

# Transactional Orc

Katie Coons

May 2, 2008

# Why Transactions?

- Performance?
  - High overhead, especially STM
  - Performance is unpredictable
- Ease of programming?
  - Efficient transactions are small
  - Suspiciously similar to fine-grained locking

In Orc, there is no construct for  
atomicity or isolation

# Outline

- Motivation
- Transactional Semantics
- Orc engine/Site Interface
- Transactions in Orc
- Transactions at the Site
- Conclusions

# Transactions

- **Atomicity**
  - All or nothing
- **Consistency**
  - Does not violate any integrity constraints
- **Isolation**
  - Intermediate state must not be observable
- **Durability**
  - Transaction will persist, not covered here

# Transactional Memory

- Memory accesses modified
  - Two versions of data modified by transaction
  - Two versions of code
  - Detect conflicts
- Transaction manager
  - Validates read/write set
  - Arbitrate conflicts
  - Commits
  - Rolls back
- Do not penalize non-transactional code
- Interaction between tx and non-tx code

# Concurrency

- TM: Atomic regions are sequential

```
T1  atomic{  
    }  
    }
```

```
T2  atomic{  
    }  
    atomic{  
    }  
    }
```

```
T3  atomic{  
    }  
    }
```

- Orc: Atomic regions may be concurrent

```
atomic( ( T1 f | T2 g | T3 h ) )
```

# Atomic Expressions in Orc

Possible semantics for atomic expr  $f$ :

$\text{atomic}(f)$  publishes  $x$

- If  $f$  is silent,  $\text{atomic}(f)$  is also silent
- If  $\text{atomic}(f)$  publishes  $x$ ,  $x$  is  $f$ 's first publication
- Only site calls necessary for publishing  $x$  called
- Aborts if first publication by  $f$  cannot be committed
- If  $f$  aborts it is retried

$\text{atomic}(f)$  publishes  $(x, b)$

- $b = \text{true}$  if minimality ensured (committed)
- $b = \text{false}$  if minimality not guaranteed (aborted)

# Alternative Semantics

`atomic(f)` publishes whatever `f` would, if atomic

- If `f` is silent, `atomic(f)` is also silent
- If `atomic(f)` publishes, all values published were computed in isolation from anything outside of `f`
- If any computation in `f` conflicts with an access outside of `f`, `atomic(f)` aborts and retries

Similar to transactional memory, but a transaction is not limited to a single thread

`atomic(f | g) | h` While `f` and `g` can conflict, they must be independent of `h`

# Outline

- Motivation
- Transactional Semantics
- **Orc engine/Site Interface**
- Transactions in Orc
- Transactions at the Site
- Conclusions

# Orc Engine/Site Interface

- Orc engine
  - Keeps track of tokens involved in transaction
  - Verifies transactions after all tokens finished
  - Asks sites whether transaction is valid
  - Prevents tokens from leaving until commit
  - Retries transaction on abort
- Sites
  - Inform orc engine if violation occurs
  - Prevent intermediate state from leaking
  - Keep track of transactional operations
  - Rollback aborted transactions

# Orc Engine/Site Interface

- All sites implement the following:
  - Non-transactional site call  
`callSite(args, token)`
  - Transactional site call for transaction with identifier tid  
`callSiteTx(args, token, tid)`
  - Validate transaction with identifier tid  
`validateTransaction(tid)`
  - Commit transaction with identifier tid  
`commitTransaction(tid)`
  - Rollback transaction with identifier tid  
`rollbackTransaction(tid)`
  - Return true if transactions are supported  
`handlesTransactions(tid)`

# Outline

- Motivation
- Transactional Semantics
- Orc engine/Site Interface
- **Transactions in Orc**
- Transactions at the Site
- Conclusions

# Let Orc Understand “atomic”

```
atomic_expr returns [Expression e = null]
{
  Expression e2;
}
: "atomic" LPAREN e2=semi_expr RPAREN
{
  e = new AtomicExpression(e2);
};
```

