Living Realistically with Nondeterminism in Fault-Tolerant, Replicated Applications

Joseph Slember
Priya Narasimhan

Electrical & Computer Engineering Department

Carnegie Mellon University
Pittsburgh, PA 15213-3890

Background & Terminology

- **Determinism**
  - Two entities are considered to be deterministic if, when they start from the same initial state and apply the same sequence of operations, they then reach the same final state
  - Should hold even if entities run on completely different machines
- **Why are fault-tolerant, replicated distributed applications required to be deterministic?**
  - Consistent replication is the backbone of fault-tolerance
  - Determinism results in reproducible state and behavior for a replicated component/object/process, even if replicas run on different machines
- **Determinism makes it possible to have *consistent replication***

Sources of Nondeterminism

- **System or environmental Interaction**
  - System calls that return host-specific information
    - gettimeofday(), gethostname(), …….
    - Random number generators
  - Environmental (third-party) interaction
    - Interaction with human through graphical interface
    - Interaction with shared memory, I/O, etc.
- **Scheduling/Control Flow**
  - Multithreading
  - Asynchronous Events
    - Interrupts
    - Exceptions
    - Signals

Having this kind of functionality in your application can cause problems for consistent replication

The Problem

- **To achieve consistency, the Fault-Tolerant CORBA (FT-CORBA) standard requires applications and ORBs to be deterministic**
  - *"If sources of nondeterminism exist, they must be filtered out. Multi-threading in the application or the ORB may be restricted, or transactional abort/rollback mechanisms may be used."*
- ***Effectively* forbids the use of local timers, random numbers, multithreading, shared memory, etc.**
- **End-result**
  - Real-world applications that contain these kinds of nondeterministic features cannot be made fault-tolerant!
  - ORBs are not deterministic according to these rules – thus, the concept of a fault-tolerant ORB today is not meaningful
- ***How do we get fault-tolerance while living with nondeterminism?***

Existing Options

- **Fault-Tolerant CORBA standard**
  - Applications must be “born” deterministic or they will not be supported
- **OS and virtual machine solutions [Bressoud 96/98]**
  - Lock-step synchronization of all system calls at the OS or VM levels
- **Special schedulers [Basile 03, Jimenez-Peris 00, Poledna 00, Narasimhan 98]**
  - Additional scheduler to handle multithreading-induced nondeterminism
- **Specific replication styles [Barrett 90, Budhiraja 93]**
  - Passive or semi-active replication with one leader replica forcing its nondeterministic state-snapshots onto follower replicas
- **Execution histories [Frolund 00]**
  - Uses previous invocations to make nondeterministic correction

Critique of Existing Options

- **Current approaches can be categorized as transparent or non-transparent**
  - Transparency is defined w.r.t. the application programmer
- **Transparent runtime handling of nondeterminism**
  - Doesn't change the application source code
  - Doesn't involve the application programmer
  - Forced synchronization or checkpointing at the middleware/VM level
  - Assumes that anything and everything could be nondeterministic – does not exploit application-level insight
- **Non-transparent development-time handling of nondeterminism**
  - Changes the application source code – eliminates all instances of potential nondeterminism from the code
  - Involves the application programmer
  - No need to have any additional runtime synchronization or compensation
  - Eliminates normal forms of application programming, e.g., no multithreading

Can We Improve Over This?

- **For the best of both worlds, an ideal technique would involve**
  - Runtime transparency assisted by development-time non-transparent insight while allowing application programmers to use nondeterministic calls and features in code
- **Why and how would this be beneficial?**
  - Runtime transparency – will not involve the application programmer at runtime
  - Development-time non-transparency – will target actual nondeterminism
    - Will not target potential nondeterminism that might never turn into a consistency problem
  - Allow application programmers the freedom to use current practices
  - Not exclusive to one source of nondeterminism – target all forms
- **Our interdisciplinary approach – *program analysis meets fault-tolerance***
  - Exploit program analysis at development time
    - Control flow, data flow, set-check-use methodology, code generation
  - Exploit transparent fault-tolerance infrastructure at runtime
    - Replication, total order, fault detection

