Software Fault Tolerance

Hybrid Systems

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University

Overview

- Duality of transactions and conversations
- Multithreaded Transactions
- Open Multithreaded Transactions
  - Look-Ahead
- Coordinated Atomic Actions
- Design Diverse Extended Models
  - N-Version Programming Variants
  - Distributed Recovery Blocks
  - Consensus Recovery Blocks
  - Two-Pass Adjudicators
  - Self-Configuring Optimal Programming

The “Object” Model

- The transaction model (“object” model) and the conversation model (“process” model) are duals [SMR93]
- OM (Object and transaction model)
  - Two primary entities:
    - Object: long lived entity for holding system state
    - Transaction: short lived entity, providing a context in which state changes take place
  - Widely used in distributed systems
  - Example: database application, e.g. banking, office information and airline reservation systems

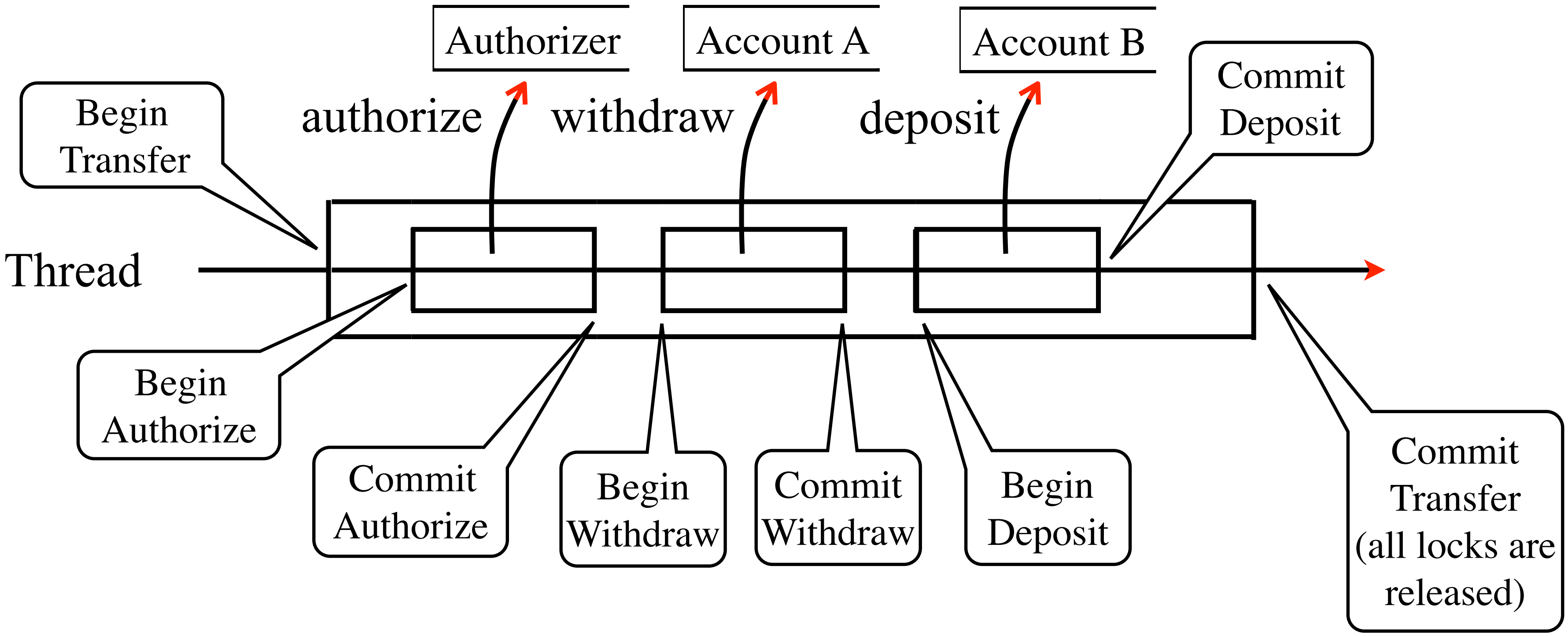
The “Process” Model

- PM (Process and conversation model)
  - Two primary entities:
    - Process: long lived entity for holding system states
    - Conversation: short lived entity, providing a context in which state changes take place
  - Widely used in real-time systems
  - Example:
    - Process control systems
    - Avionics systems
    - Telephone switching systems

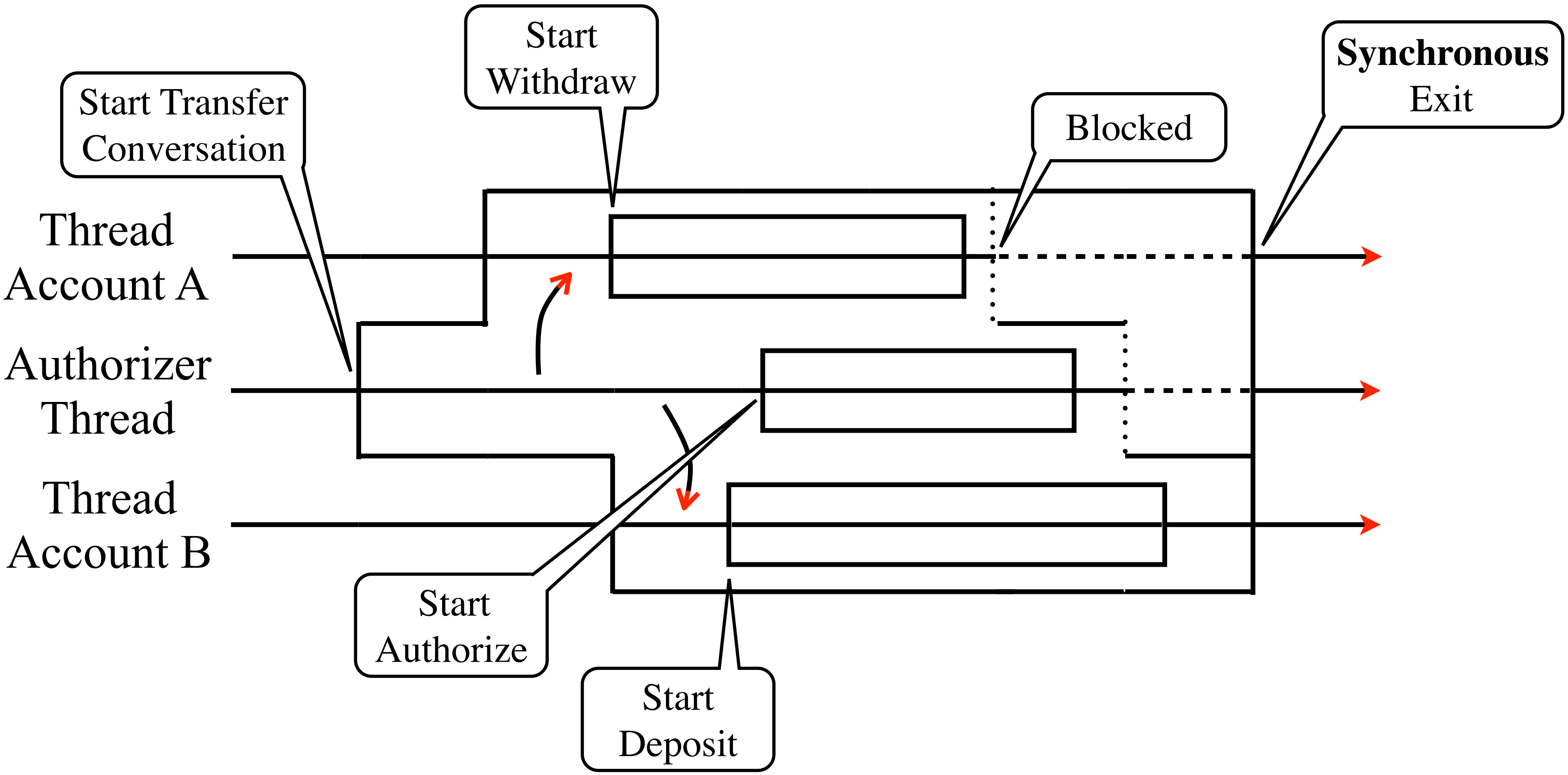
Duality Mapping

OM model	PM model
Objects	Processes
Transaction	Conversation
Object invocations	Message interactions
Concurrency control for serializability	Conversation rules: no outside communication
Stable objects	Stable processes
Growing phase (get locks)	Processes enter a conversation
Shrinking phase (release locks)	Processes leave a conversation

