Handling Nondeterminism in Multi-Tiered Distributed Systems

Joseph Slember
Priya Narasimhan

Electrical & Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA

Motivation

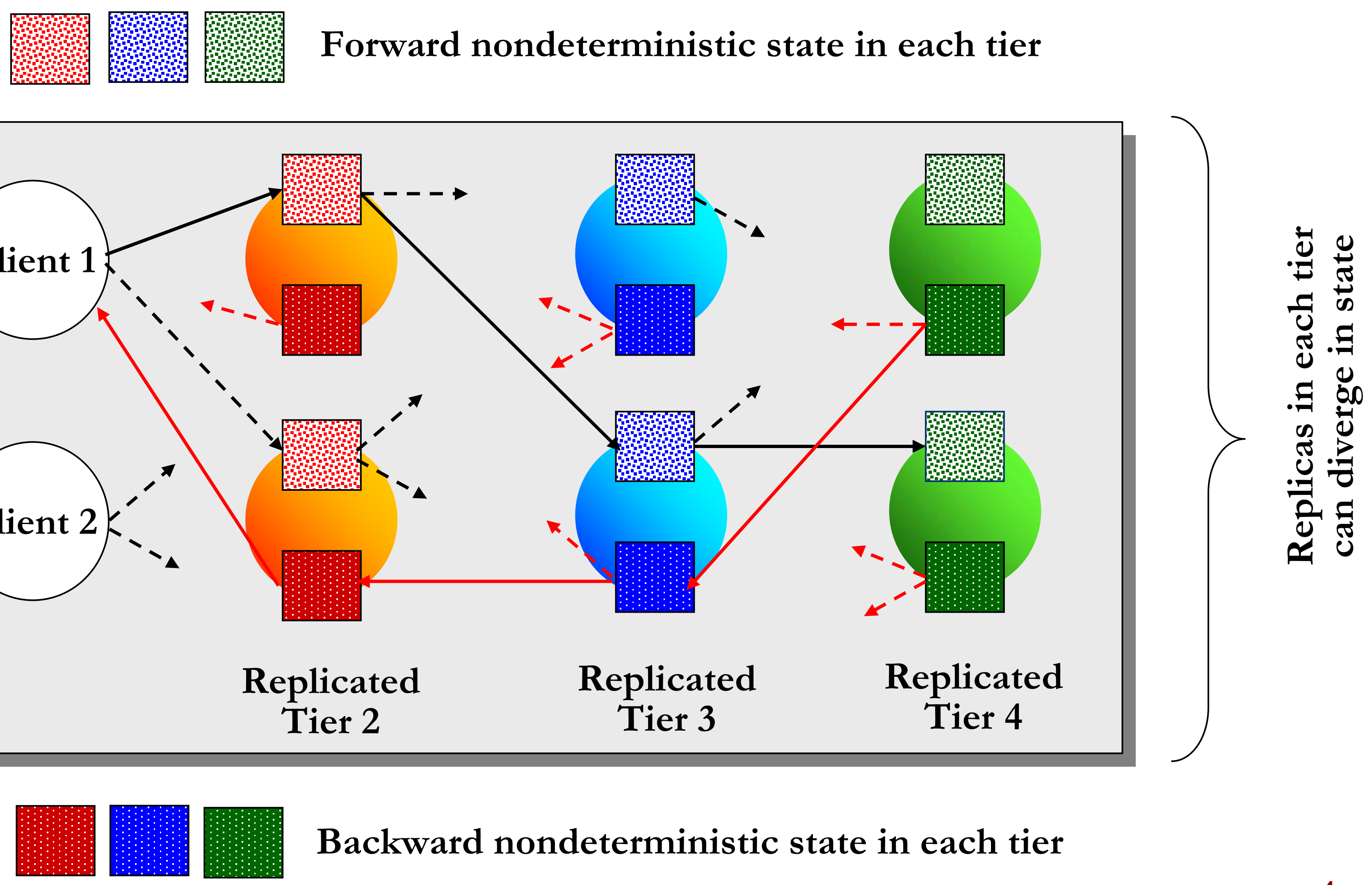
- Consistent state-machine replication requires determinism
  - Any two deterministic replicas should reach the same final state if
    - They start from the same initial state *and*
    - Execute the same ordered sequence of operations
  - Even if the replicas run on completely different machines
- Challenges
  - Many primary (first-hand) sources of nondeterminism
    - System calls, multithreading, ……
  - Nondeterminism can "propagate" through invocations and responses in a distributed multi-tier, multi-client application
- Research question
  - How do we live with nondeterminism in a *multi-client*, *multi-tier* distributed system, without compromising replication?

The Problem

- Multi-tier setting
  - End-to-end operation spanning all (server) tiers
  - Client ⇆ Server 1 ⇆ Server 2 ⇆ ………….. ⇆ Server *n*
- Forward (downstream) path of invocations
  - Client → Server 1 → Server 2 → ………….. → Server *n*
- Backward (upstream) path of replies
  - Client ← Server 1 ← Server 2 ← ………….. ← Server *n*
- Nondeterminism in any tier can "contaminate" other tiers
  - *Forward nondeterminism* – on the invocation path
  - *Backward nondeterminism* – on the reply path
- Multiple clients can aggravate this further
  - Clients' operations can intermingle and execute concurrently at each tier

Just How "Ugly" Can It Get?

Or the Multi-Tier, Multi-Client Problem



Objectives

- *Consistent server replication* in the face of
  - *Any* kind of nondeterminism at a server tier
  - *Forward* propagation of nondeterminism across tiers
  - *Backward* propagation of nondeterminism across tiers
  - *Multiple clients* causing concurrency side-effects at server tiers
  - *Failures* (loss of a replica) at any of the server tiers
- *Efficiency* in addressing only the nondeterminism that matters
- *Programmer intent* must be respected
  - Retain the application-level semantics that the programmer desires
    - Example: Uphold any concurrency programmed into the application

Our Approach

- Midas: Synergistic combination of
  - Compile-time analysis with runtime compensation
- Compile-time static analysis
  - (Currently) targets application-level nondeterminism
  - Requires access to application source-code
  - Flags nondeterminism that will cause replica divergence
  - Tracks the propagation of nondeterminism
  - Inserts code to perform compensation
- Runtime compensation
  - Two possible techniques to restore consistency
  - Transfer of nondeterministic checkpoints
  - Re-execution of inserted code

Taxonomy of Nondeterminism – I

Pure (or first-hand) nondeterminism

- Originating (primary) source of nondeterministic execution
- `random(), gettimeofday(),` ….
  - Must directly touch the persistent state that matters for replication
- Shared state among threads

Contaminated (or second-hand) nondeterminism

- Persistent state that has any dependency on pure nondeterministic state
- Example

```
for (int j = 0; j < 100; j++ ) {
     foo[ j ] = random();
     bar[ j + 100 ] = foo[ j ];
}
```

Taxonomy of Nondeterminism – II

Superficial nondeterminism

- Potentially nondeterministic execution that does not ultimately lead to divergence in persistent state across replicas
  - Nondeterministic functions that do not touch persistent state
  - System calls that appear to be nondeterministic but do not affect consistent replicated state, upon further examination
  - “Shared” state between threads, where each thread only operates on its individual and distinct piece of the state

Superficial nondeterminism does not matter for consistent replication!