Objectives of Our Approach

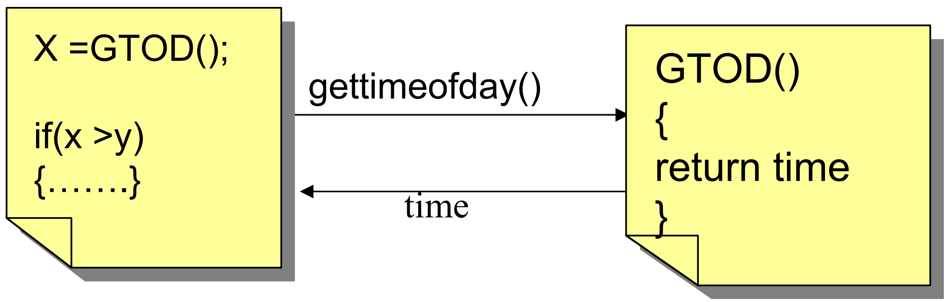
- **Allow application programmers to continue to program as before**
  - Do not need to forbid the use of nondeterministic features, e.g., multithreading
- **Categorize the different forms of nondeterminism that can be present in distributed applications**
  - Identify solutions for each category of nondeterminism and understand the cost/benefit associated with each solution
- **Targeted compensation for nondeterminism at the application level**
  - *Automatically compensating for all nondeterminism can result in significantly increased overhead*
  - Execution of a nondeterministic call does not automatically imply the need for compensation
  - Need application-level insights to determine usage and effect on system state

Program Analysis Meets Nondeterminism

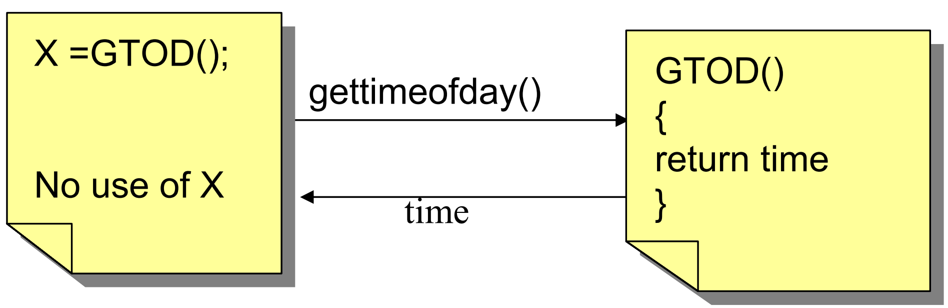
- **Take substantially proven compiler techniques and adapt them to the identification of nondeterminism**
- **At compile time**
  - Analyze source code to create compensation code in the event of nondeterminism
- ***Targeted compensation*** **– Only correct nondeterminism when it occurs**
  - Actual vs. perceived nondeterminism (next slide)
- ***Comprehensive compensation*** **– Address all forms of nondeterminism**
  - Ability to identify all nondeterminism that is known as well as future nondeterminism that may be introduced due to emerging programming techniques
- ***Deliberately not transparent***
  - Requires source code…..but the process can be automated
  - No need to rewrite application from scratch
  - Can be applied to COTS software