# Let Orc Understand “atomic”

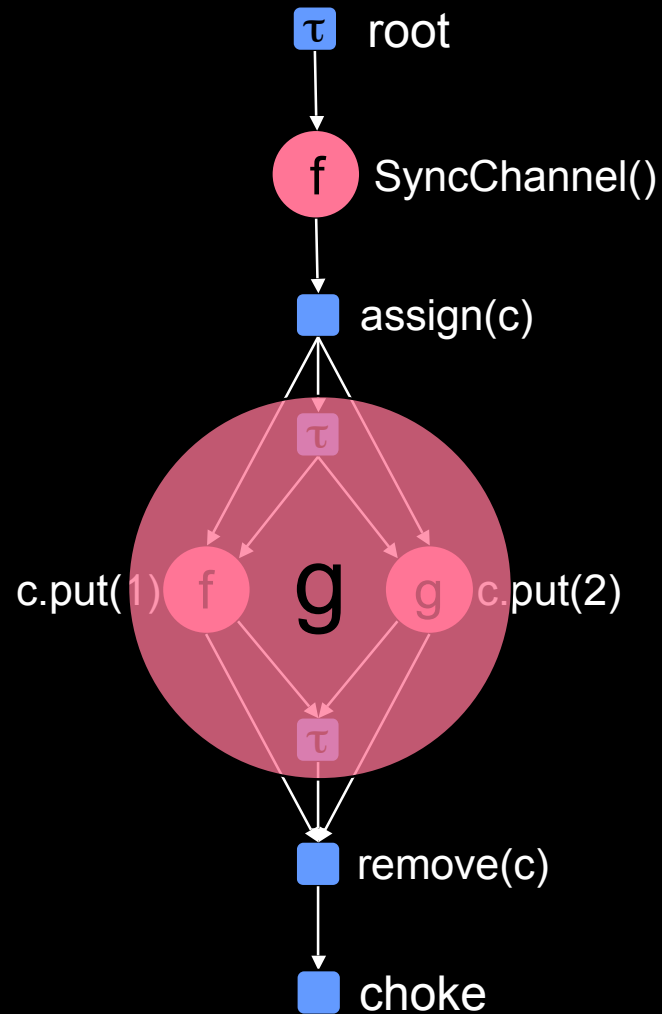
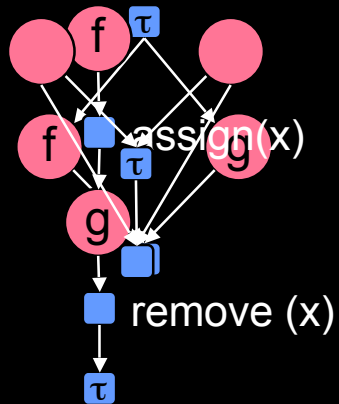
```
basic_expr returns [Expression e = null]
  : e=invoke_expr
  | e=lambda_expr
  | e=tuple_expr
  | e=list_expr
  | e=silent_expr
  | e=literal
  | e=atomic_expr;
```