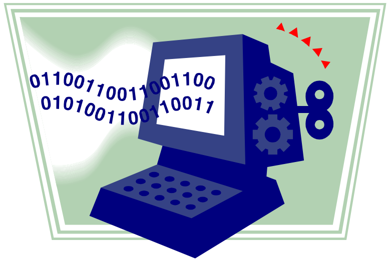
Pure determinism

- Persistent state that has neither any dependency on pure nondeterminism nor represents pure nondeterminism in itself

```
for (int j = 0; j < 100; j++ )
bar[ j ] = bar[ j ] + 10;
```

Midas' Static-Analysis Framework – I

- Front-end of a compiler
- Source-code analyzer and regenerator
- Control-flow and data-flow analyses to determine the extent to which nondeterminism has pervaded the application code
- Custom-built for analyses of various kinds
  - Nondeterminism analysis – presence/type/amount of nondeterminism
  - Concurrency analysis – thread-level interactions and interleaving
  - Dependency analysis – dependencies across clients/servers
    - Forward nondeterminism
    - Backward nondeterminism



Midas' Static-Analysis Framework – II

- (Currently) works for C, C++ and Java distributed applications
  - Converts all source-code to annotated intermediate representation
  - Similar to an AST (abstract syntax tree)
  - Intermediate representation is amenable to our analyses

- "Nondeterminism dictionary"
  - 262 system calls
    - `read`, `write`, `gettimeofday`, etc.
  - 163 library functions within C/C++ standard I/O, memory and machine-dependent OS libraries

Midas for Multi-Tier Architectures

- Midas' program analysis used to analyze the architecture
  - To extract dependencies between tiers
  - To extract effects on state within each tier
- Architecture across tiers broken down into *compensation-tier pairs*
  - Consider each tier in conjunction with its immediate communicating tiers
  - Compensation of nondeterminism can then be performed in a scalable way
- Architecture at each tier broken down into *tier-centric slivers*
  - Consider execution within each tier in terms of blocks ("slivers") of code
  - Each sliver encapsulates a basic unit of forward/backward nondeterminism at that tier
  - Allows for easier compensation

Tier-Centric Slivers

- Forward sliver
  1. An incoming request from an upstream tier
  2. Some post-request processing that might lead to execution and state changes
  3. An outgoing (nested) request to some downstream tier
- Backward sliver
  4. Incoming replies for requests sent in the previous step
  5. Some post-reply processing that might lead to additional execution and state changes
  6. An outgoing reply to the upstream tier that issued the request in step 1
- Possible nested behavior where steps 3, 4 and 5 repeat
  - Yields multiple forward slivers and one backward sliver