Perceived *vs.* Actual Nondeterminism

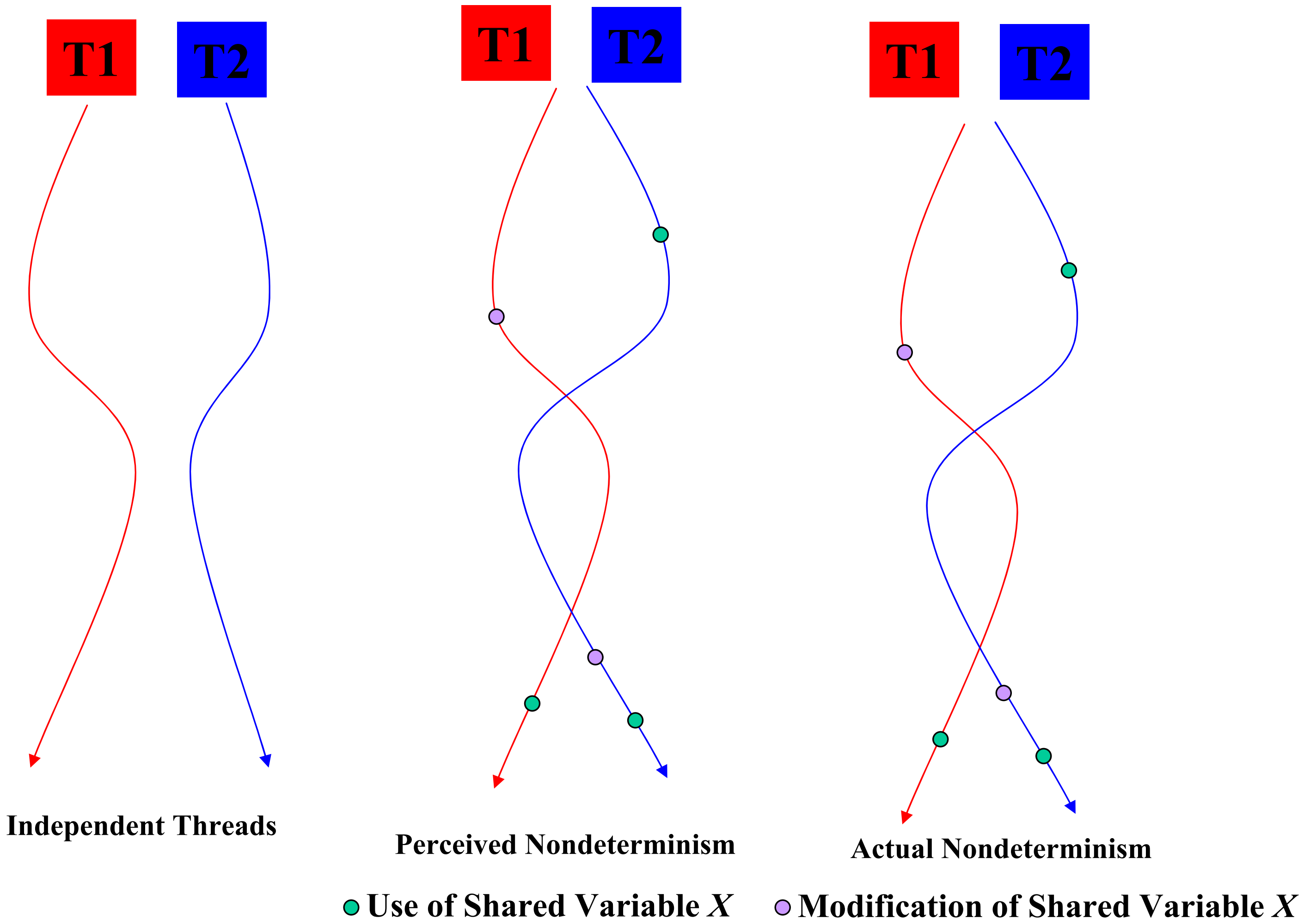
- **Actual: If GTOD is stored in a variable that is then used later, the value of GTOD has an impact on the future “slice” of the client**



- **Perceived: Value that holds nondeterministic information is never used**



Multithreaded Nondeterminism (Actual vs. Perceived)



Assumptions

- **Access to application source code to perform program analysis**
- **Runtime compensation requires underlying fault-tolerance infrastructure with specific guarantees**
  - Reliable, totally ordered delivery of messages
  - Checkpointing for the consistent retrieval and assignment of application state
  - We're using the MEAD system (http://www.ece.cmu.edu/~mead), but any system with similar guarantees will work
- **Previous Assumption:**
  - CORBA implementation (i.e., ORB) and operating system are deterministic

    Currently: We have extended our approach to perform program analysis on TinyOS as well as the MICO ORB to compensate for the ND they contain.