# AST Modifications

```

SyncChannel() >c> (
  c.put(1) | c.put(2)
)
    
```

AST Modification

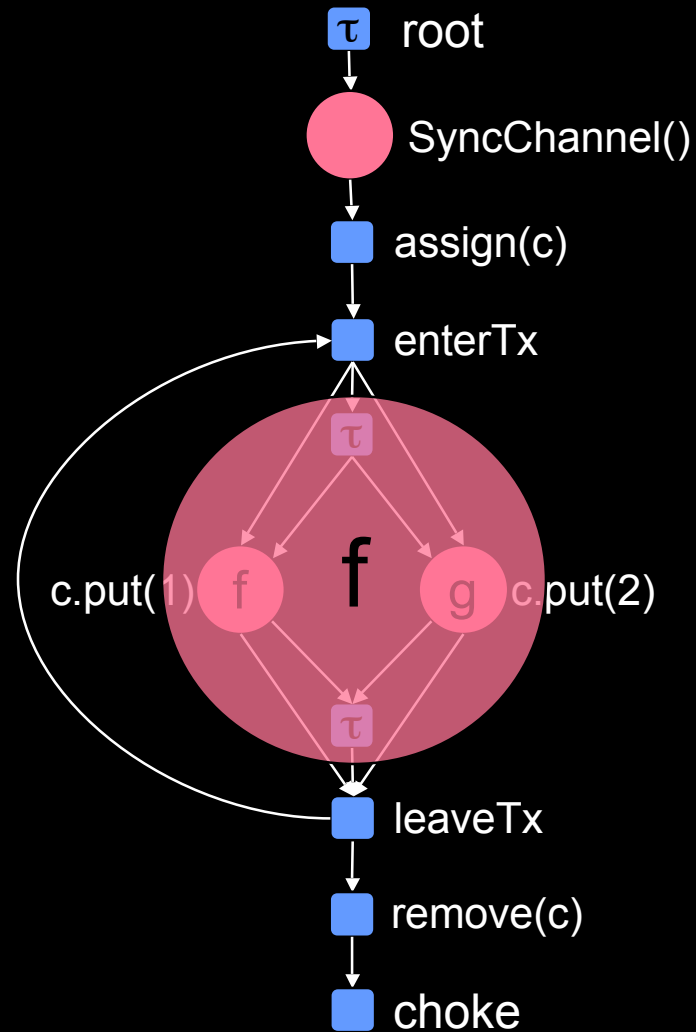
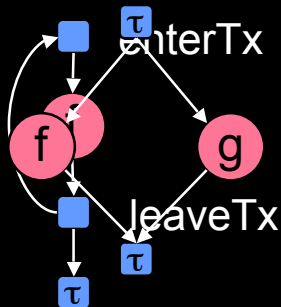


# AST Modifications

```

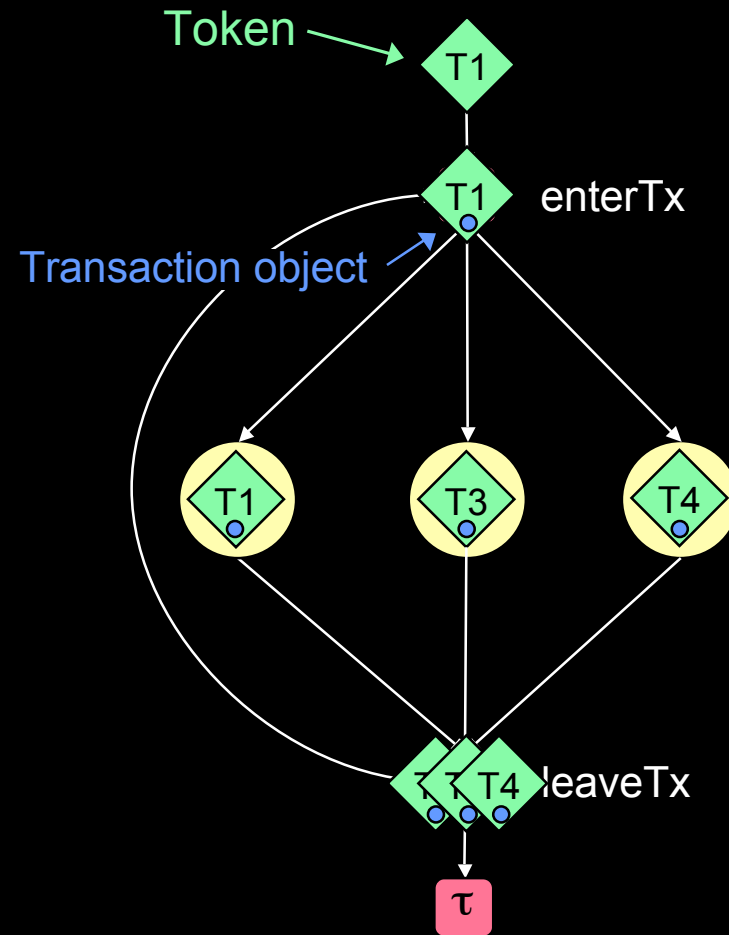
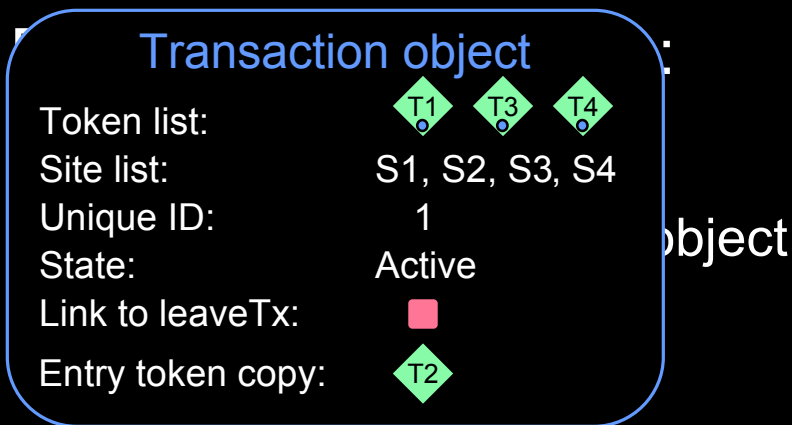
SyncChannel() >c> (
  atomic(c.put(1) | c.put(2))
)
    
```