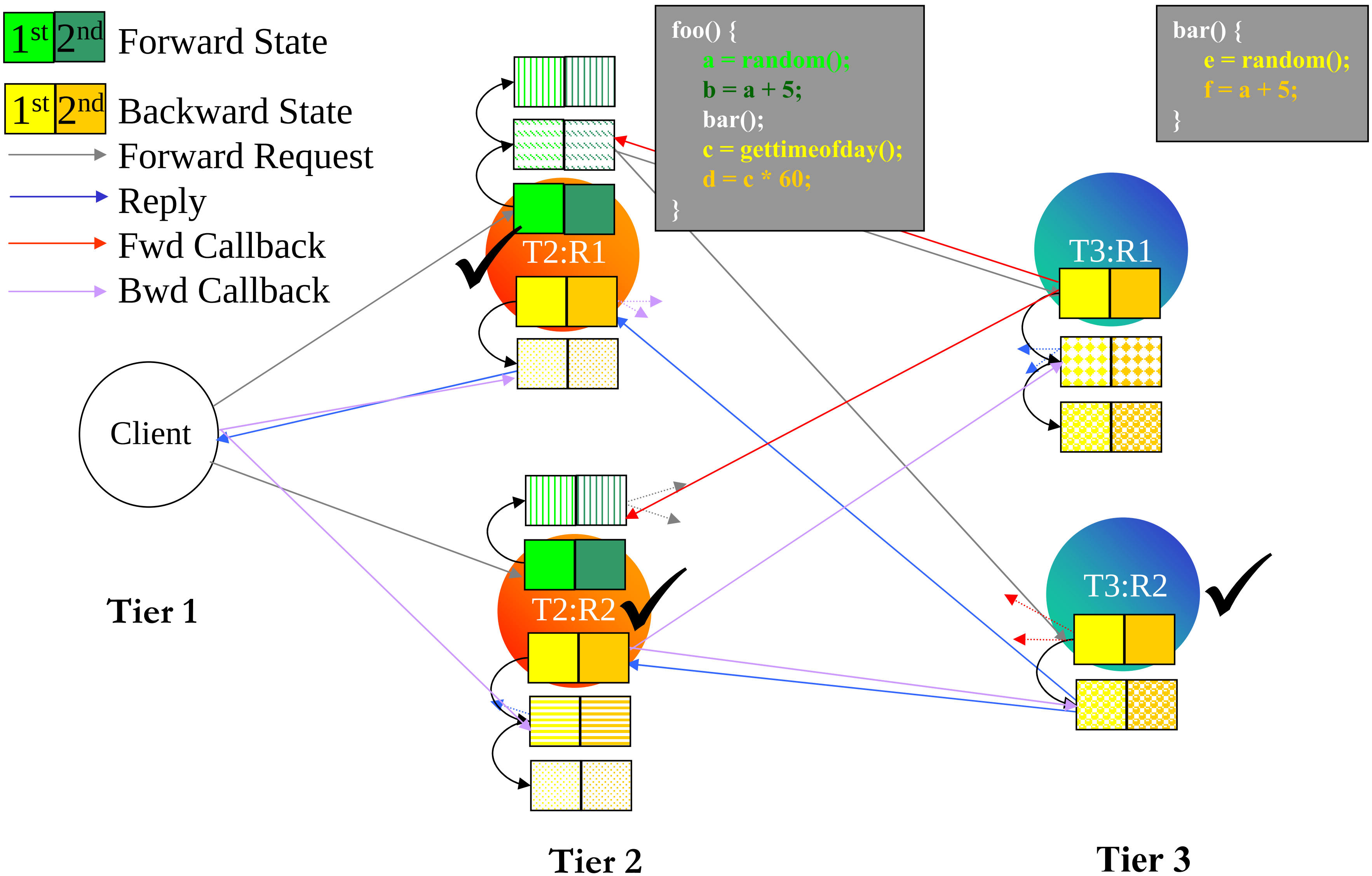
Compensation Tier-Pairs

- Replicas in each tier need to know which state is actually used by the adjacent tiers with which they communicate
  - If the replicas of tier A make a downstream request to tier B, which replica's request was chosen by tier B?
- Consider an operation C ⇆ T1 ⇆ T2 ⇆ T3 ⇆ T4
  - Possible compensation tier-pairs: (C, T1), (T1, T2), (T2, T3) and (T3, T4)
  - A tier can be in more than one pair, e.g., tier T2
- Group into forward and backward compensation tier-pairs
  - Forward compensation tier-pairs encapsulate forward slivers' communication
  - Backward compensation tier-pairs encapsulate backward slivers' communication

Midas' Compensation Techniques

- Technique #1: Checkpoint-to-compensate
  - Track all first-hand and second-hand nondeterminism
  - Nondeterministic checkpoint consists of the tracked information
- Technique #2: Reexecute-to-compensate
  - Track only first-hand nondeterminism
  - Execute inserted code to regenerate second-hand nondeterministic state, given the tracked (first-hand) information as input
- Totally ordered, reliable multicast messages between tiers
- How does compensation happen at runtime?
  - Tier T1 issues a request to Tier T2
  - T2's replicas track nondeterminism and piggyback it to reply to T1
  - T1 sends an asynchronous callback to T2's replicas with choice of T2 replica and that replica's nondeterminism
  - T2's replicas copy received nondeterministic information onto their state
  - Re-execute, if technique #2 is being used; otherwise, nothing to do

Putting It All Together



Conclusion

- Midas: Inter-disciplinary approach to handling nondeterminism
  - Synergistic combination of compile-time analysis with runtime compensation
  - Intentionally non-transparent
- For multi-tier distributed software architectures
  - Replica consistency in the face of "propagating" nondeterminism
  - Forward and backward nondeterminism
  - Compensation-tier pairs
  - Tier-centric slivers
- Next steps
  - Deploy and evaluate with a real-world, multi-tier application
  - Determine scalability with number of tiers and number of clients
  - Determine performance of various compensation techniques

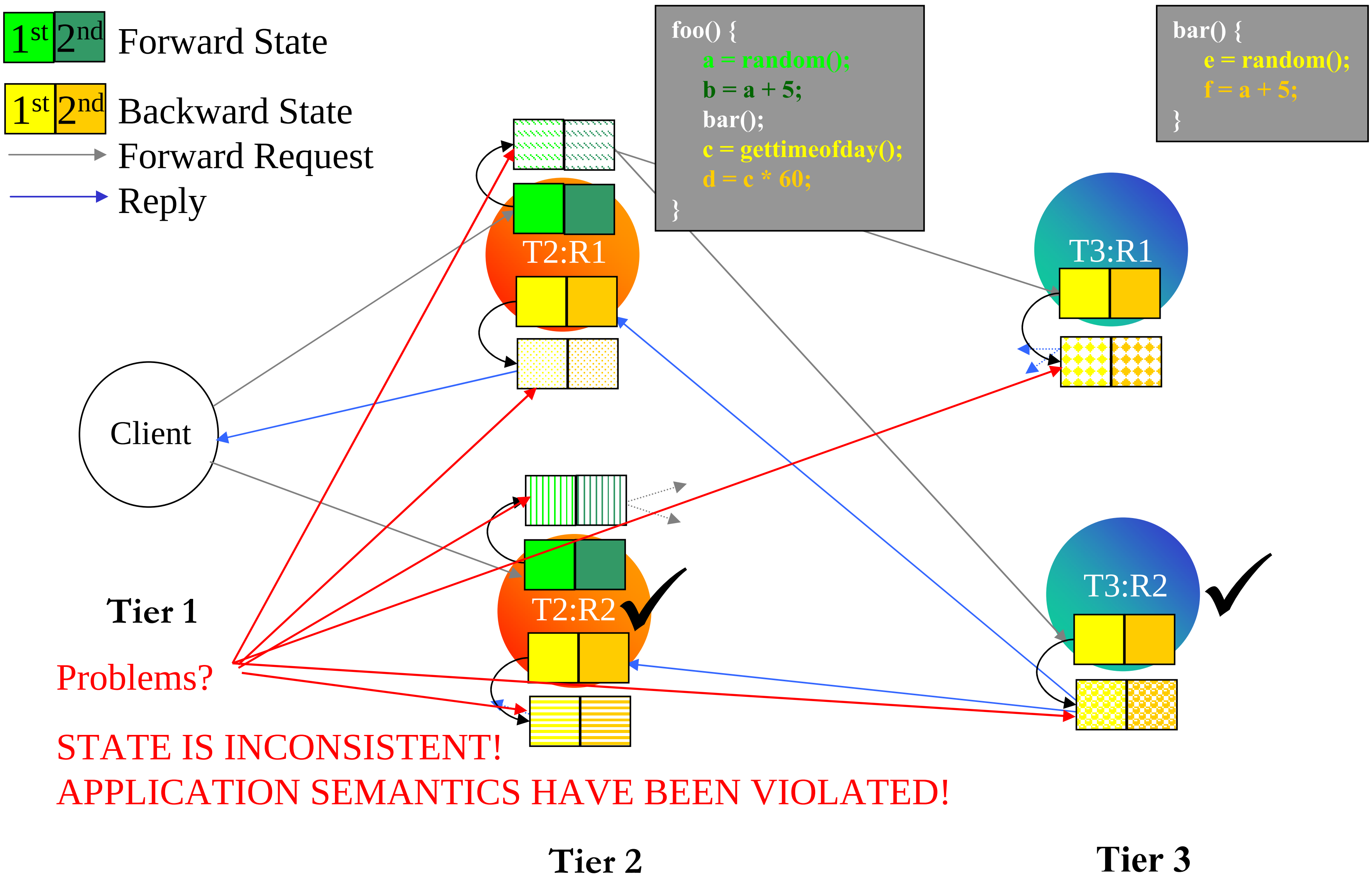
Joe Slember
jslember@ece.cmu.edu
www.ece.cmu.edu/~jslember