Development-Time Preparation Phase

- *Automatic identification of nondeterminism*
- *Automatic creation and insertion of compensation snippets*
- Program analysis to extract application-specific information and dependencies
- Discovers the actual usage (and impact on state) of nondeterministic calls
- Control-flow analysis, data-flow analysis, set-check-use methodology
- Program analysis to insert checks for consistency across invocations and compensation, if inconsistency is determined
- Can involve the application programmer at development time (indirect benefit: programmer education in fault-tolerance issues)

Two Distinct Analyses

- **System/Environmental Interaction**
  - Track all function and system calls
  - Track state that passes through these calls
  - Store nondeterministic state information at runtime
- **Scheduling/Control Flow**
  - Track all launches of threads
  - Determine all possible thread interweaving
  - Store nondeterministic information as threads execute
- **Both of these solutions are implemented**

Runtime Compensation Phase

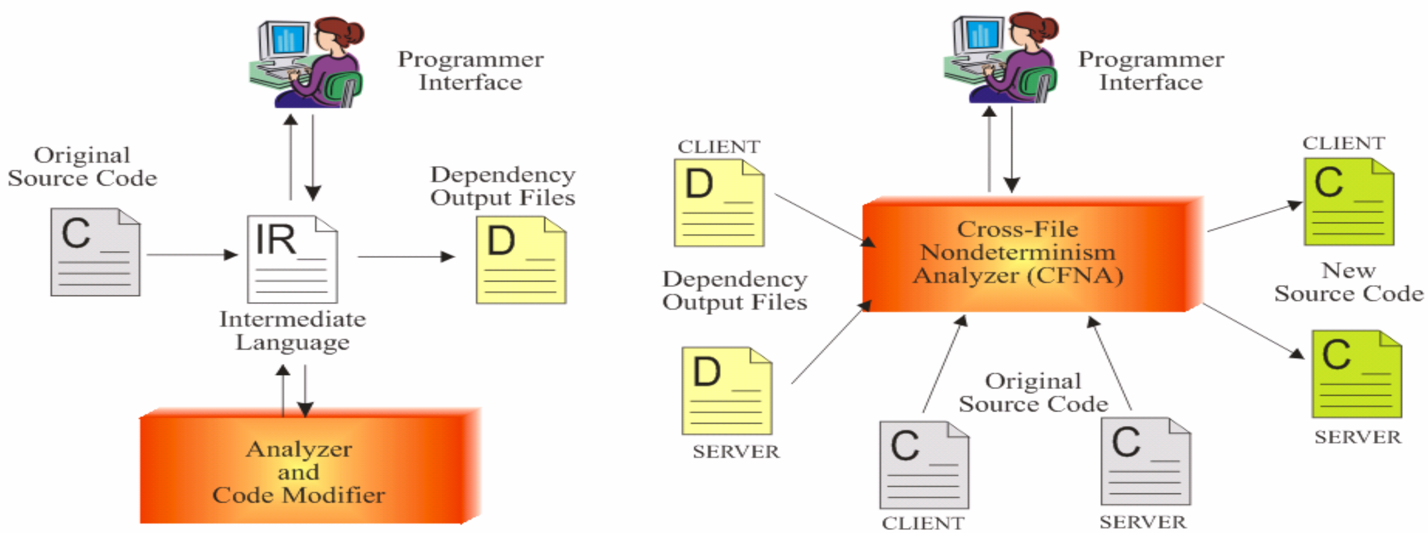
- Checking conditional to see if state is inconsistent
- Piggybacking of sufficient nondeterministic information and compensation information
- *Execution of compensation snippets*
- Saving of local nondeterminism
- Does not involve the application programmer at runtime
- Current focus on handling distributed CORBA applications
  - Approach can be easily extended to non-CORBA applications, too