Example: Transfer Operation using OM



Example: Transfer Operation using PM



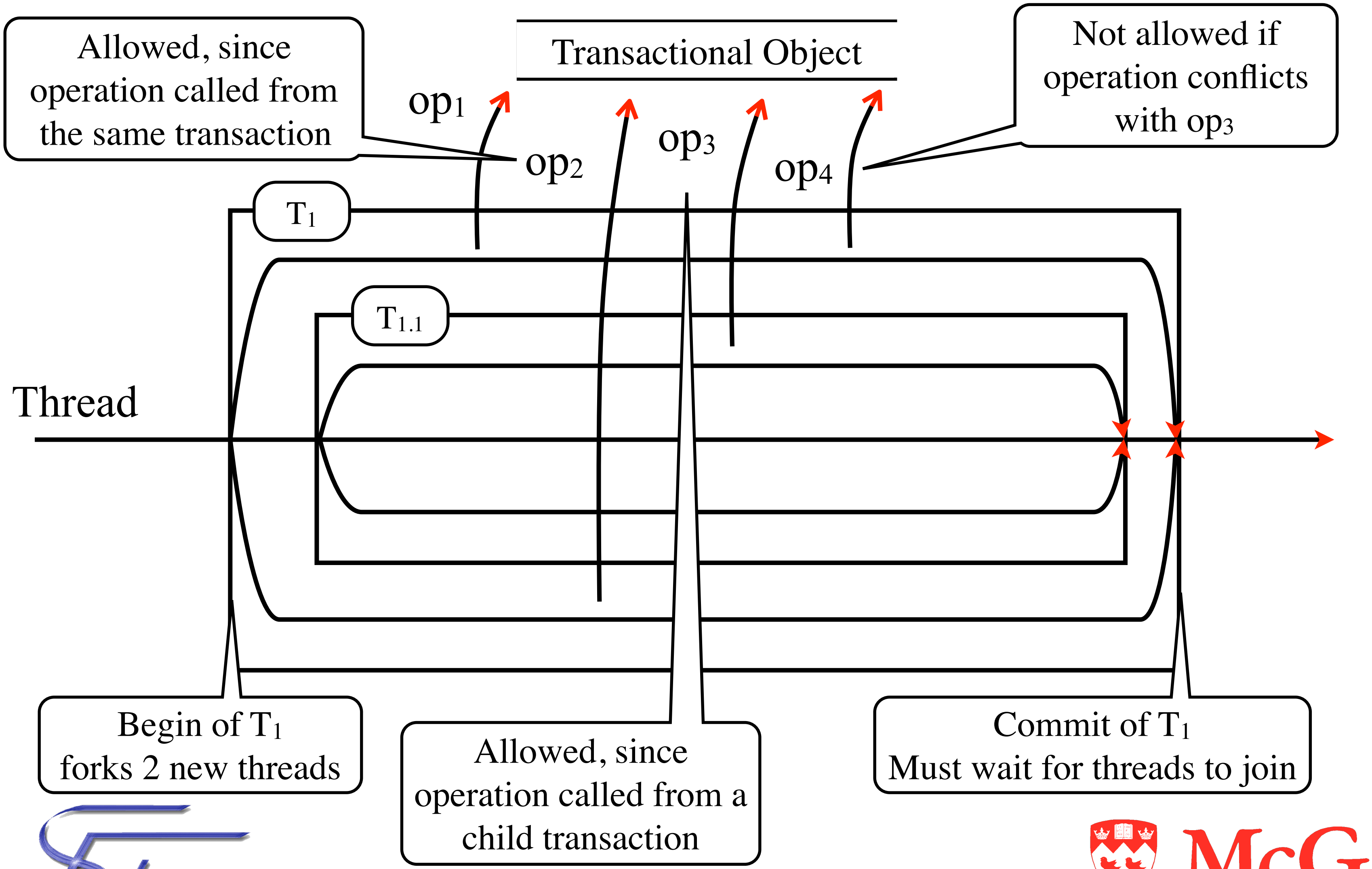
Competition vs. Cooperation

- Different application domains traditionally use one model
  - Process control: conversations
  - Data-intense applications: transactions
- There’s a need for integration of cooperation and competition
  - When the two domains want to interact
  - When concurrency is required
    - Distributed systems
    - Multi-processors
    - Threads to handle user interface and / or network

Multithreaded Transactions (1)

- Venari/ML [HKM+94] and Transactional Drago [JPPMA00]
- A thread in a transaction can spawn new threads
  - The forking takes place at the transaction border
  - The additional threads must terminate before the main thread commits / aborts the transaction
- Disadvantage
  - External threads can not join a running transaction

Multithreaded Transactions (2)



Allowed, since operation called from the same transaction

Transactional Object

Not allowed if operation conflicts with op_3

Begin of T_1 forks 2 new threads

Allowed, since operation called from a child transaction

Commit of T_1 Must wait for threads to join

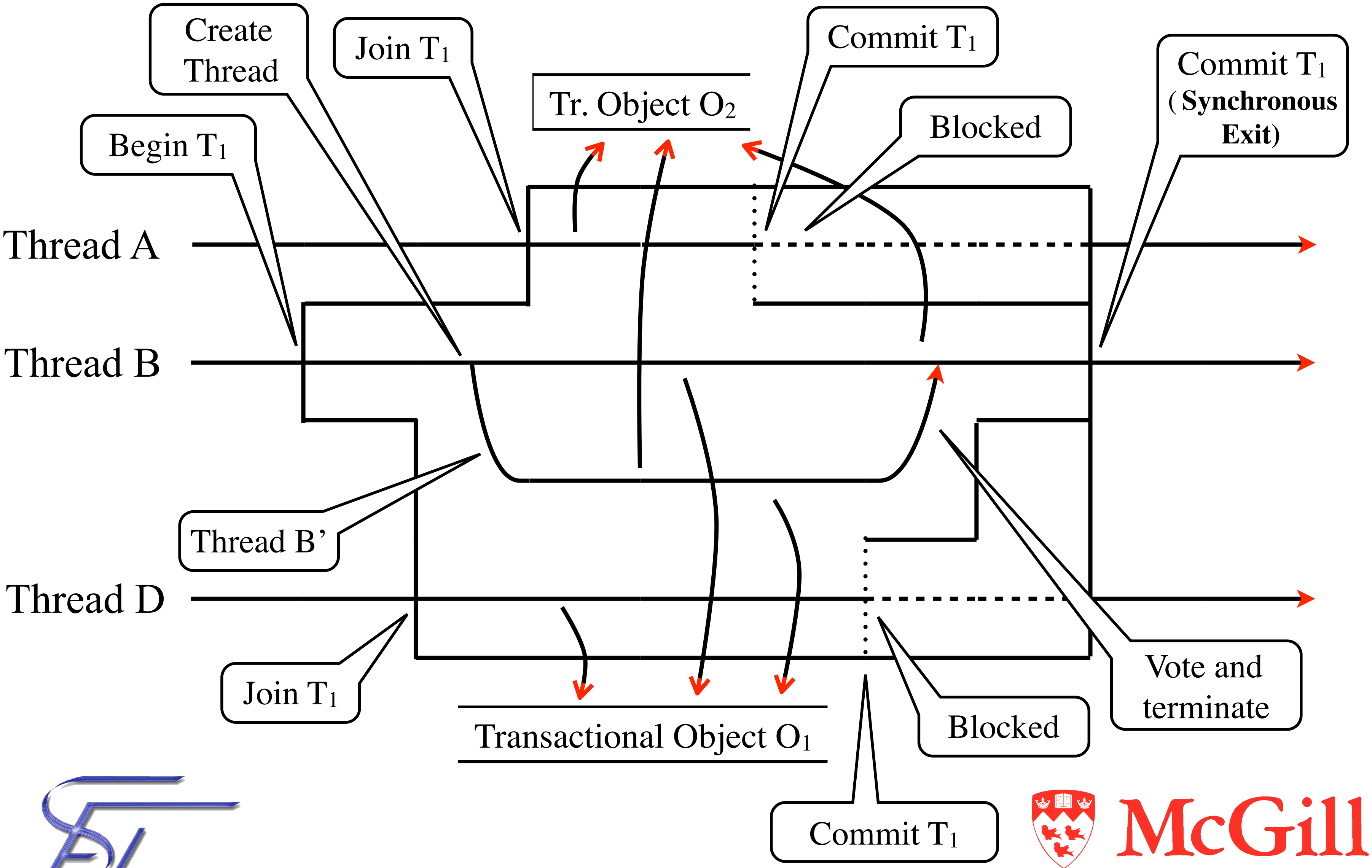
Open Multithreaded Transactions [KRS01]

- Thread creation / termination possible at any time, but:
  - Threads created outside a transaction are not allowed to terminate inside
  - Threads created inside must terminate inside
- Starting an Open Multithreaded Transaction
  - Any thread can start a transaction (*joined participant*)
  - Open Multithreaded Transactions can be nested
- Joining an Open Multithreaded Transaction
  - A thread can join an ongoing transaction iff it is not participating in any transaction other than ancestor transactions (also *joined participant*)