**atomic(f)**



# Graph Traversal




atomic( f | g | h )

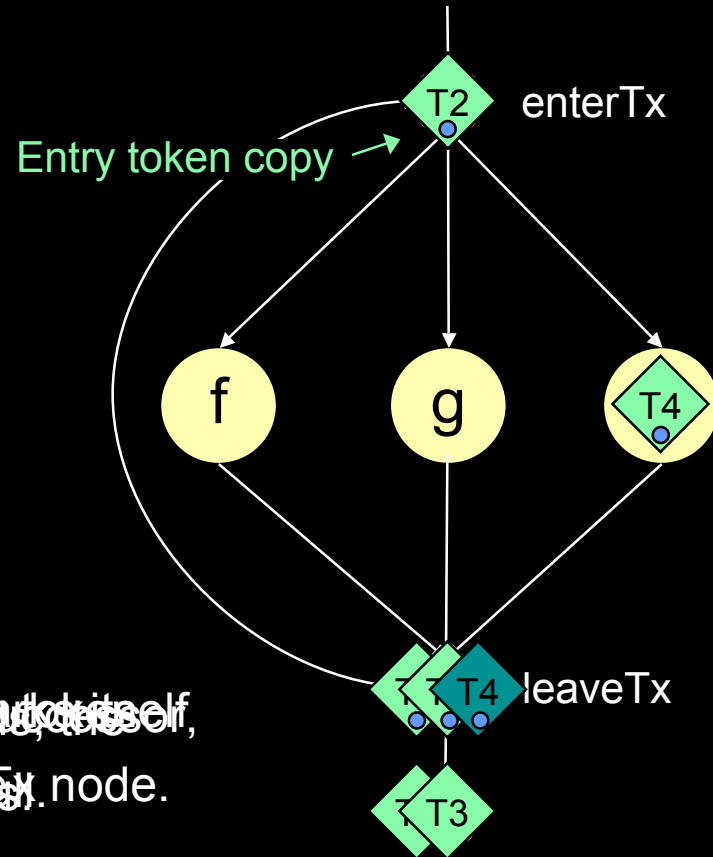


# Graph Traversal

atomic( f | g | h )