Extra Slides

Midas' Source-Code Modifications

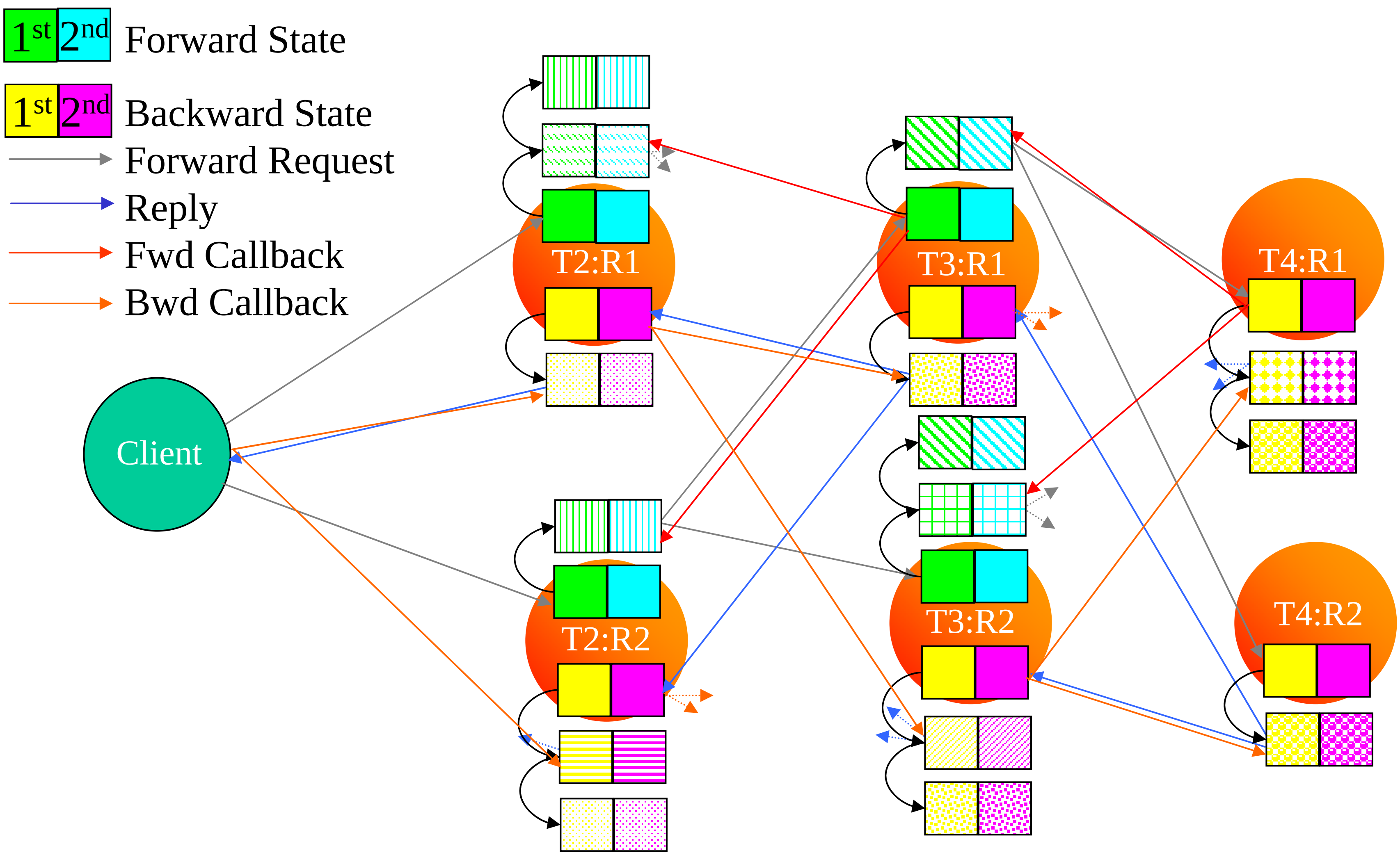
- Data structures added to store results of nondeterministic actions
  - What is stored depends on the compensation technique
    - Store first-hand nondeterministic state OR
    - Store both first-hand and second-hand nondeterministic state
  - Tracks thread-level execution and interleaving of state
- Code snippets generated and inserted as functions
  - Re-execute second-hand nondeterministic actions, given the first-hand nondeterministic state as input
  - Snippets only replay the minimum needed to recreate the second-hand nondeterministic state
  - Example: first-hand nondeterministic variable `x` contaminates two other variables `y` and `z` through functions *f()* and *g()*, respectively
    - Code snippet will contain *f(x)* and *g(x)* to recreate the second-hand nondeterministic variables `y` and `z`, given `x` as input

Nondeterminism in Multi-tier Architecture



STATE IS INCONSISTENT!
APPLICATION SEMANTICS HAVE BEEN VIOLATED!

Multi-tier Example



Conclusion

- Midas: Program-analytic approach to handling nondeterminism
  - Deliberately non-transparent
  - Consistency in the face of nondeterminism
  - Synergistic combination of compile-time analysis with runtime compensation
- Efficient: Addresses only the nondeterminism that matters
- Different analyses to gain insight into application behavior
  - Dependency analysis, concurrency analysis, nondeterminism analysis
- Different techniques for runtime compensation
  - checkpoint-to-compensate, reexecute-to-compensate
- Leaves application semantics (and programmer intent) unaffected

Insights from Results

- Lower amounts of nondeterminism cause much less overhead
- Adding more clients increases the overhead due to increase in the number of callbacks
- Application characteristics will determine overhead
- Re-execution vs. transfer of contaminated state
  - Depends on processing costs of second-hand nondeterminism