Combined Development-Time & Runtime Phases

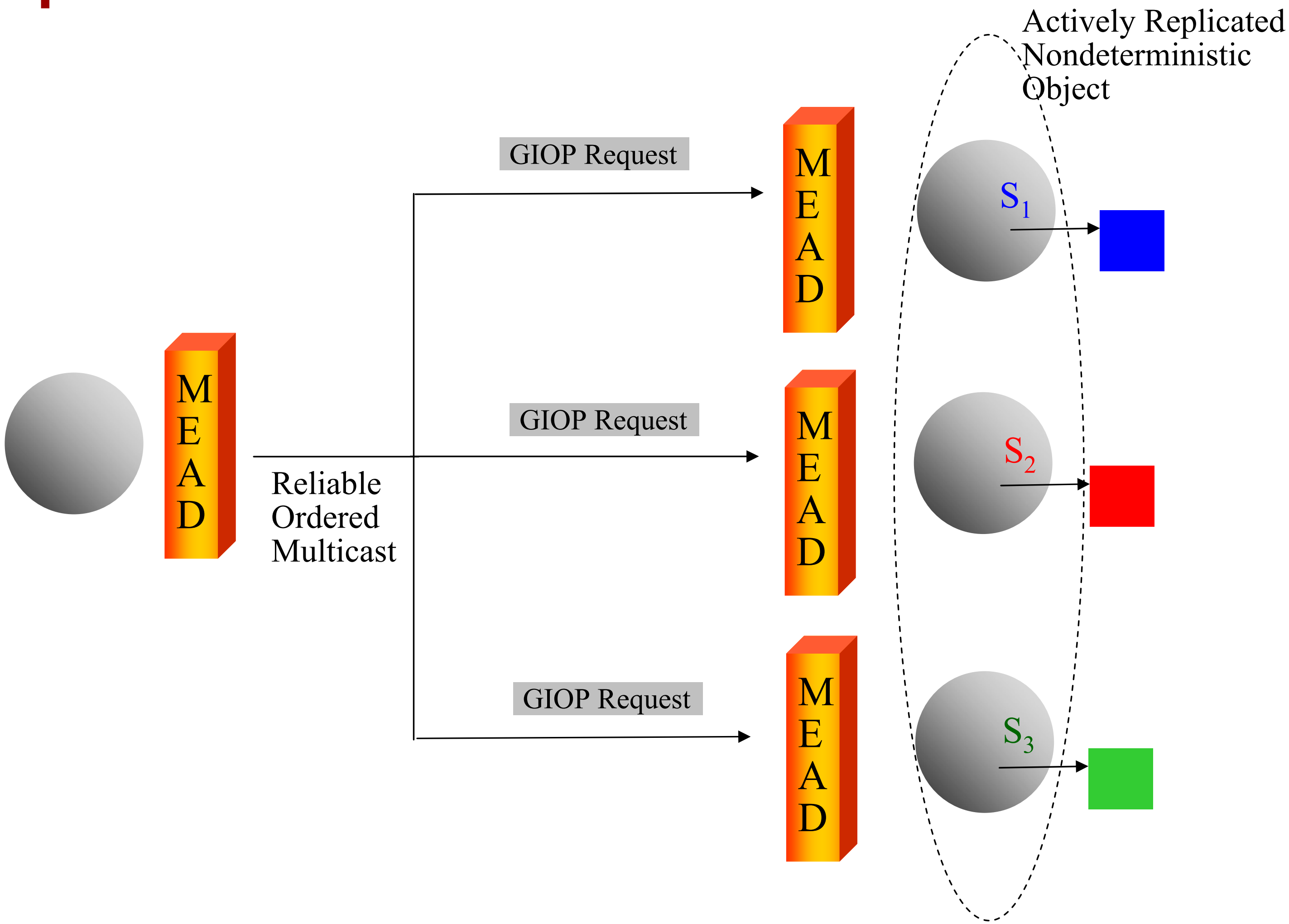
- **Client sends out a request to a replicated object running on different nodes**
- **Each replica receives the request and sends its own reply**
  - Saves local nondeterministic information
  - Passes back to client a message with prepended nondeterministic decisions
- **Client invokes replicated server again, this time prepending previous received nondeterministic values**
- **Each replica compares the prepended information and executes a compensation snippet, if mismatch exists**
- **After processing the current invocation, the replicas are consistent for all past invocations except the current one**
- **Amount of nondeterministic state does not increase with number of invocations**