**Transaction object**

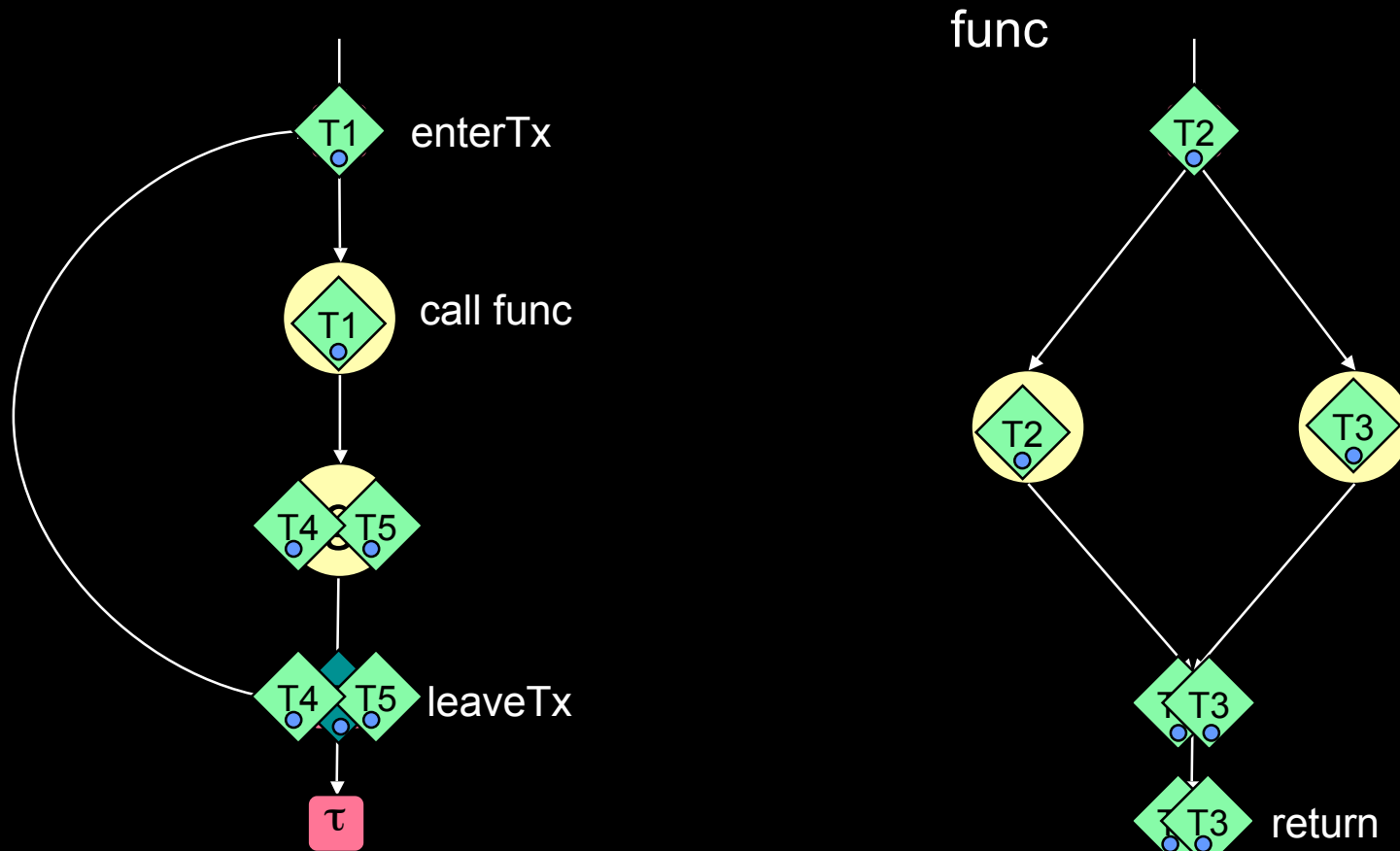
- Token list: 
- Site list: S1, S2, S3, S4
- Unique ID: 1
- State: Active
- Link to leaveTx: 
- Entry token copy: 



Control flow graph for atomic( f | g | h ) showing transaction objects T2, T3, T4 and nodes f, g, h. Transaction T2 is active and has an entry token copy. Transaction T4 is active and has a link to leaveTx. Transaction T3 is active and has a link to leaveTx. Transaction T4 is active and has a link to leaveTx. Transaction T3 is active and has a link to leaveTx. Transaction T4 is active and has a link to leaveTx. Transaction T3 is active and has a link to leaveTx.

Doomed transactions killed.