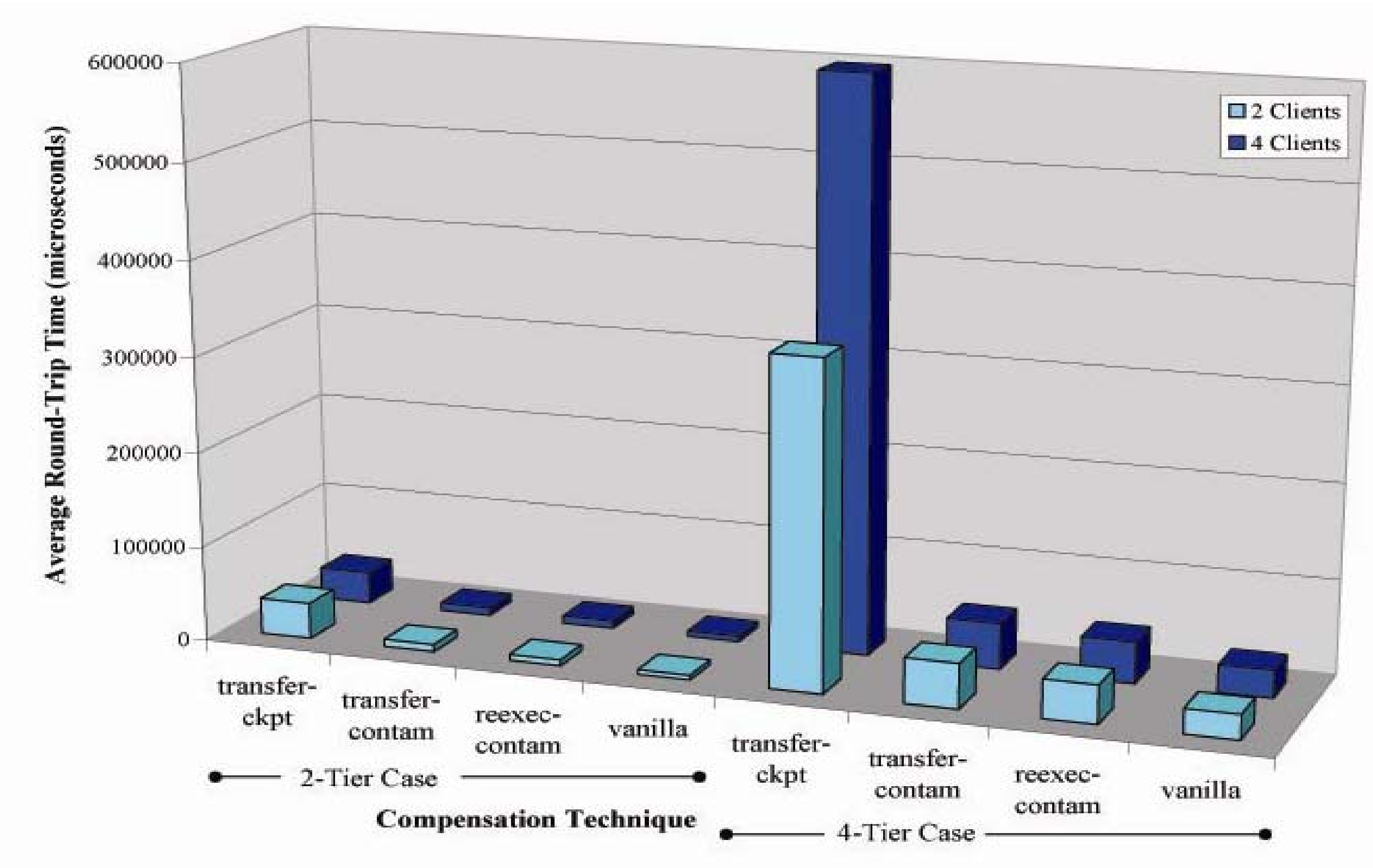
Preliminary Evaluation

- Multi-tier, multi-client nondeterministic application
  - Multi-threaded application with shared state across threads
  - Nondeterministic system calls
- Experimental setup
  - Pentium III, 850MHZ, 256MB RAM
  - Timesys Linux 2.4, Emulab, 100 Mbps Lan
- Varied number of clients: 2 and 4
- Varied number of tiers: 2 and 4
- Varied amount of forward and backward ND: 5% and 60%

Techniques Evaluated

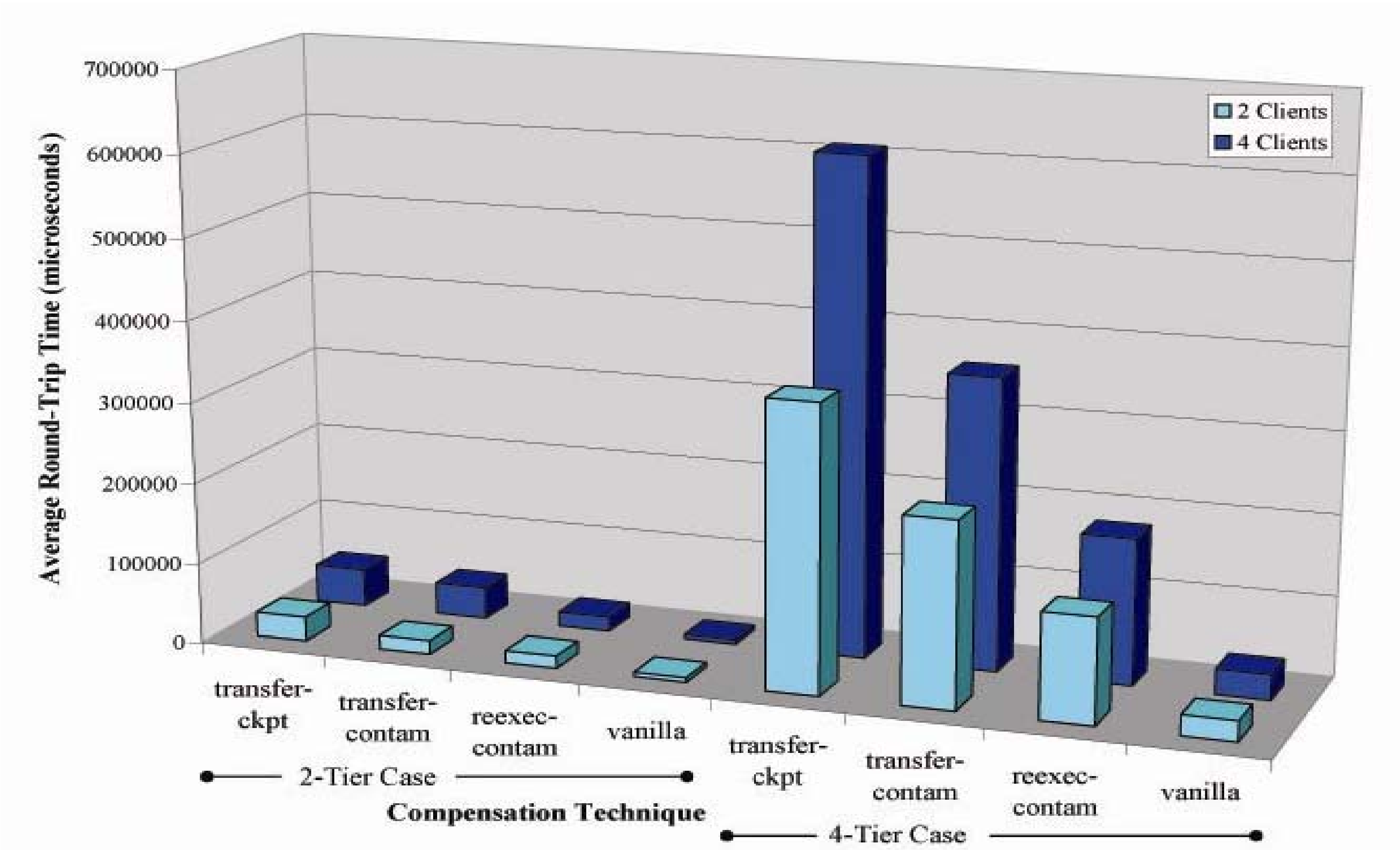
- Vanilla (serves as baseline)
  - Nondeterministic application running with no compensation
  - State will be divergent across replicas (but we don't care)
- Transfer-checkpoint (*transfer-ckpt*)
  - Transfers all of the persistent state in all callbacks
- Checkpoint-to-compensate (*transfer-contam*)
- Reexecute-to-compensate (*reexec-contam*)
- Metric of comparison: Round-trip latency on the client-side