Implementation Details

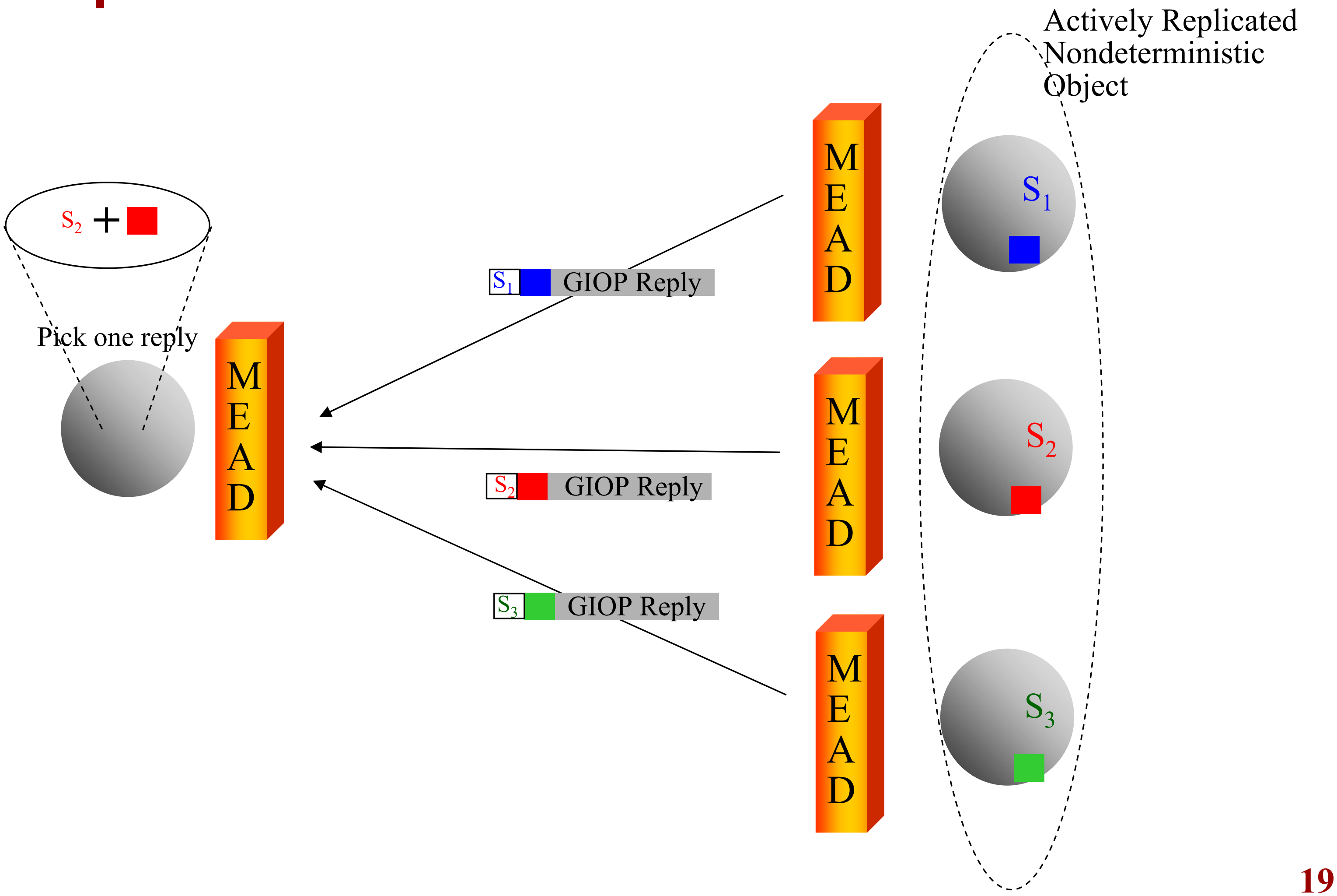
- **Stage I**
  - Automatically convert source code to intermediate language
  - Automatically compute external dependencies
- **Stage II**
  - Combine and resolve external dependencies across entire application
  - Modify source code to handle nondeterministic information.
  - Generate new application source code



