hotelInfo info)
  ```
  {
  f := university!book(info);
  …;
  res := f.get;
  return true;
  }
  ```

- No check for room availability
- No facilities for undoing the booking

Introducing error handling in ABS

- Failures are possible both on server and on client side
- On server side
  - The invoked method may fail
    - » E.g., no rooms available at Hotel University
  - The method execution is interrupted
  - Failure notified to the caller
    - » It may thus react, e.g., trying to book a different hotel
- On client side
  - The invoking method may fail
    - » E.g., trip annulled for health reasons
  - The invocation may become useless or even undesired
    - » Don't want to pay for the hotel
  - The invocation should be annulled or compensated

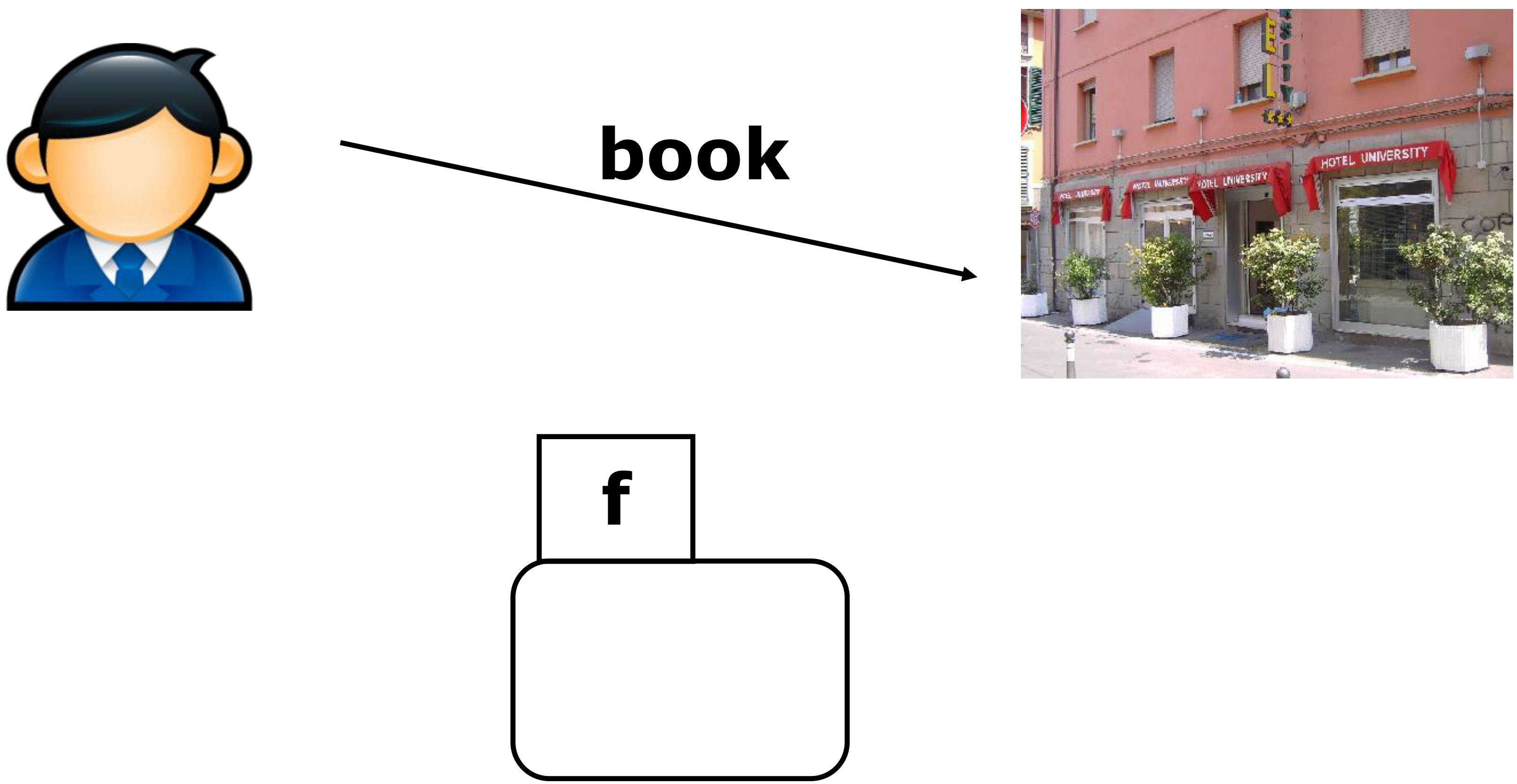
Primitives for error handling: server side errors