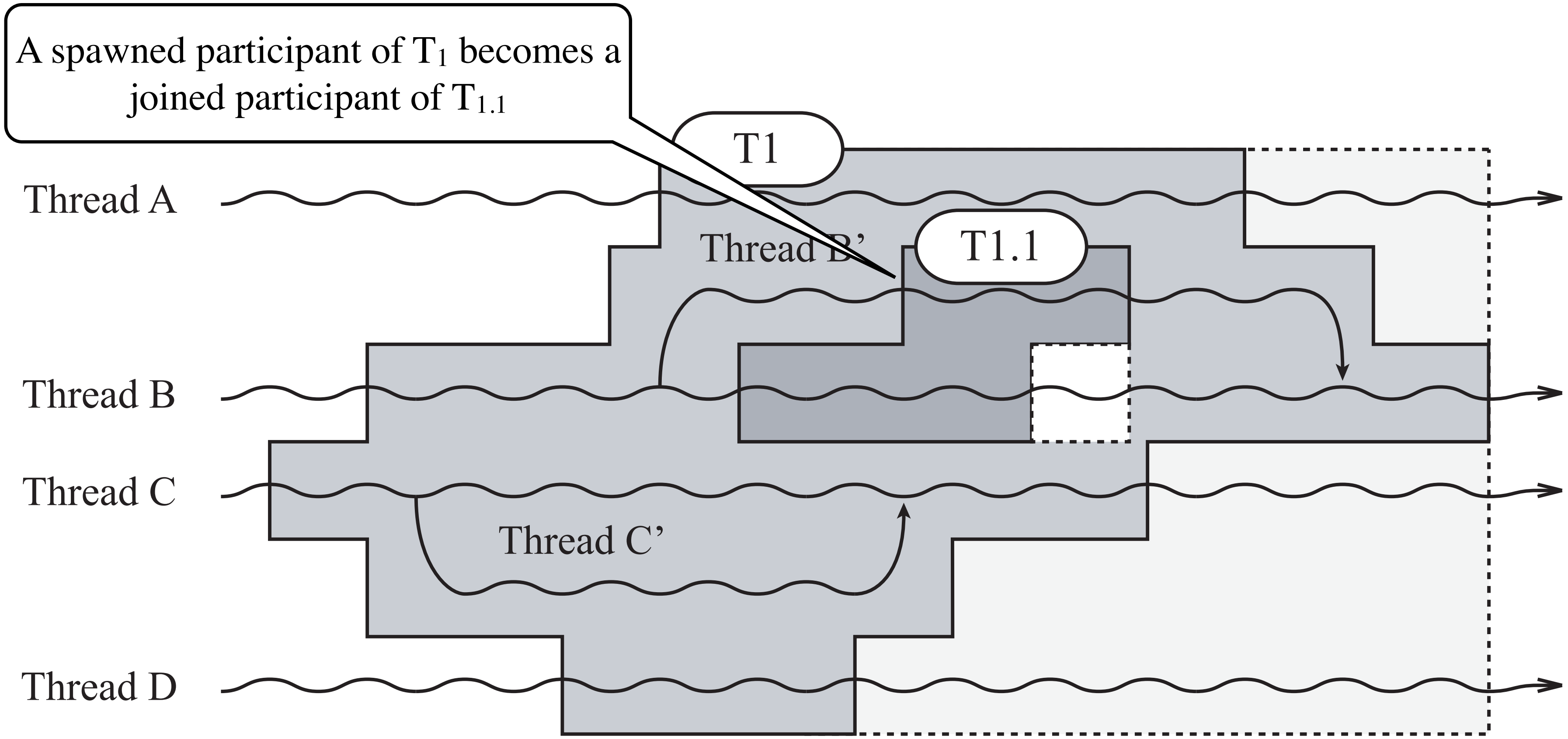
Open Multithreaded Transactions (2)

- Threads spawned inside a transaction become *spawned participants* of the transaction
- Ending an Open Multithreaded Transaction
  - All participants vote commit or abort
  - The transaction commits iff all participants vote commit
  - Spawned participants terminate after voting
  - Joined participants are blocked until the outcome of the transaction has been determined.

Open Multithreaded Transaction (3)



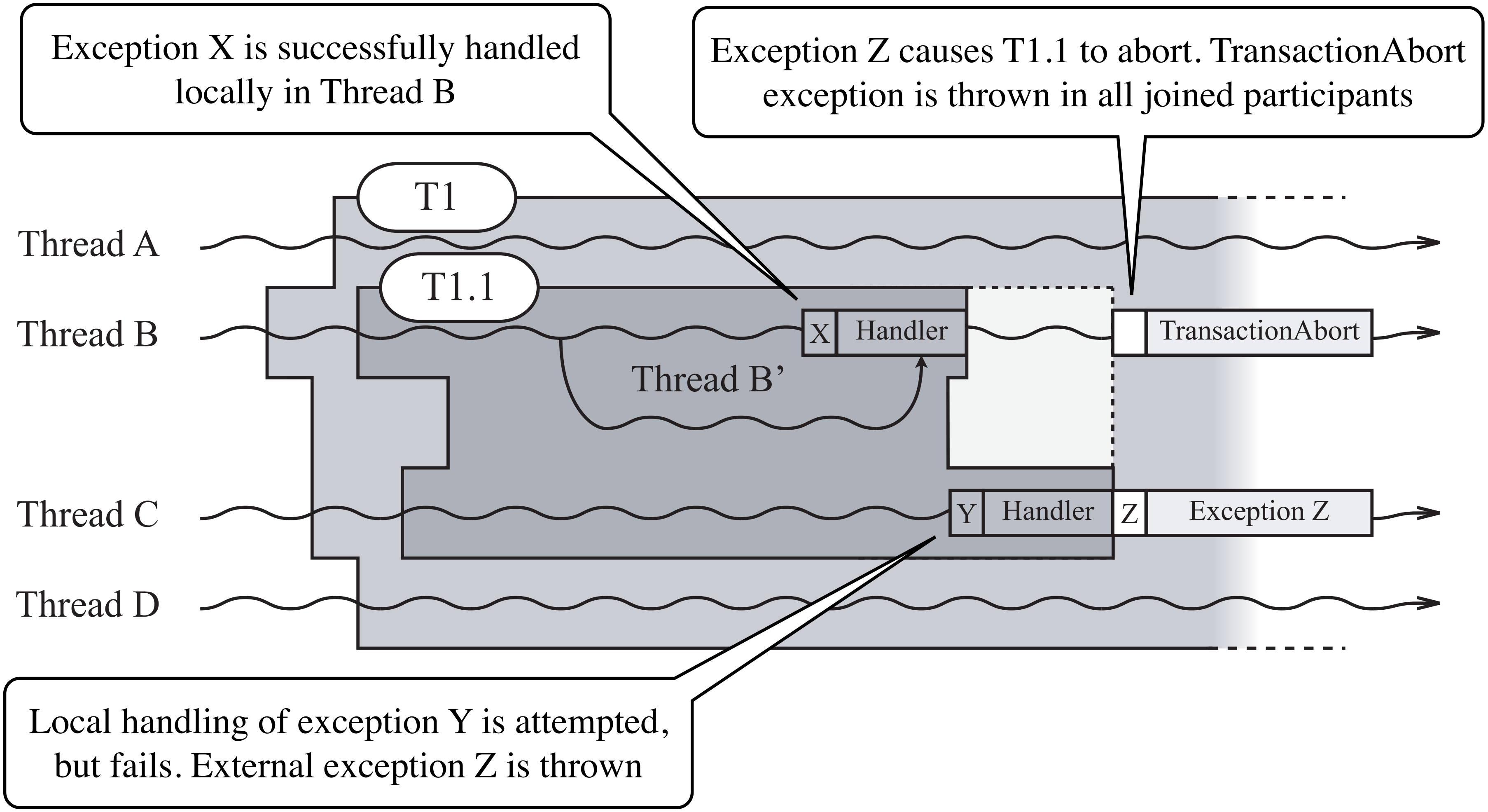
Open Multithreaded Transactions (4)


A spawned participant of T_1 becomes a joined participant of $T_{1.1}$

Exceptions in OMTTs (1)

- Internal exceptions are handled locally by a participant
- External exception result in aborting the transaction
  - Participants are notified with TransactionAbort exception
- Unhandled exceptions crossing the transaction boundary result in aborting the transaction

Exceptions in OMTTs (2)



Exception X is successfully handled locally in Thread B

Exception Z causes T1.1 to abort. TransactionAbort exception is thrown in all joined participants

Local handling of exception Y is attempted, but fails. External exception Z is thrown

Additional Features of OMTTs

- Closing an Open Multithreaded Transaction
  - Once closed, no new participants can join
  - Fix number of participants at creation-time
  - Any participant can close the transaction explicitly
- Naming an Open Multithreaded Transaction
  - Unnamed transactions -> asymmetric
  - Named transactions -> symmetric
- Deserters are treated as errors -> the transaction is aborted

Transactional Objects in OMTTs

- Two-level Concurrency Control
  - Competitive: Inter-transaction isolation
  - Cooperative: Mutual exclusion for updates performed by participants of the same transaction
- Self-checking Transactional Objects
  - Help the programmer guarantee consistency by invariants
  - Pre- and post-conditions for operations
  - Upon violation, an exception is raised
  - Triggers abort, if not handled