Initial Results – 5% Fwd and 5% Bwd ND



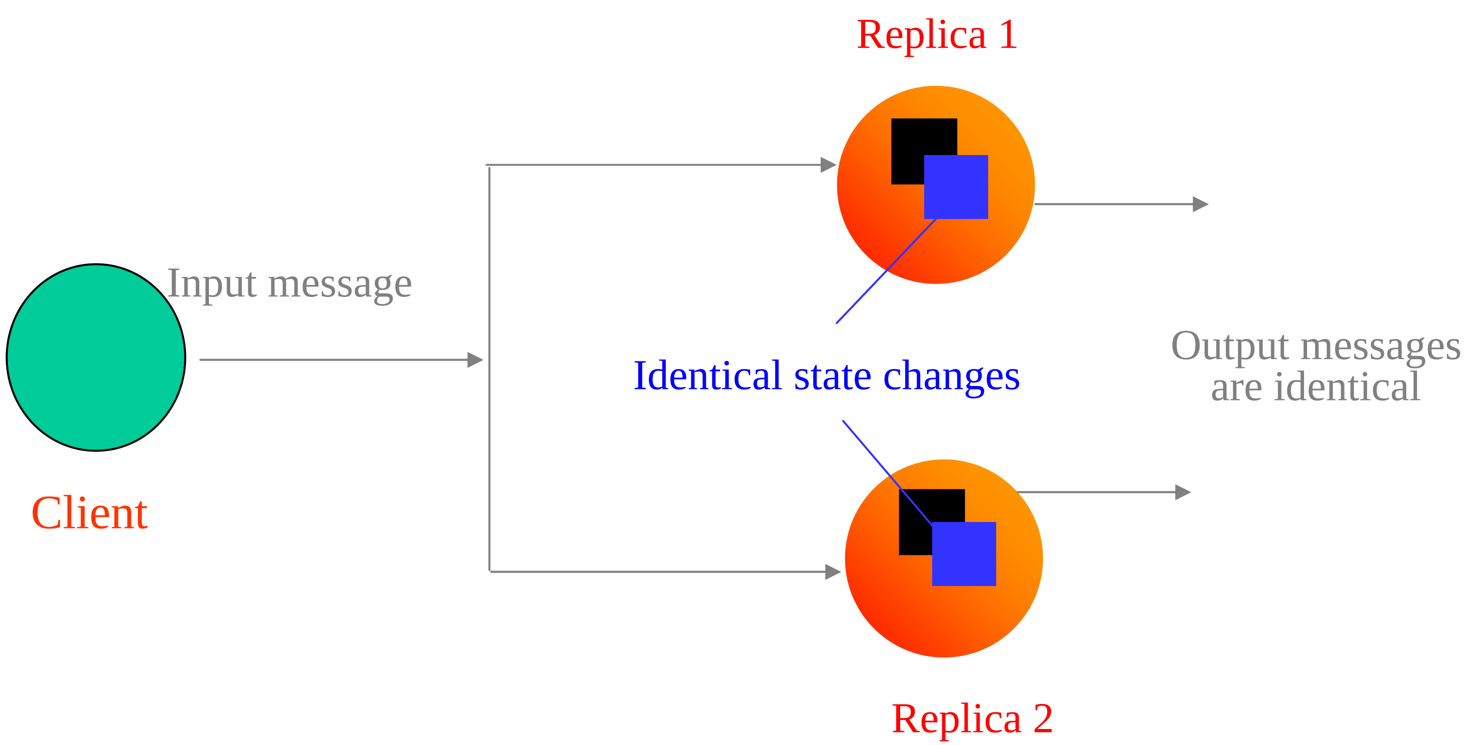
In 4-tier case, *transfer-contam* and *reexec-contam* scale well