- **Abort** for throwing a server side error
  - Method execution is interrupted
  - Fault notification stored in the future
- **Get** is extended with **on fail** clauses
  - One for each possible fault
  - Specifying how to manage it
- Condition **?f** in an **await** guard is true
  - When the future f contains a value
  - When the future f contains a fault notification

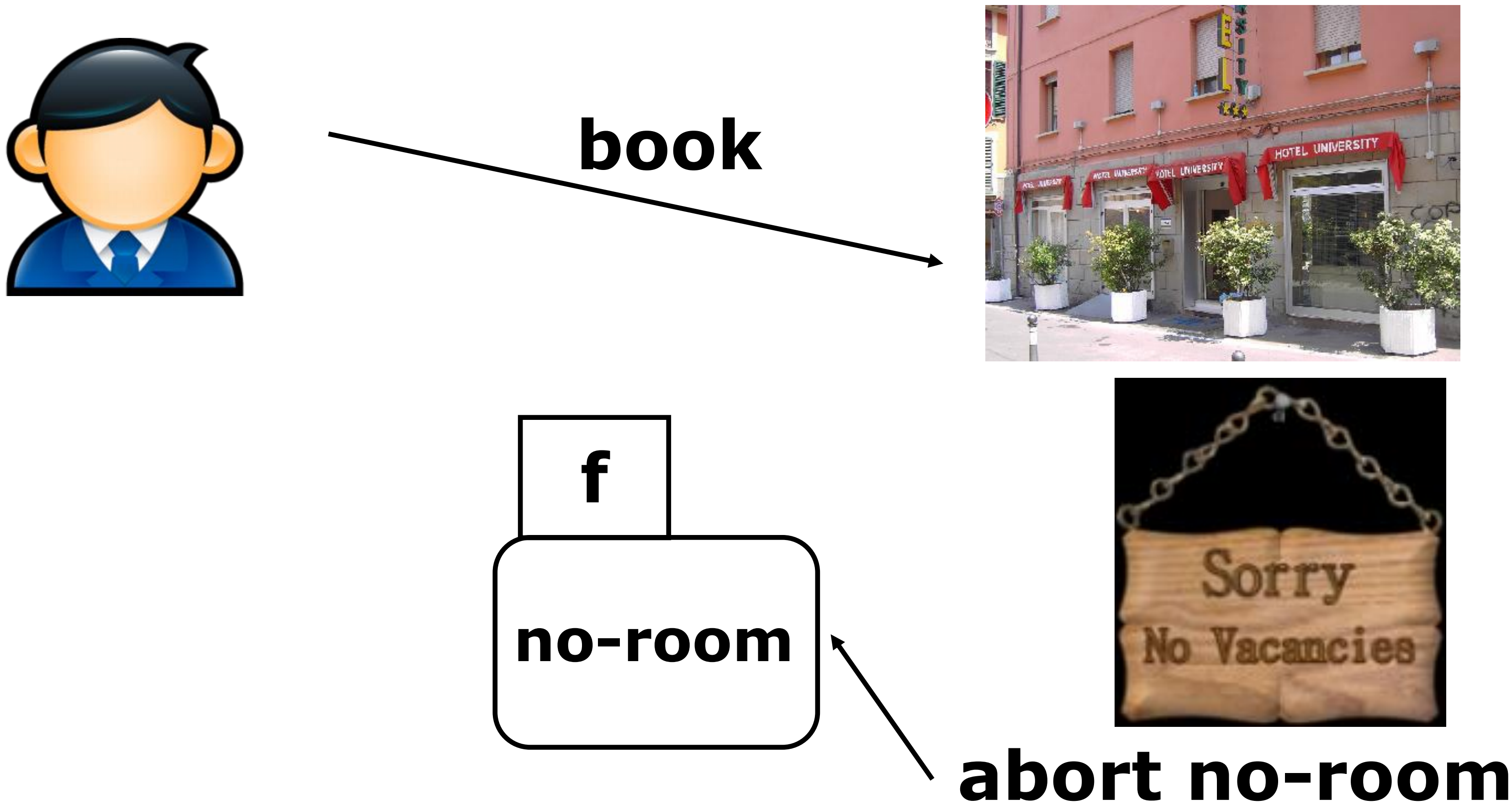
Primitives for error handling: client side errors