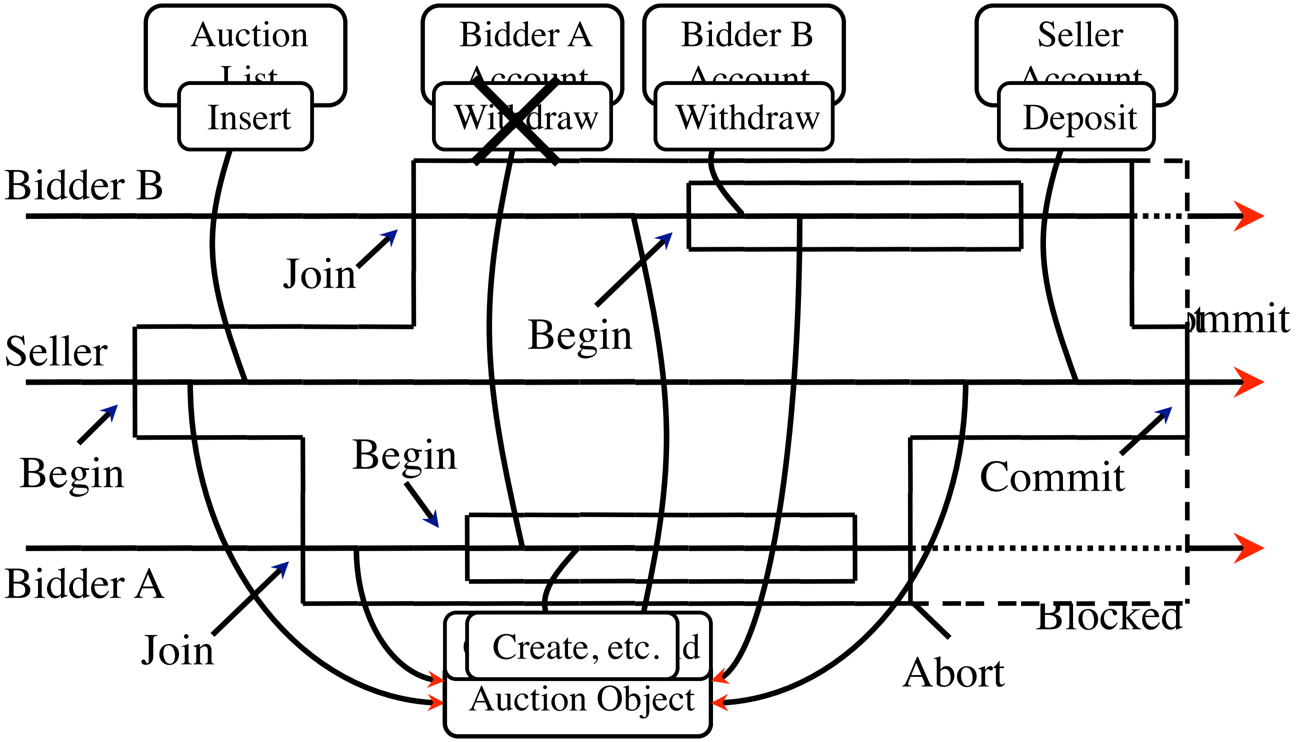
Auction System Example

- Dynamic system with cooperative and competitive concurrency
- Users register with the auction system
- Members can:
    - Deposit money on their account
    - Sell an item, starting a new auction
    - Consult the list of current auctions
    - Participate in an auction and bid for an item

Transactional Objects in the Auction System

- Data that should survive failures must be encapsulated inside a transactional object
- Transactional Objects in the Auction system:
  - Member Information
  - Member Directory
  - Accounts
  - Auctions
  - Auction List

Auction Design using OMTTs

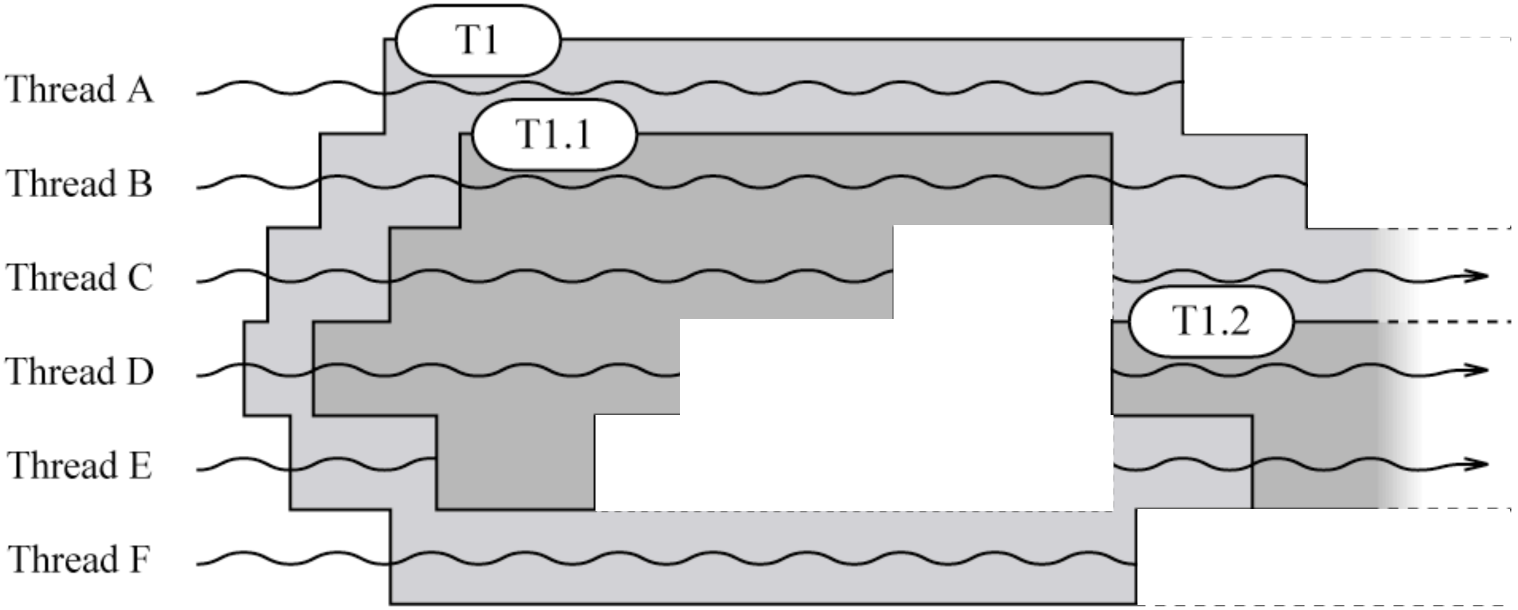


Advantages of using OMTTs for Auctions

- Consistency of the application state is guaranteed in spite of concurrent auctions
- All-or-nothing semantics: either the auction completes as a whole, or no money is transferred
- Fault tolerance
- Partial undo using nesting
- Users participating in several auctions cannot overdraw their account

Waisted Time due to Synchronous Exit

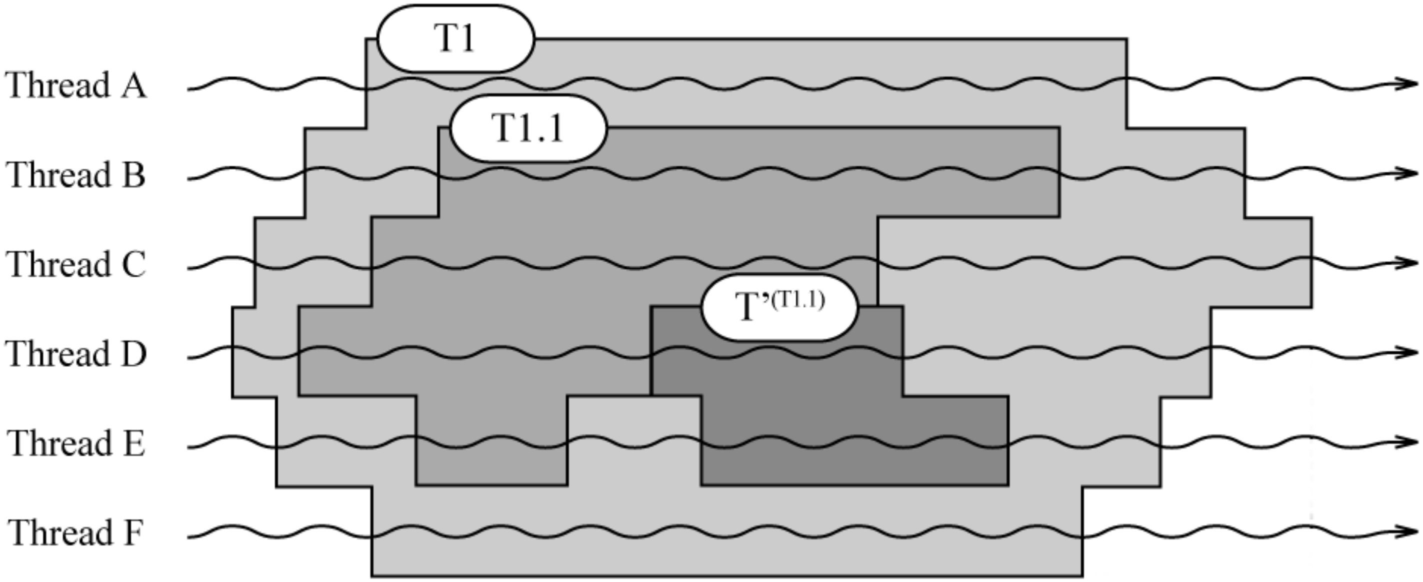
- To ensure isolation property, threads are blocked at commit time until outcome is known