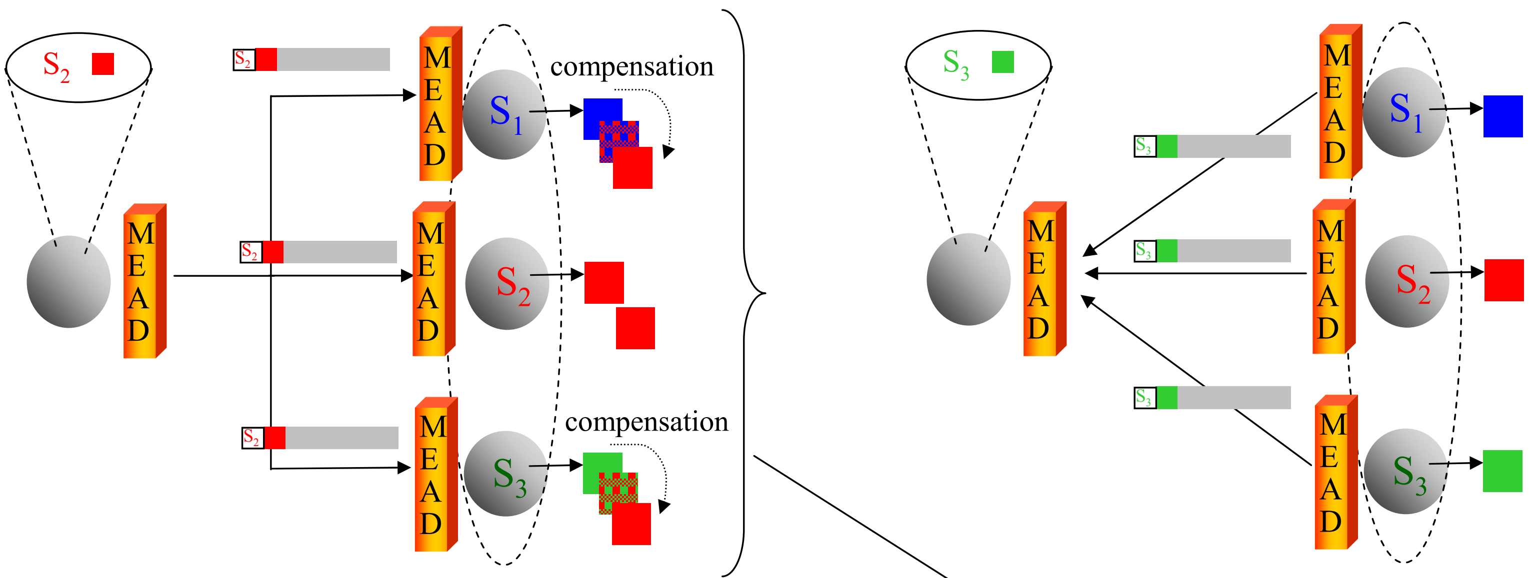
Implementation Details



Implementation Details



Implementation Details



Compensation ensures replica consistency

Test Application

- **Nondeterministic Application**
  - Invokes local timer
  - Calculates how many cycles the processor has gone through since last invocation
  - Stores local clock time

```
CORBA::Long Time_impl::get_cycles()  throw (CORBA::SystemException)
{
  time_t time_now = time(0);
  struct tm * time_p = gmtime(&time_now);
  time_p->tm_hour += (24 + this->time_zone_st);
  time_p->tm_hour %= 24;
  long cycles = ( ( (time_p->tm_hour - this->past_tod.hour) *3600) +
(time_p->tm_min - this->past_tod.minute*60) + (time_p->tm_sec - this->past_tod.second) * 18000000);
  this->past_tod.hour = time_p->tm_hour;
  this->past_tod.minute = time_p->tm_min;
  this->past_tod.second = time_p->tm_sec;
  return cycles;
}
```

Test Application Compensation

- Test Condition
- Compensation
- No Compensation

```
TimeTransfer::NonDetStruct Time_impl::get_cycles_nondet_corr(const TimeTransfer::NonDetStruct & nd_pass)
  throw (CORBA::SystemException)
{
  time_t time_now = time(0);
  struct tm * time_p = gmtime(&time_now);
  TimeTransfer::NonDetStruct tod;
  tod.sid = this->sid;
  tod.time = time_p;
  if(this->sid != nd_pass.sid)
  {
     int sec_diff = ((nd_pass.hour - this->past_tod.hour) *3600) + (nd_pass.minute - this->past_tod.minute*60) +
     (nd_pass.second-this->past_tod.second);
     tod.cycles = (((((tod.hour - this->past_tod.hour) *3600) + (tod.minute - this->past_tod.minute*60) +
     (tod.second-this->past_tod.second)- sec_diff))*18000000);
    this->past_tod – time_p;
    return tod;
  } else
  {
    tod.cycles = (((tod.hour - this->past_tod.hour) *3600) + (tod.minute - this->past_tod.minute*60) +
     (tod.second-this->past_tod.second)*18000000);
    return tod;
  }
}
```

Current Contributions of Approach

- **Demonstrated ability to handle nondeterminism**
  - Without hampering application programmer's ability to use programming practices