- **Kill** to ask to annul a method call
  - Annulled if not already started
  - (Completed and) compensated otherwise
- Compensation installed by **return** statement
  - Extended with an **on compensate** clause
  - A method may have different compensations

Server error example



Server error example



Server error example

book

on x:=f.get

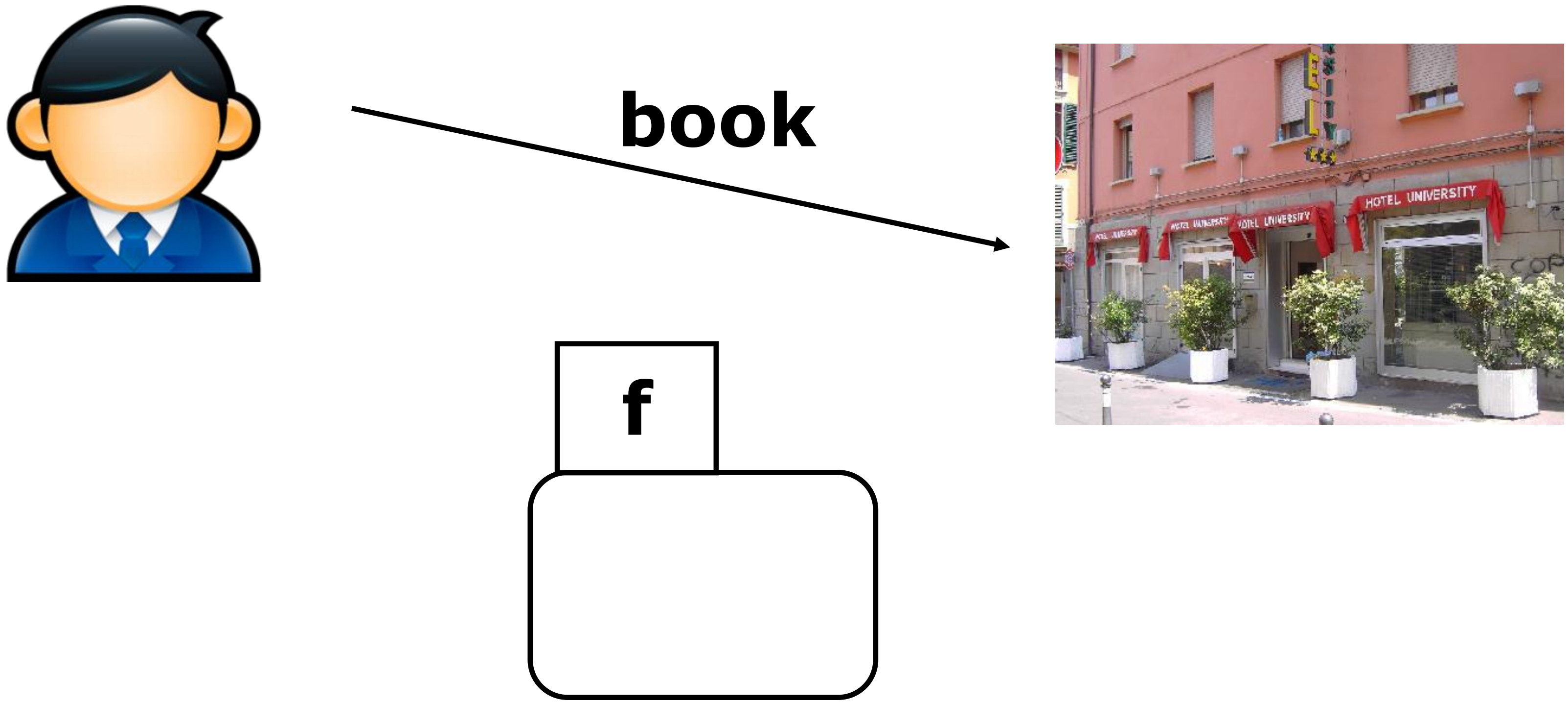
f

no-room

abort no-room

on fail no-room …

Client error example



Client error example

book

f

kill(f′)

f′:=f.kill

f′

Client error example

book

f

kill(f')

return res

f':=f.kill

on compensate

f'

Client error example

book

f

kill(f′)

return res

f′:=f.kill

on compensate

f′

v

return v

…

Extended syntax

• S ::= …	(standard ABS)
abort n	(server abort)
on x:=f.get	
do s	(getting the result)
on fail n_i s_i	
f':=f.kill	(killing a call)
return e on compensate s	(compensation def)
await g do {s}	(await)

Kinds of faults

- Programmer-defined faults
  - e.g., no-rooms
- Language faults
  - Ann: returned by **kill** when method execution has been annulled
    - » Either killed before it started
    - » Or aborted on its own before being killed
  - NoC: returned when a method that defines no compensation is asked to compensate

Example: hotel University server

- Result book(hotelInfo info)

```
{
avail := localDB.check(info);
if (avail == false) then abort no-rooms
res := localDB.update(info);
return res on compensate
      r := localDB.undo(info);
      return r;
}
```

Example: client

- Bool bookHotel(hotelInfo info)

```
{
f := university!book(info);
g := health_monitor!state("myself");
on state := g.get
      do if state == "ill" then
            f' := f.kill