Time spent Waiting

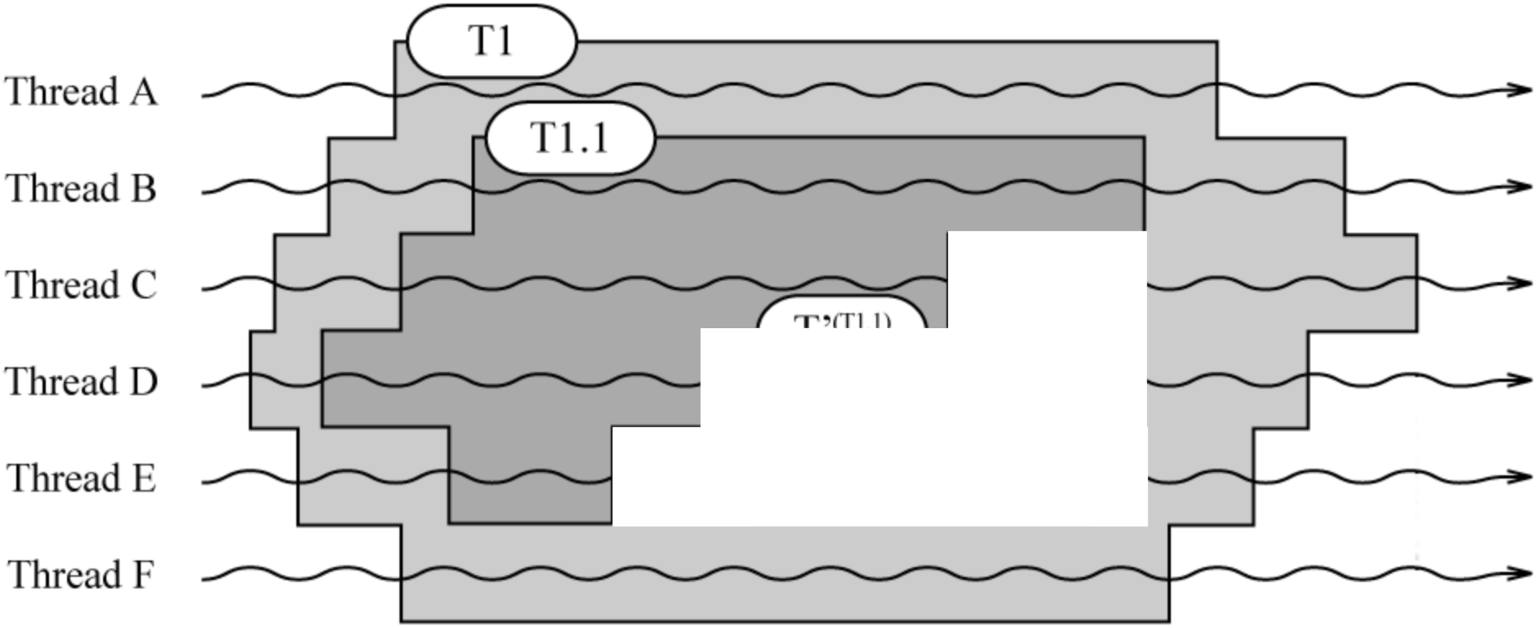
Look-Ahead

- Allow threads to look-ahead, i.e. continue optimistically as if the transaction committed
- Transparent!



Look-Ahead Complications

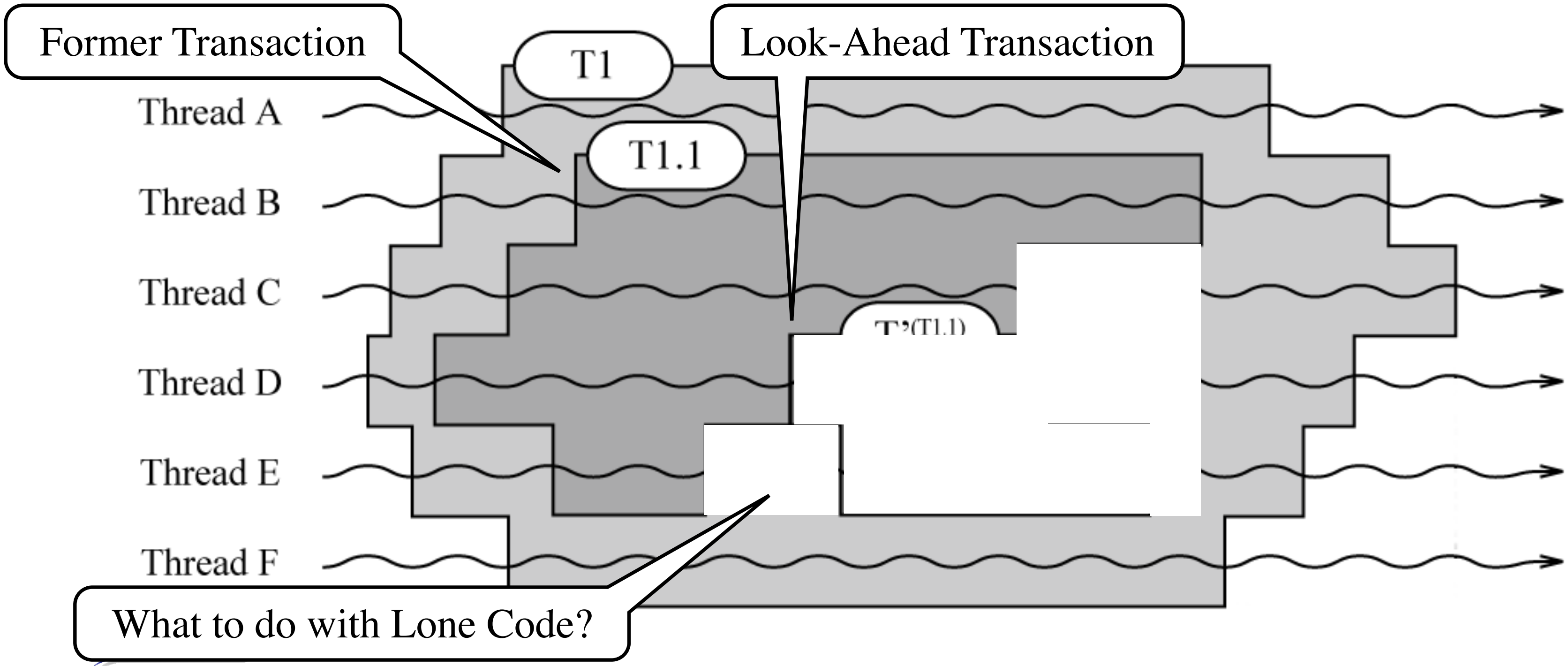
- Look-ahead operations have to be undone if former transaction aborts!



Operations Potentially Executed under Wrong Assumptions

Dealing with Look-Ahead Transactions

- Commit of look-ahead transactions is delayed until the outcome of the former transaction is known
  ⇒ constraint on serialization order



Former Transaction

Look-Ahead Transaction

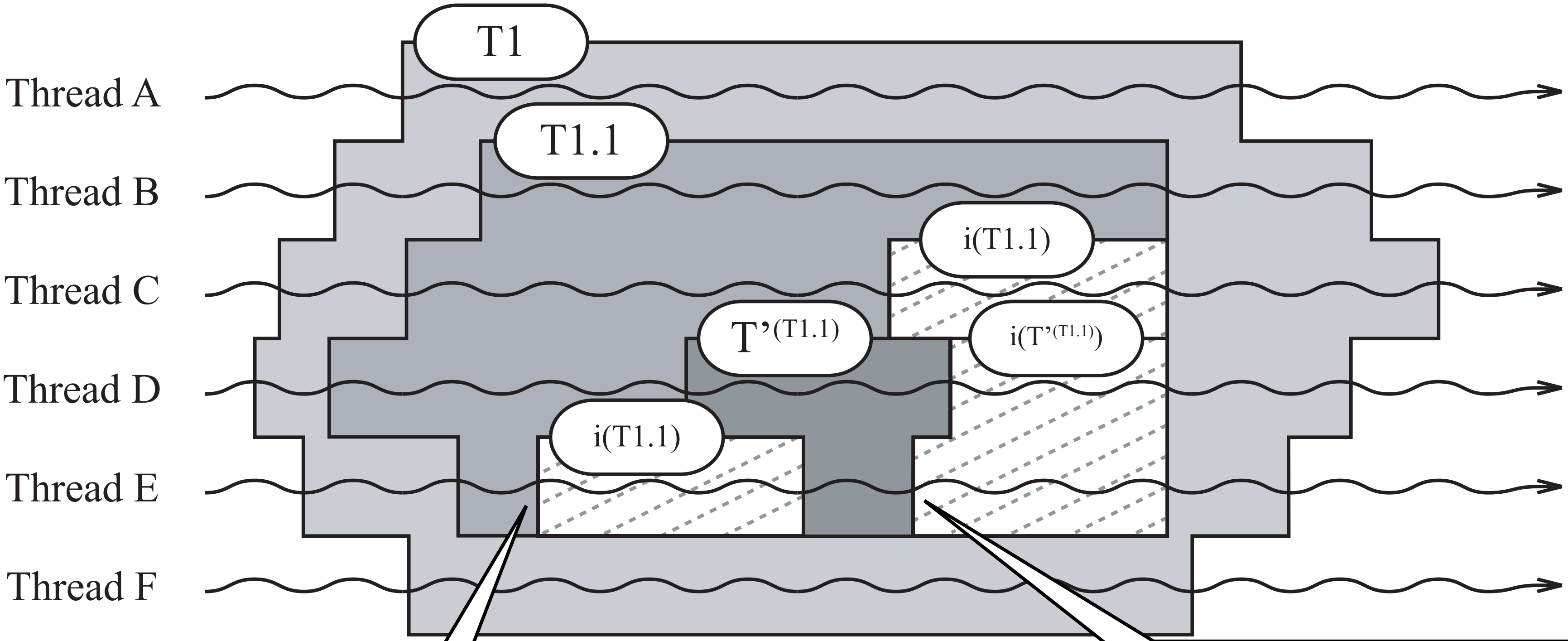
What to do with Lone Code?