# Calls and Returns



# Outline

- Motivation
- Transactional Semantics
- Orc engine/Site Interface
- Transactions in Orc
- **Transactions at the Site**
- Conclusions

# Channels

- Three channel types
  - Non-blocking
  - Receiver blocks
  - Both sender and receiver block (SyncChannel)
- Three transaction manager policies
  - Single owner
  - Concurrent put and get/peek
  - Concurrent put and get/multiple peeks

- Added a peek method

Each channel has its own type and policy:

Channel(0,0) | Channel(2,2)

# Transaction Manager Policies

- Single Owner
  - Entire channel is a single data item
  - Reads/writes are not differentiated
  - Very pessimistic
- Concurrent put and get/peek
  - Differentiates data - two ends to channel
  - Reads/writes still not differentiated
  - Very pessimistic
- Concurrent put and get/multiple peeks
  - Differentiate reads and writes

# Conflict Resolution

- Transaction vs. transaction
  - First writer wins (gets abort peeks)
  - First access wins (peeks may abort gets)
  - New transaction aborts old (livelock)
  
- Non-transaction vs. transaction
  - Non-transactional access always wins
    - We only know its *next* node
    - Disrupting non-transactional code undesirable
  - Abort conflicting transaction
    - Must roll back transaction for access to complete
    - Alert engine that transaction is doomed

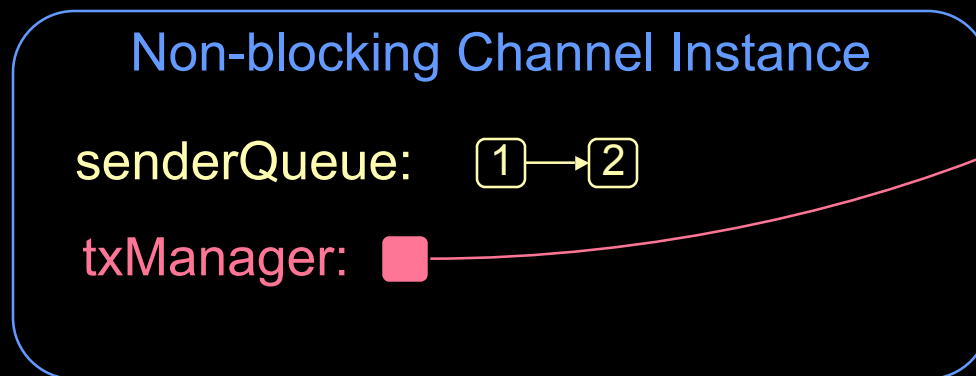
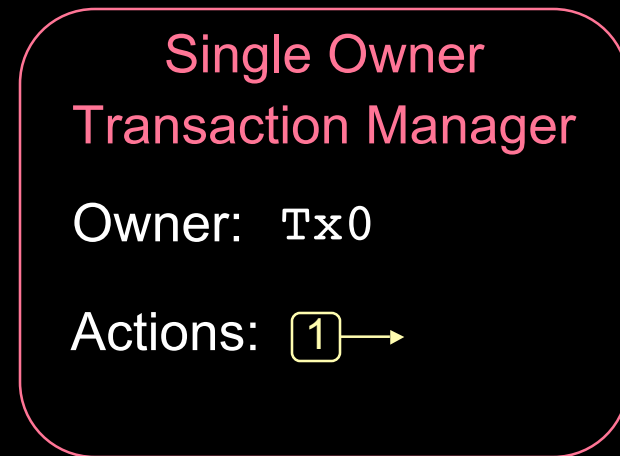
# Conflict Detection and Version Control

- Conflict Detection
  - Eager: Detect conflicts as soon as they occur
    - To handle non-transactional accesses
  - Lazy: Wait until validation
  
- Version Management
  - In-place: Modify visible state
    - Because it was easy to implement :)
  - Deferred: Modify state “on the side”

Made no changes to JVM - inefficient, but fast and easy to implement

# Channel Example

```
Channel(2,0) >c> (  
→ c.put(1)  
→ c.put(2)  
→ atomic(c.put(3))  
→ atomic(c.get() | c.get())  
→ c.get()  
)
```