      on fail error screen!print("Warning: no health
            information")
on res := f.get
      do return true
            on compensate f' := f.kill
      on fail no-rooms return false
      on fail Ann return false
}
```

Other contributions

- Full formal semantics using rewriting logic
  - Extending ABS semantics
- Extension of ABS type system
  - The client is able to manage all faults it may receive
  - Standard subject reduction and type safety results hold

Summary

- We proposed a new framework for error handling in asynchronous object-oriented languages
- It integrates asynchronous method calls, futures, faults and compensations
- Fully formalized and well-typed
- The approach is based on asynchronous invocations and futures, but does not rely on cooperative scheduling
  - It can be applied to Java

Future work: on fault model

- Adding standard language faults such as division by zero or array out of bound
  - x=y/0 behaves as **abort** DivBy0
  - One should extend the semantics of constructs raising them
- Adding system level faults
  - E.g., related to shortage of resources
  - Same effect of **abort**, but triggered by system conditions
  - On which method(s) should the fault trigger?
- Comparing/integrating with other fault models (cfr. next talk)

Future work: on analysis

- Fault model has an impact on analysis
  - Should be carefully taken into account when developing it
- Compensations should restore the invariant
  - Correctness of compensations not much discussed in the literature
  - This seems a reasonable requirement
- Fault notification may allow to preserve invariants involving more than one object
  - How to reason on such invariants?

End of talk

Thanks!

Questions?