Initial Results – 60% Fwd and 60% Bwd ND



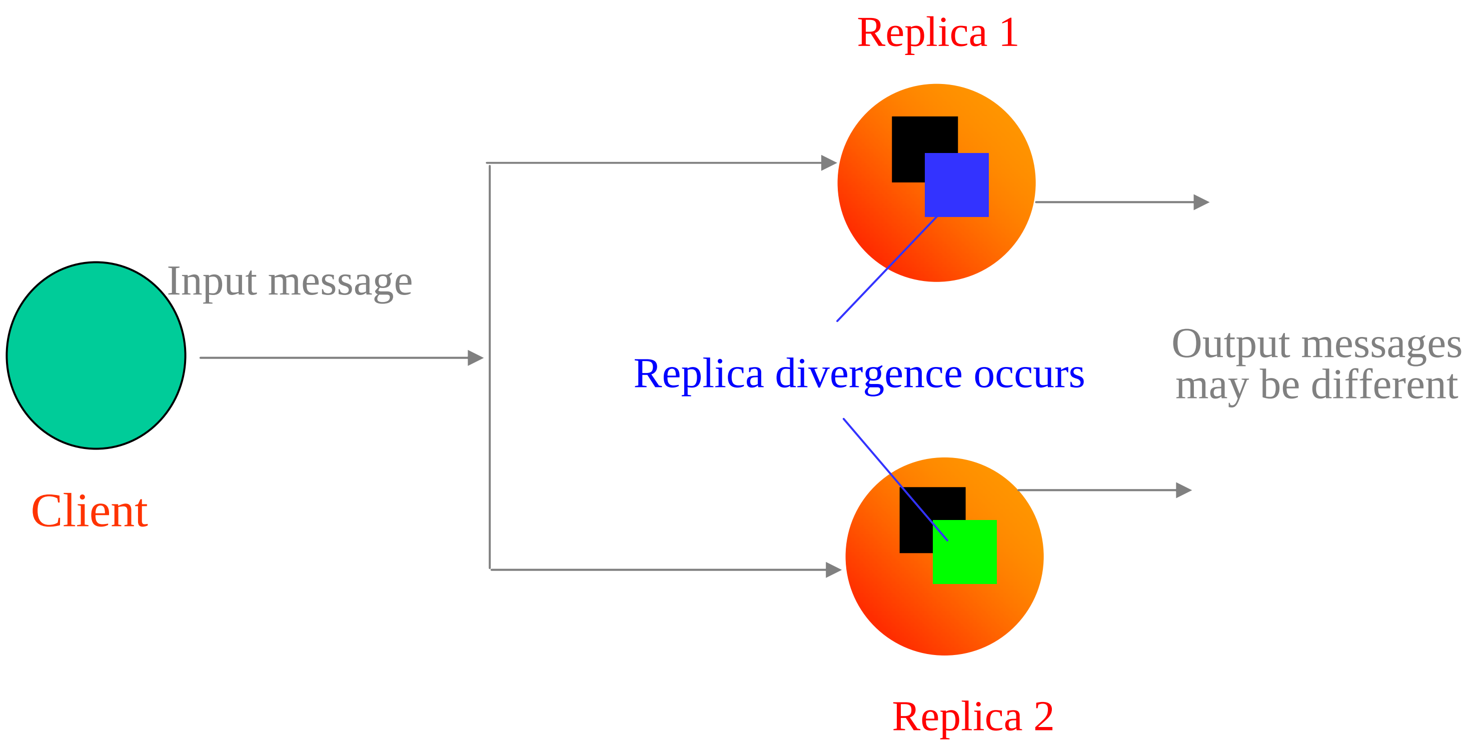
In 4-tier case with high actual nondeterminism, *transfer-contam* and *reexec-contam* see increased overhead

Deterministic Behavior



Nondeterministic Behavior

- Examples of nondeterminism
  - `gettimeofday(), random()`
  - Multithreaded execution



Current & Future Directions

- Vary application-level characteristics in evaluation
  - Request size, state size, processing time, inter-request latency
- Add dynamic analysis techniques
- Comparative analysis with a transparent technique
- Combine transparent technique with Midas
- Real-world benchmark
  - Welcome suggestions
  - Petstore?
  - Apache?

Transparent Handling of ND

Pros

- Does not need access to source code
- Can typically be applied to any application in a plug and play fashion

Cons

- Not every nondeterminism action results in state divergence
- Many transparent techniques don't know dependencies
    - Transparent techniques are unable to differentiate between actual and superficial nondeterminism

Types of Nondeterminism

- Two kinds of ND: Interaction and Control Flow
- Interaction
  - System Calls
    - `gettimeofday, read, write`
  - Input-output
    - Input from user, database, NIC card, etc.
- Control Flow
  - Multithreading
  - Asynchronous Events
    - Interrupts, Exceptions, Signals

Searching for Additional Sources of ND

- Functions are extracted from all source code
- App. defined functions removed from list
  - Some application-level functions might be added back in due to control flow nondeterminism
- Matches between the remaining list and the dictionary are removed
  - We know that these are nondeterministic
- Functions dependent on functions in dictionary are added to the dictionary and removed from list
- Remaining functions are potentially nondeterministic
  - Must go through manually with programmer

Searching for Control Flow ND

- Determine all shared state between threads
- Classification of shared state as ND
  - All reads and writes are considered 1^{st}-hand ND
- Do not impose interlocking
- Assume all interleaving is possible
  - This may be naïve, but optimizations are future work
- Compensation is done after the fact
  - Techniques described later in talk

Second-hand Nondeterminism

- Control-Flow and data-flow analysis used for dependency analysis
- Need to determine dependencies on 1st-hand nondeterminism
- These dependencies are determine based on execution path
- 2nd-hand nondeterminism is determined by tracing possible paths of execution
- Both 1st-hand and 2nd-hand ND can cause state to diverge across replicas

Some Related Work

- Fault-Tolerant CORBA standard
- OS and virtual machine solutions [Bressoud 96/98]
- Special schedulers [Basile 03, Jimenez-Peris 00, Poledna 00, Narasimhan 98]
- Specific replication styles [Barrett 90, Budhiraja 93]
- Execution histories [Frolund 00]

Checkpoint-to-compensate

- Only data structure annotations are used
- Track all first and second-hand ND
- Assume a multi-tier example
  - **client C ↔ server S1 ↔ server S2**
  - **S1 and S2 are replicated server groups**
- Assume nondeterminism exists in S2
- When S1 makes a request to S2 tier, S2 replicas will process request and they will all reply
- Piggyback their ND data structures on reply

Checkpoint-to-compensate cont.

- S1 replicas will all choose same response due to totally ordered delivery of messages
  - Remaining messages are dropped
- S1 replicas pull the ND checkpoint piggybacked information and make an asynchronous callback to S2 replicas with this chosen checkpoint
- S2 replicas update their state with the ND checkpoint sent
- All replicas should be consistent at this point