T'$^{(T1.1)}$ must commit after T1.1

Dealing with Lone Code

- Automatic encapsulation of lone code inside an implicit transaction
  - Implicitly created by first look-ahead participant
  - Other look-ahead participants join
  - Isolate the look-ahead operations from non-look-ahead participants
- In case of an abort of the former transaction, the implicit transaction is aborted as well
  - No effect on non-look-ahead participants

Transactional Lone Code Encapsulation

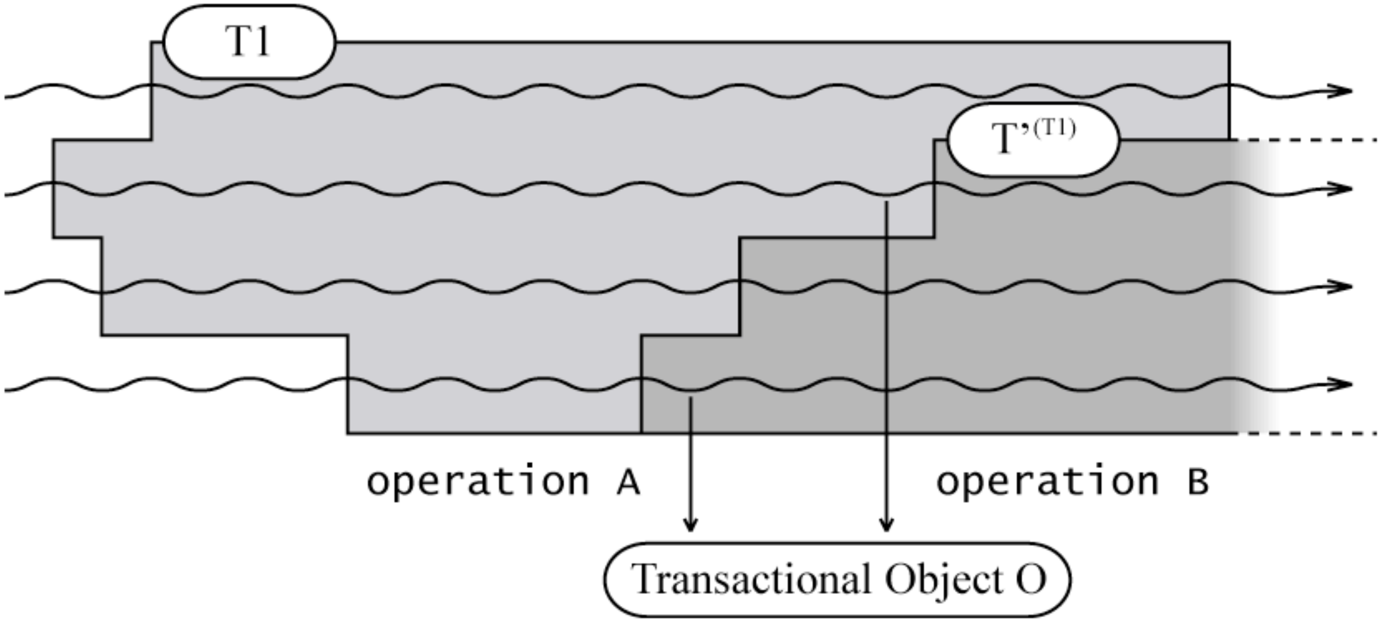


Lone code encapsulated in implicit transaction

A new implicit transaction is created for each new look-ahead

Dealing with Transactional Objects

- A look-ahead transaction might access an object that is going to be accessed by a former transaction in the future



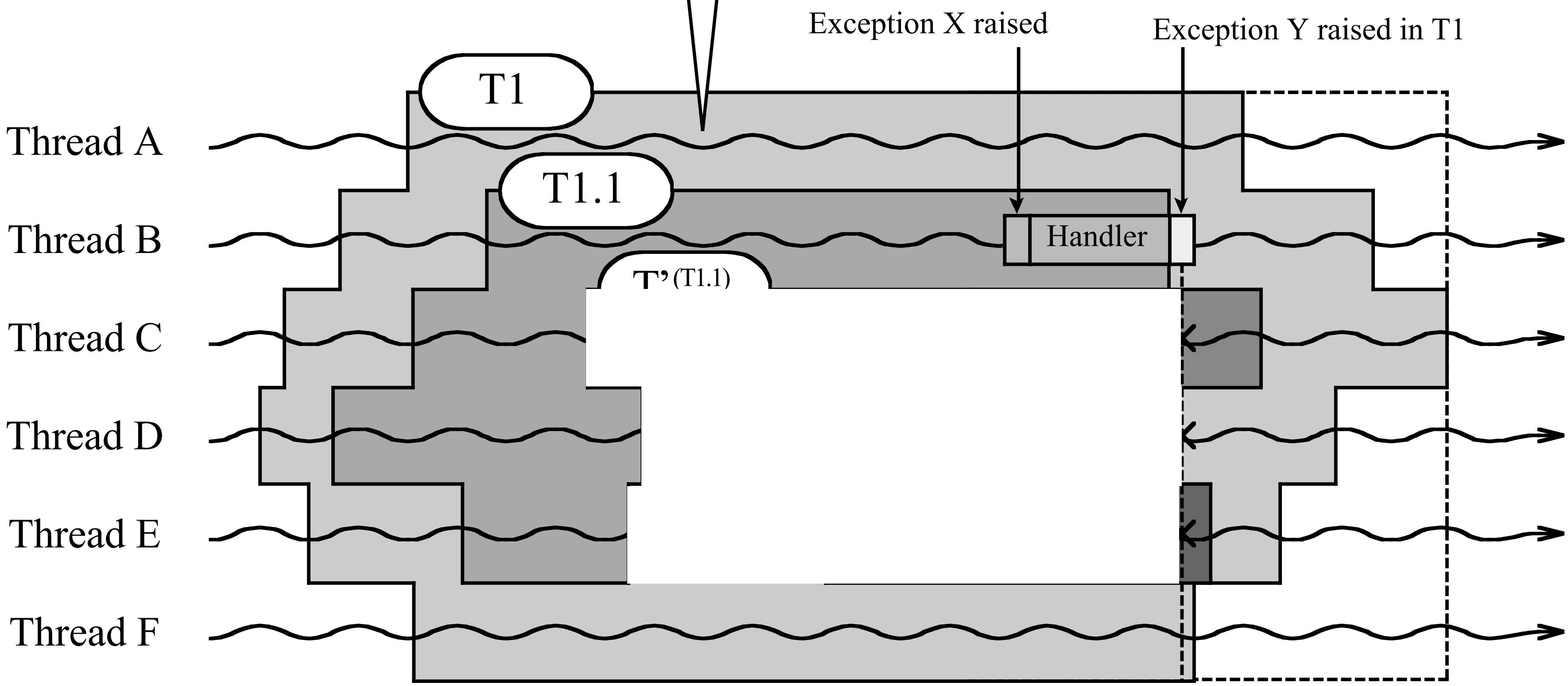
Concurrency Control of Object O
Must Be Made Look-Ahead Aware

Optimizing Pessimistic Concurrency Control

- Pessimistic Concurrency Control
  - Before allowing a transaction to perform an operation on a transactional object, it has to get the permission to do so
  - If there is a potential conflict with any other ongoing transaction, access is denied
  - Block / abort / notify the calling transaction
- A Look-ahead transactions should not cause a former transaction to abort, because it depends on the former transaction to commit
  - Pessimistic concurrency control must be modified
  - If the conflicting operation is a look-ahead transaction, then abort the look-ahead!

Dealing with Exceptions: Case 1

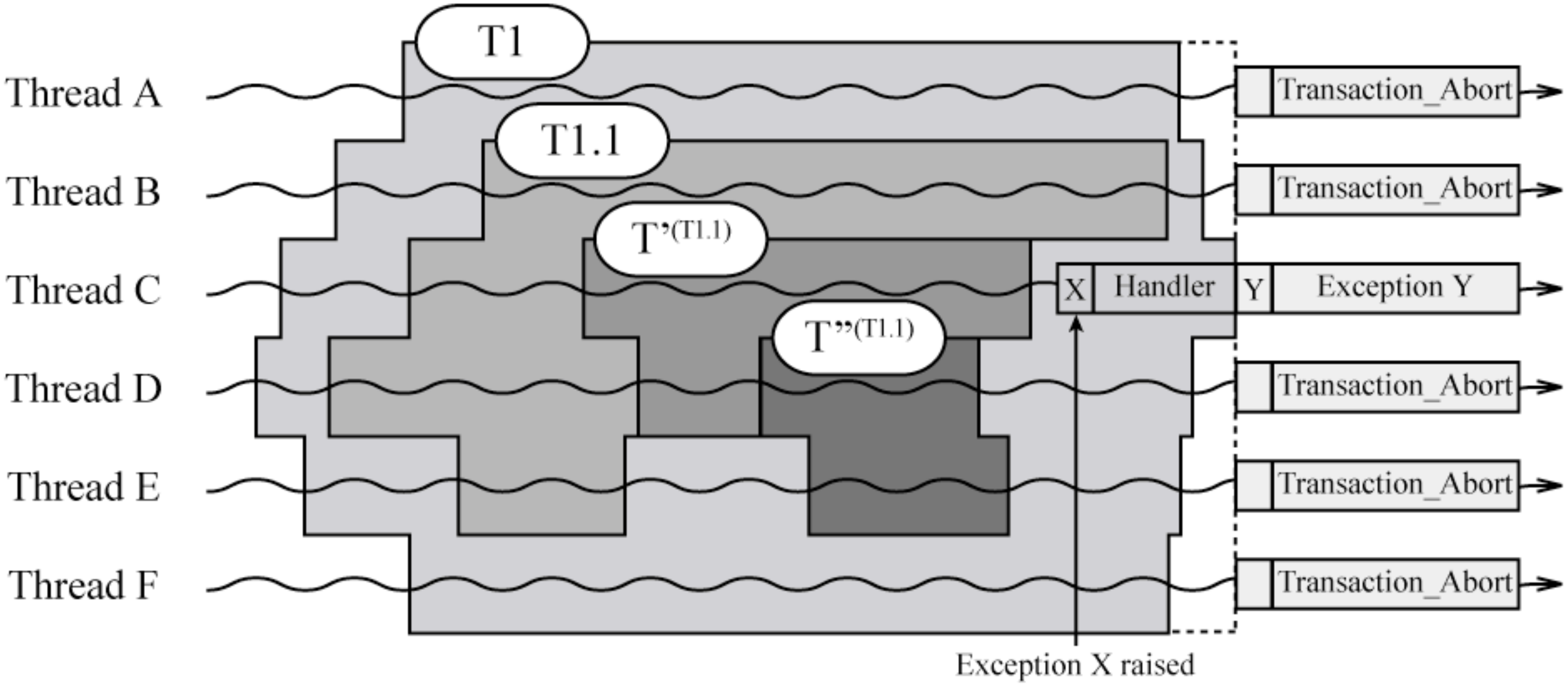
Threads A & F are isolated from B - E thanks to implicit transactions! They do not speculate on commit of $T_{1.1}$