  - With sufficient application-level insight through program analysis
- **Differentiated between perceived and actual nondeterminism**
  - Allows for targeted and more efficient compensation
  - Novel contribution – this distinction has not been made before
- **Technique applicable to both middleware and applications**
  - Applied this to identify and compensate for nondeterminism in applications
    - Quantified reasonable overheads [SRDS 2004]
  - Applied this to identify nondeterminism in off-the-shelf ORBs
    - Yes, it turns out that ORBs themselves can be nondeterministic, too!

Current & Future Directions

- **Ongoing focus of nondeterminism compensation**
  - Multithreading
  - Asynchronous signals
- **Further experimentation**
  - Multiple clients
  - Multiple tiers
  - Increased number of replicas
  - Validation of consistency and correctness
- **Future extensions of this approach**
  - Checkpointing
    - Use program analysis for more efficient checkpointing schemes
  - Network partitioning
    - Treat this problem as similar to nondeterminism
  - Security
    - Use program analysis to differentiate between nondeterminism and malice

Conclusions

- **Novel approach to handling nondeterminism**
  - Exploiting program analysis to identify nondeterminism
  - Categorizing the different forms of nondeterminism
  - Runtime compensation for nondeterminism
- **Benefits**
  - Compensates for actual (and not perceived) nondeterminism
  - Programmer free to continue to program and use standard techniques
  - Incorporates application-level insight for targeted compensation
  - Not focused on only one kind of nondeterminism
- **Next steps**
  - Increased experimentation, catalog of solutions for every form of nondeterminism, support for multi-tier multi-client distributed applications

For More Information

Joe Slember
Electrical & Computer Engineering Dept.
Carnegie Mellon University
Pittsburgh, PA 15213-3890
jslember@ece.cmu.edu
www.ece.cmu.edu/~jslember