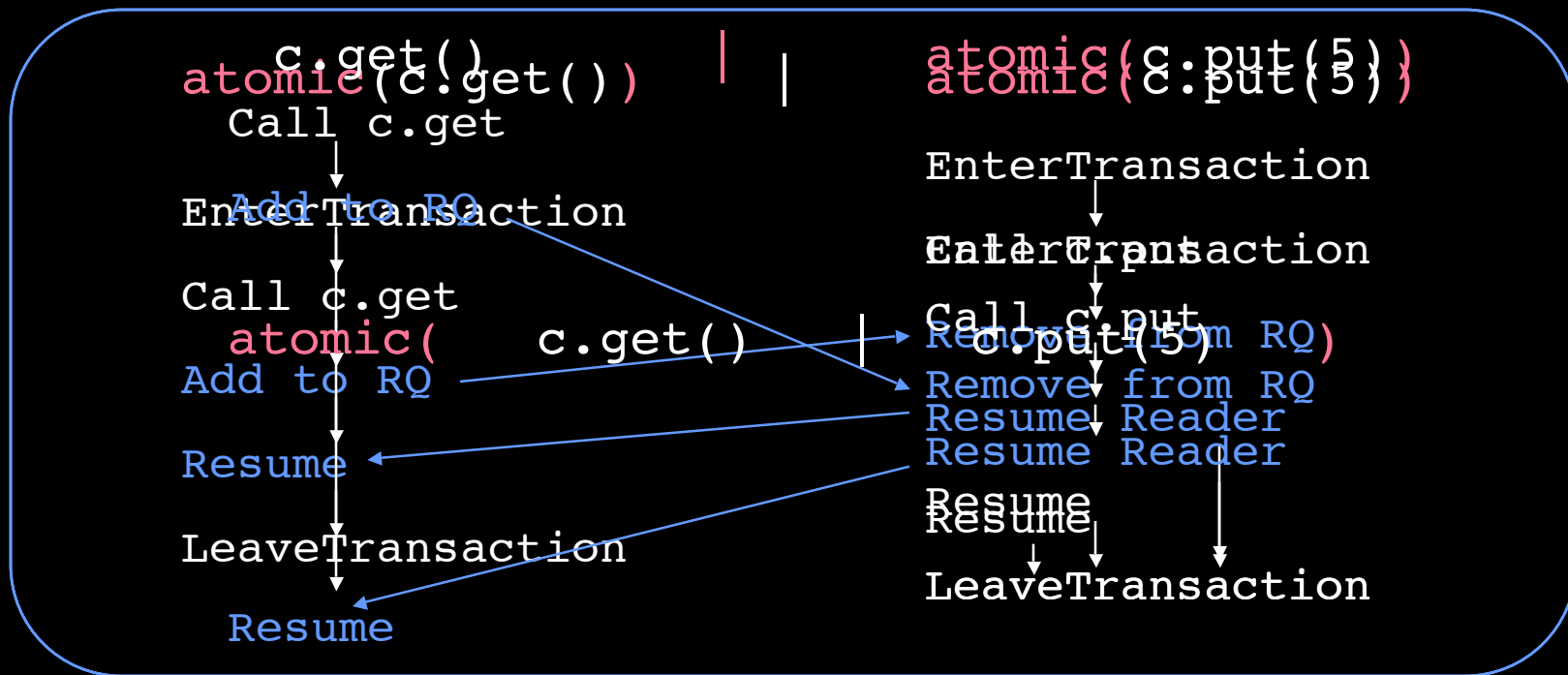


Transaction aborts and  
New transaction aborts!  
rolls back

Text with a red arrow pointing from the txManager box to the Transaction Manager box.

# Observations about Channels

- Blocking channel
  - Transactions cannot communicate with each other
  - Threads within transaction can only communicate on an empty channel



# Future Work

- Nesting
  - Concurrent nested transactions
  - What if sites disagree about how to nest?
- Timers
  - Can time elapse within a transaction?
  - What does a timer mean inside a transaction?
- I/O
- Commit protocols
- Abstract away from transactional memory?
- Asymmetric composition, backoff, synchronization

# Conclusions

- Orc needs a mechanism for atomicity
- Separation of concerns interesting, but risky?
  - More risk for deadlock? Livelock?
  - Nesting problems?
- One transaction, many threads
  - Nesting may look different
- Lots of exciting future work!

**Questions?**