Operations Executed Under
Wrong Assumption

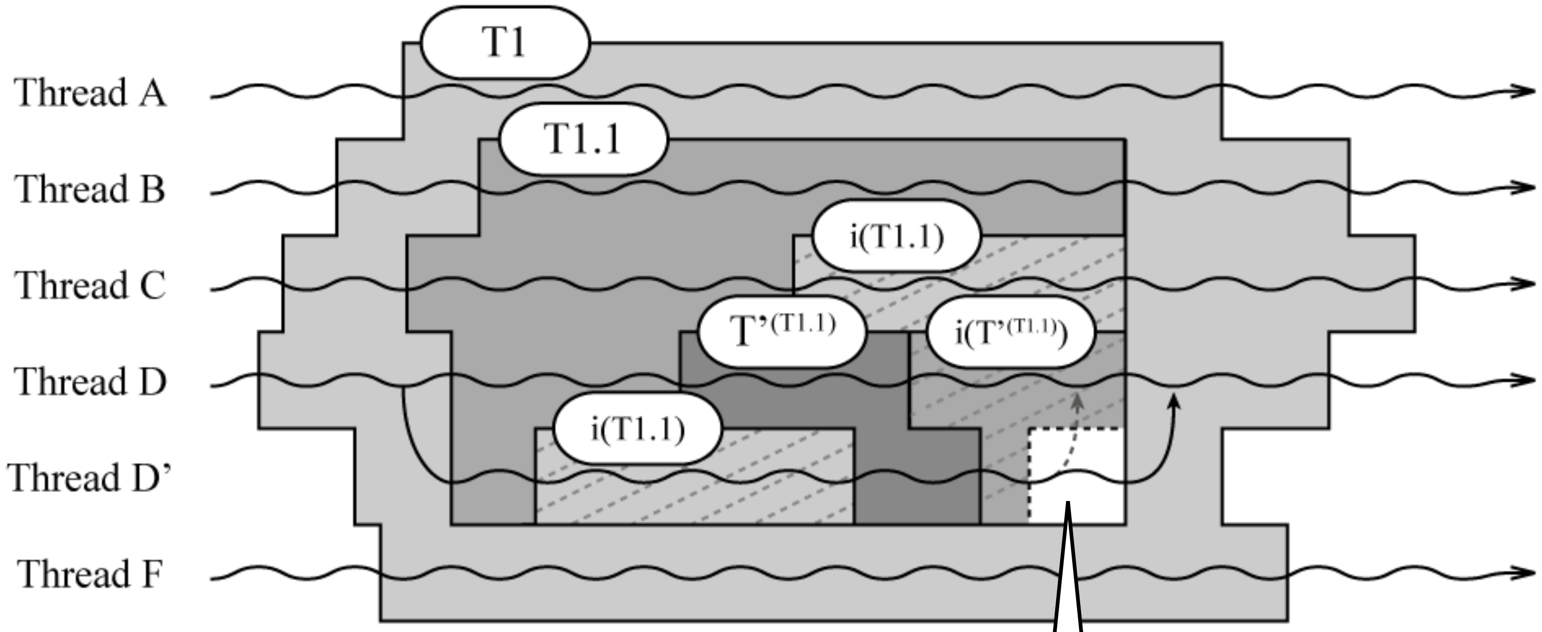
Dealing with Exceptions: Case 2

- First block, then, if resource conflict is detected, abort look-ahead, else handle exception



Dealing with Spawned Participants

- Creation and termination of threads is delayed until implicit transaction ends



Termination of Thread D' postponed until former transaction commits (end of implicit transaction)

Dealing with Joining and Nesting

- Joining rules for look-ahead participants
  - Joining of non-look-ahead transactions blocks the look-ahead participant until the former transaction commits
  - Prevent cascading aborts
  - Joining of look-ahead transactions is allowed
- Nesting
  - Looking ahead over different nesting levels is supported
  - Look-ahead from top-level transaction is supported as well

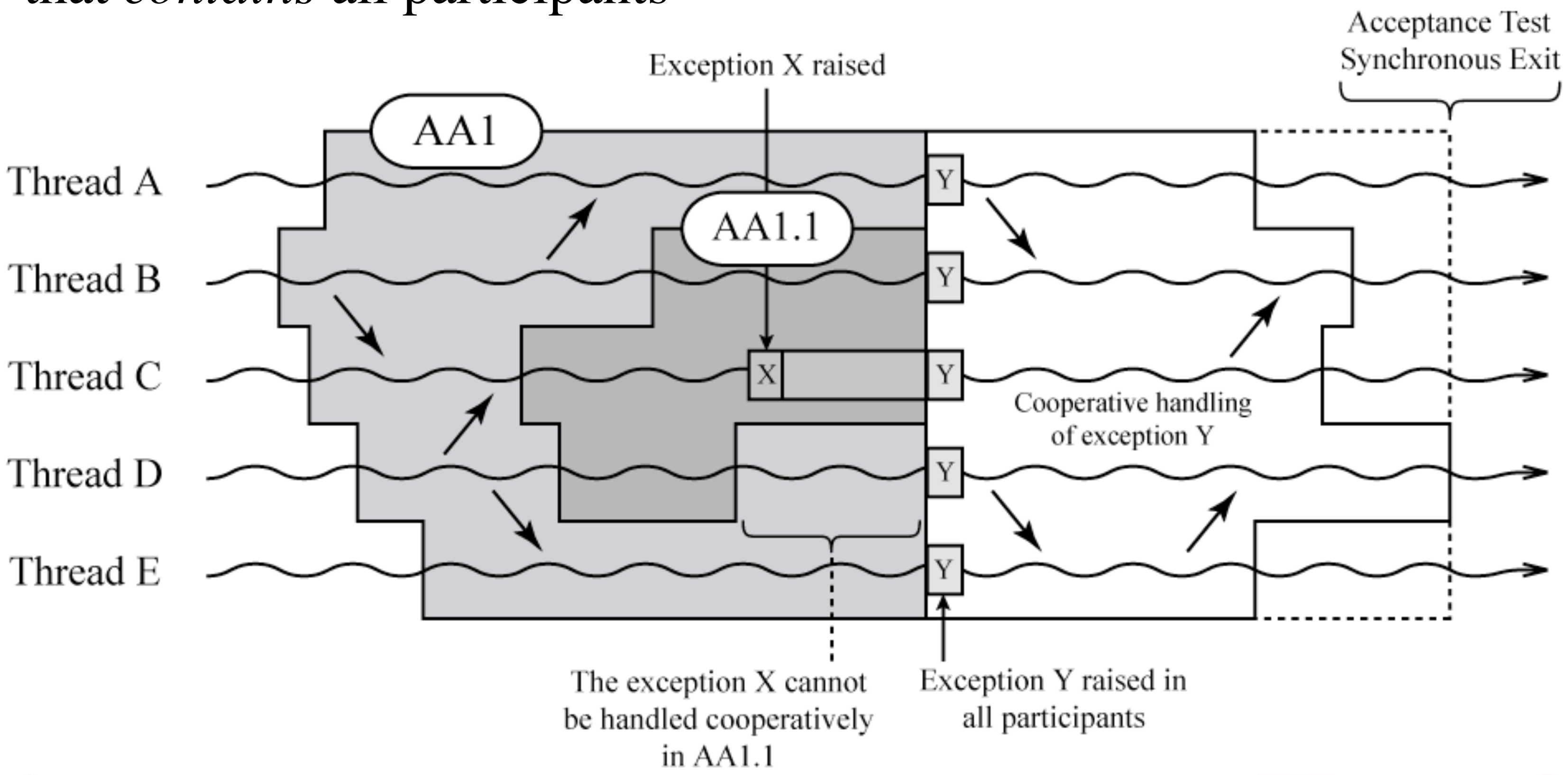
Look-Ahead Conclusions

- Look-ahead improves performance for “fast” participants of OMTTs
  - Transparent for the application programmer
- Non-trivial implementation consequences
  - Transaction commit dependencies
  - Concurrency control must be aware of look-ahead
- Future Work
  - Dynamic switching between standard and look-ahead execution depending on run-time information
  - Implementation of look-ahead for AspectOPTIMA

McGill

Look-Ahead for Atomic Actions [Rom01]