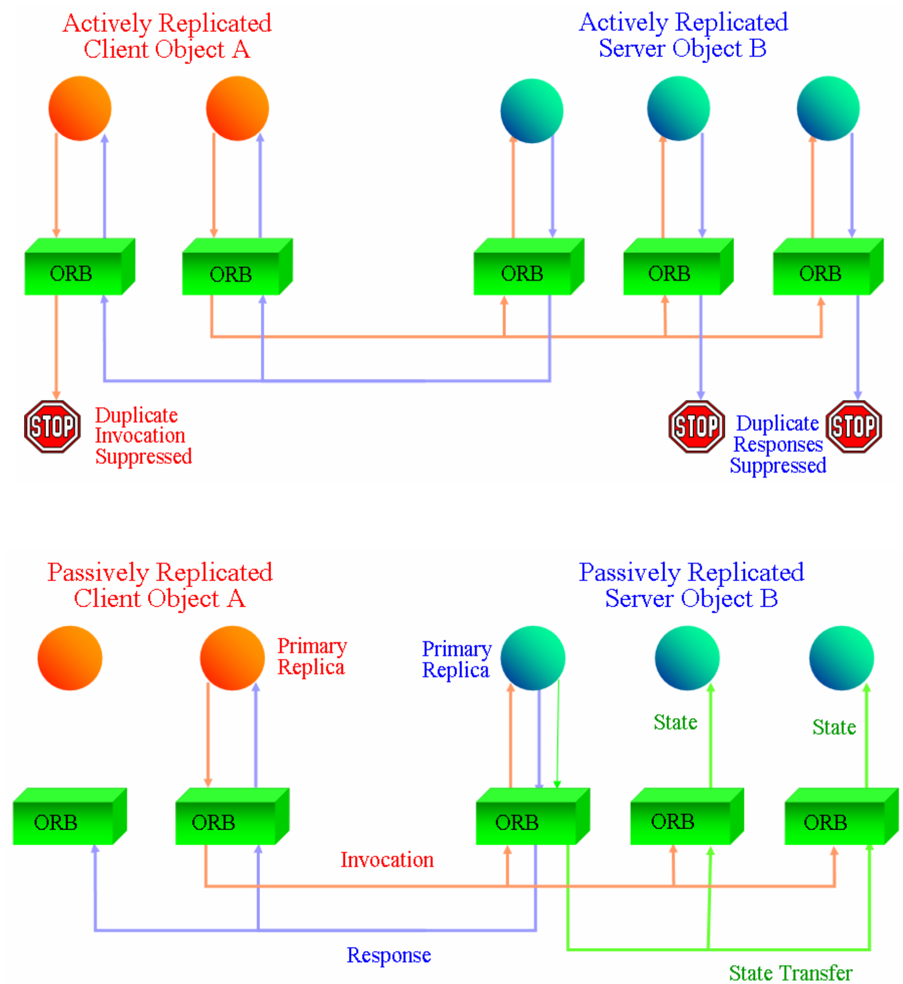
Reexecute-to-compensate

- Both types of annotations to source-code are used
- Only first-hand nondeterminism is tracked
- S2 replicas only piggyback first-hand ND on reply to S1
- S1 send out asynchronous message to S2 replicas with first-hand ND choice
- S2 replicas copy over first-hand information to their state, but then execute code snippets to compensate for second-hand ND

Forward and Backward ND

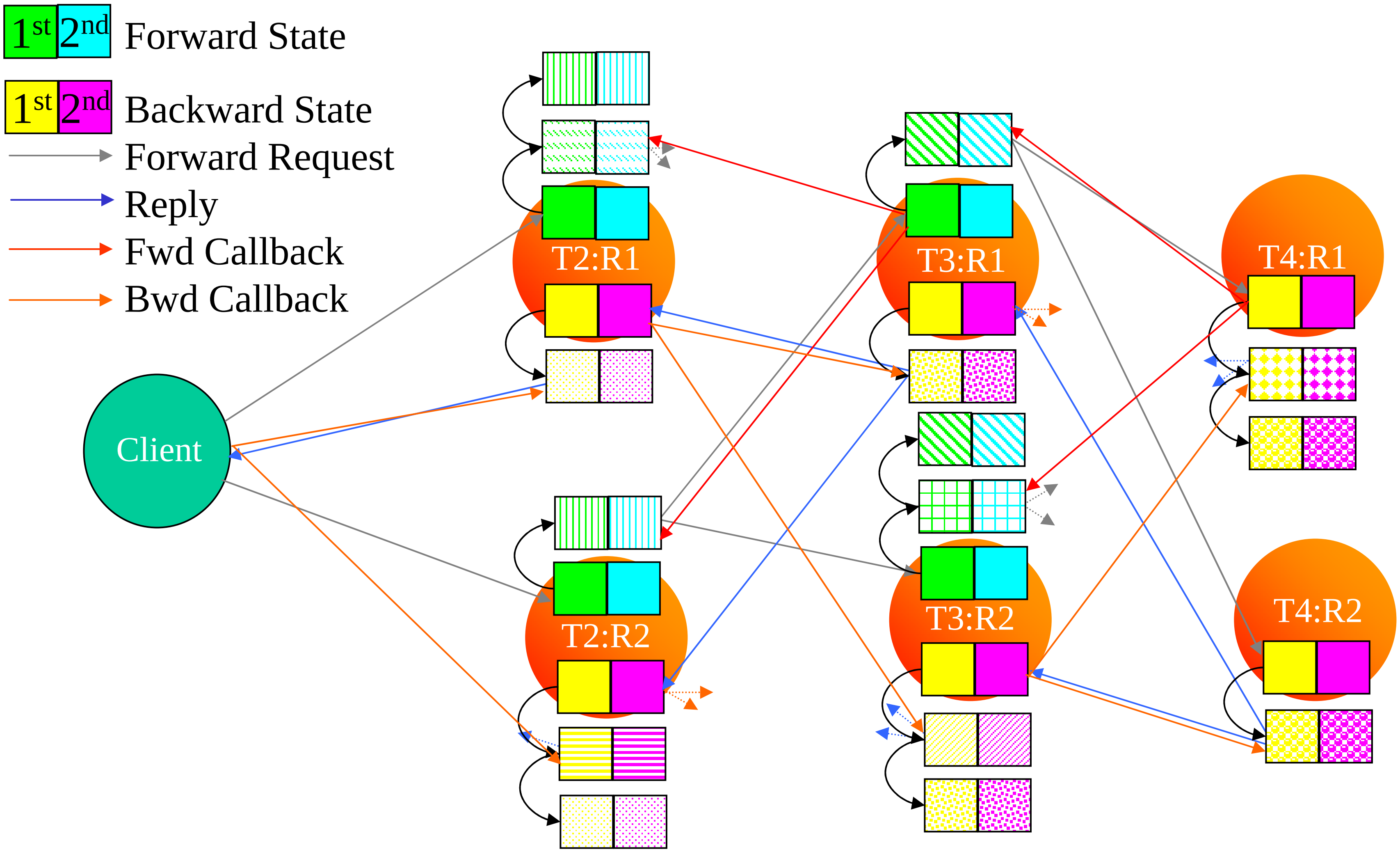
- The compensation callbacks described above can be both forward and backward
- Forward and backward ND need to be handled with different callbacks, both forward and backward

Different Fault-Tolerance Strategies



- Active / State-machine
  - Every copy receives and processes every message
  - Every copy is **active**
- Passive (primary-backup)
  - Only one (primary) copy processes all of the messages
  - Other (backup) copies receive state updates from the primary
  - Backups are **passive**

Multi-tier Example



Three-Tier Example