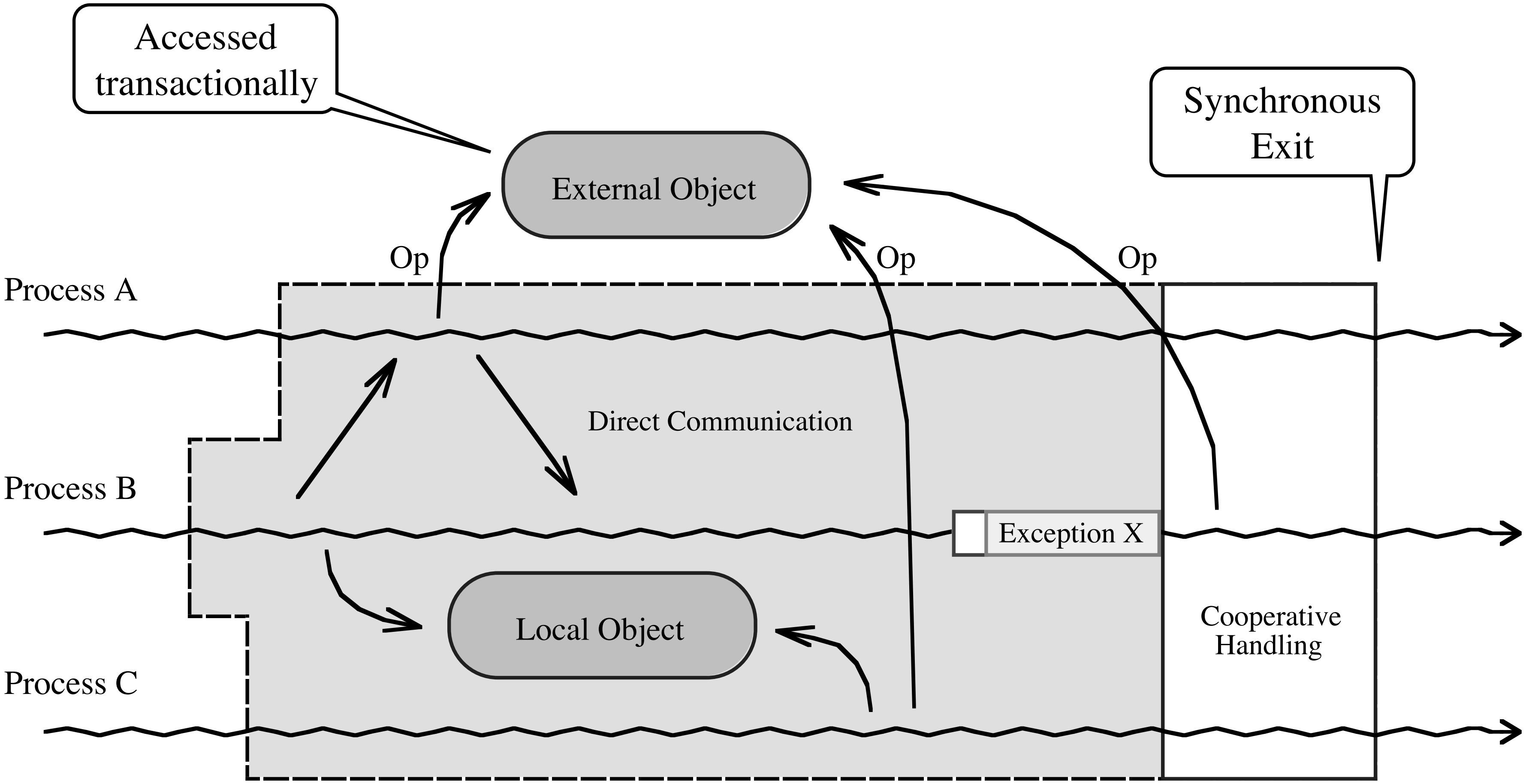
- Participants can leave (if there is a containing action)
- In case of an exception, handling is initiated at the level of the action that *contains* all participants



Coordinated Atomic Actions [XRR+95]

- Atomic actions with external objects
  - Each CA action has an associated transaction. External objects are accessed with transactional semantics
- Exception handling
  - Structured exception handling following the ideas of idealized fault tolerant component
  - Concurrent exception resolution and coordinated handling
- Disadvantages
  - Fixed number of participants
  - Not possible to create threads in the inside
  - Exception handling always coordinated / global

Coordinated Atomic Actions (2)



Design Diverse Extended Models

- N-version Programming Extensions
- Distributed Recovery Blocks
- Consensus Recovery Blocks
- Two-Pass Adjudicator
- Self-Configuring Optimal Programming

N-Version Programming Extensions

- Acceptance Voting [A89]
  - Only results that pass an acceptance test are voted upon
- N-Version Programming with Tie Breaker and Acceptance Test [TM93]
  - Compare the two fastest versions
  - If they match, proceed
  - Else wait for all results and vote, then execute acceptance test

Distributed Recovery Blocks [K84]

- Provide hardware and software fault tolerance for Real-Time systems

```
    ensure Acceptance Test on Node 1 or Node 2
   by Primary on Node 1 or Alternate on Node 2
else by Alternate on Node 1 or Primary on Node 2
         else signal failure exception
```

- Concurrent execution of the two algorithms
- If primary fails the AT, then the alternate result is used
- If both fail, backward error recovery is applied and the roles are interchanged
- Watchdogs monitor the local execution and the execution of the other node

Consensus Recovery Block [SGM83]

- N-version and recovery block combined
- The N versions are ranked with respect to their reliability
- All versions are run concurrently, and the result is voted upon
- If the voter fails, then the highest ranked version's result is submitted to an acceptance test, and so on…
- Idea: Reduce importance of acceptance test, and be able to handle cases where N-version fails due to MCR

Two-Pass Adjudicator [P92]

- Design and data diverse technique
- First pass
  - N-version programming
- If voting not successful, then perform a second pass
  - Re-express input data
  - Execute the N versions again with re-expressed input

Self-Configuring Optimal Programming

- Idea
  - Reduce cost of fault tolerance (time and space)
  - Adjust trade-off dynamically at run-time
- Select a set of versions to be run in phase one, according to the number of results needed to make a decision, and the number of processors available
- If more results are needed, add additional phases

References (1)

- [KRS01]
  Kienzle, J.; Romanovsky, A.; Strohmeier, A.: “Open Multithreaded Transactions: Keeping Threads and Exceptions under Control”. In *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems*, pp. 197 – 205, IEEE Computer Society Press, Los Alamitos, California, USA, 2001.
- [KJRP01]
  Kienzle, J.; Jiménez-Peris, R.; Romanovsky, A.; Patiño-Martinez, M.: “Transaction Support for Ada”. In *Reliable Software Technologies - Ada-Europe’2001*, pp. 290 – 304, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.
- [A89]
  Athavale, A.: “Performance Evaluation of Hybrid Voting Schemes”, M.S. thesis, North Carolina State University, Department of Computer Science, 1989.
- [TM93]
  Tai, A. T.; Meyer, J. F.; Aviziensis, A.: “Performability Enhancement of Fault-Tolerant Software”, IEEE Transactions on Reliability 42(2), pp. 227 - 237, 1993.

References (2)

- [K84]
  Kim, K. H.: “Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults”, Proceedings of the Fourth International Conference on Distributed Computing Systems, pp. 526 - 532, 1984.
- [SGM83]
  Scott, R. K.; Gault, J. W.; Mc Allister, D. F.: “The Consensus Recovery Block”, Proceedings of the Total Systems Reliability Symposium, Gaithersburg, MD, pp. 95 - 104, 1983.
- [P92]
  Pullum, L. L.: ”Fault-Tolerant Software Decision-Making Under the Occurrence of Multiple Correct Results”, Ph.D. thesis, Southeastern Institute of Technology, 1992.
- [SMR93]
  S. Shrivastava, L. Mancini and B. Randell: “The Dualty of Fault-Tolerant System Structures”, Software Practice and Experience, Volume 23(7), pages 773 - 798, July 1993.

McGill

References (3)

- [Obj00]
  Object Management Group, Inc.: Object Transaction Service, Version 1.1, May 2000.
- [SSK03]
  Silaghi, R.; Strohmeier, A.;Kienzle, J.: “Porting OMTTs to CORBA”, International Symposium on Distributed Objects and Applications, DOA 2003, to be published.
- [HKM+94]
  Haines, N.; Kindred, D.; Morrisett, J. G.; Nettles, S. M.; Wing, J. M.: “Composing First-Class Transactions”, ACM Transactions on Programming Languages and Systems 16(6), Nov 1994, pp. 1719 – 1736.
- [JPPMA00]
  Jiménez-Peris, R.; Patiño-Martinez, M.; Arévalo, S.: “TransLib: An Ada 95 Object-Oriented Framework for Building Transactional Applications”, Computer Systems: Science & Engineering Journal 15(1), 2000, pp. 7 – 18.
- [XRR+95]
  Xu, J.; Randell, B.; Romanovsky, A.; Rubira, C. M. F.; Stroud, R. J.; Wu, Z.: “Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery”, in FTCS-25: 25th International Symposium on Fault Tolerant Computing, pp. 499 – 509, Pasadena, California, 1995.

References (4)

[CR86] Campbell, R. H.; Randell, B.: “Error Recovery in Asynchronous Systems”, *IEEE Transactions on Software Engineering* SE-12(8), August 1986, pp. 811 – 826.

[KRS01] Kienzle, J.; Romanovsky, A.; Strohmeier, A.: “Open Multithreaded Transactions: Keeping Threads and Exceptions under Control”. In *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems*, pp. 197 – 205, IEEE Computer Society Press, Los Alamitos, California, USA, 2001.

[Rom01] Romanovsky, A.: “Looking Ahead in Atomic Actions with Exception Handling”, 20th Symposium on Reliable Distributed Systems (SRDS 2001), October 21-28, New Orleans, p.142-151, 2001.

[K03] Kienzle, J.: *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers, 2003.