Two-Tier vs Three-Tier Architecture

Comparison of Client Side, Middleware Side and Server Side

Two-Tier Architecture

- Structure:
- Client $\leftrightarrow$ Server (Database)

- Client directly communicates with database.
- Suitable for small-scale applications.

Two-Tier Architecture – Layers

- Client Side:
- • User Interface
- • Business Logic

- Server Side:
- • Database
- • Data processing

- Middleware Side:
- • Not present

Three-Tier Architecture

- Structure:
- Client ↔ Middleware ↔ Server (Database)

- Middleware handles business logic and security.
- Used for large-scale systems.

Three-Tier Architecture – Layers

- Client Side:
- • User Interface only

- Middleware Side:
- • Business Logic
- • Validation
- • Security

- Server Side:
- • Database storage

Comparison: Two-Tier vs Three-Tier

- Client Side:
- • Two-Tier: UI + Logic
- • Three-Tier: UI only

- Middleware:
- • Two-Tier: No
- • Three-Tier: Yes

- Server Side:
- • Two-Tier: DB + Logic
- • Three-Tier: DB only

Advantages

- Two-Tier:
- • Simple design
- • Fast for few users

- Three-Tier:
- • Scalable
- • Secure
- • Easy maintenance

- Two-Tier fits small applications.

- Three-Tier suits enterprise and web applications.

Transaction Servers:

Transactions:

- application design philosophy that guarantees robustness in distributed systems.
- Under the control of a TP Monitor, a transaction can be managed from its point of origin—typically on the client—across one or more servers, and then back to the originating client
- When a transaction ends, all the parties involved are in agreement as to whether it succeeded or failed.
- The transaction becomes the contract that binds the client to one or more servers.

ACID Properties

Atomicity

- a transaction is an indivisible unit of work: all of its actions succeed or they all fail
- The actions under the transaction's umbrella may include the message queues, updates to a database, and the display of results on the client's screen.
- Atomicity is defined from the perspective of the consumer of the transaction.

Consistency

- means that after a transaction executes, it must leave the system in a correct state or it must abort.
- If the transaction cannot achieve a stable end state, it must return the system to its initial state

ACID Properties

Isolation

- means that a transaction's behavior is not affected by other transactions that execute concurrently.
- The transaction must serialize all accesses to shared resources and guarantee that concurrent programs will not corrupt each other's operations.
- A multiuser program running under transaction protection must behave exactly as it would in a single-user environment.
- The changes that a transaction makes to shared resources must not become visible outside the transaction until it commits.

Durability

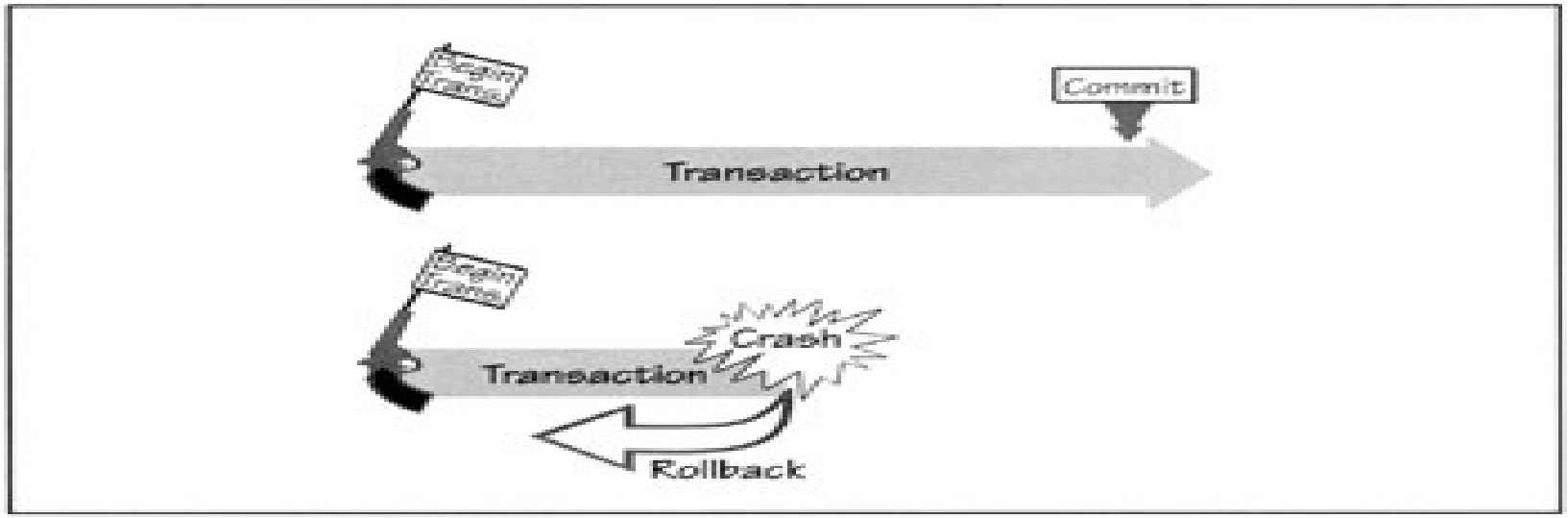
- means that a transaction's effects are permanent after it commits.
- Its changes should survive system failures.
- The term "persistent" is a synonym for "durable."

- A transaction becomes the fundamental unit of recovery, consistency, and concurrency in a client/server system.
- The application, in turn, relies on the underlying system—usually the TP Monitor—to help achieve this level of transactional integrity.
- The programmer should not have to develop tons of code that reinvents the transaction wheel.
- A more subtle point is that all the participating programs must adhere to the transactional discipline because a single faulty program can corrupt an entire system
- transaction that unknowingly uses corrupted initial data—produced by a non-transactional program—builds on top of a corrupt foundation.
- In an ideal world, all client/server programs are written as transactions.

Transaction Models:

- **Flat transaction:**
- **The Distributed Flat Transaction**
- **Chained and Nested Transactions**
- **Nested Transactions**

Flat transaction:

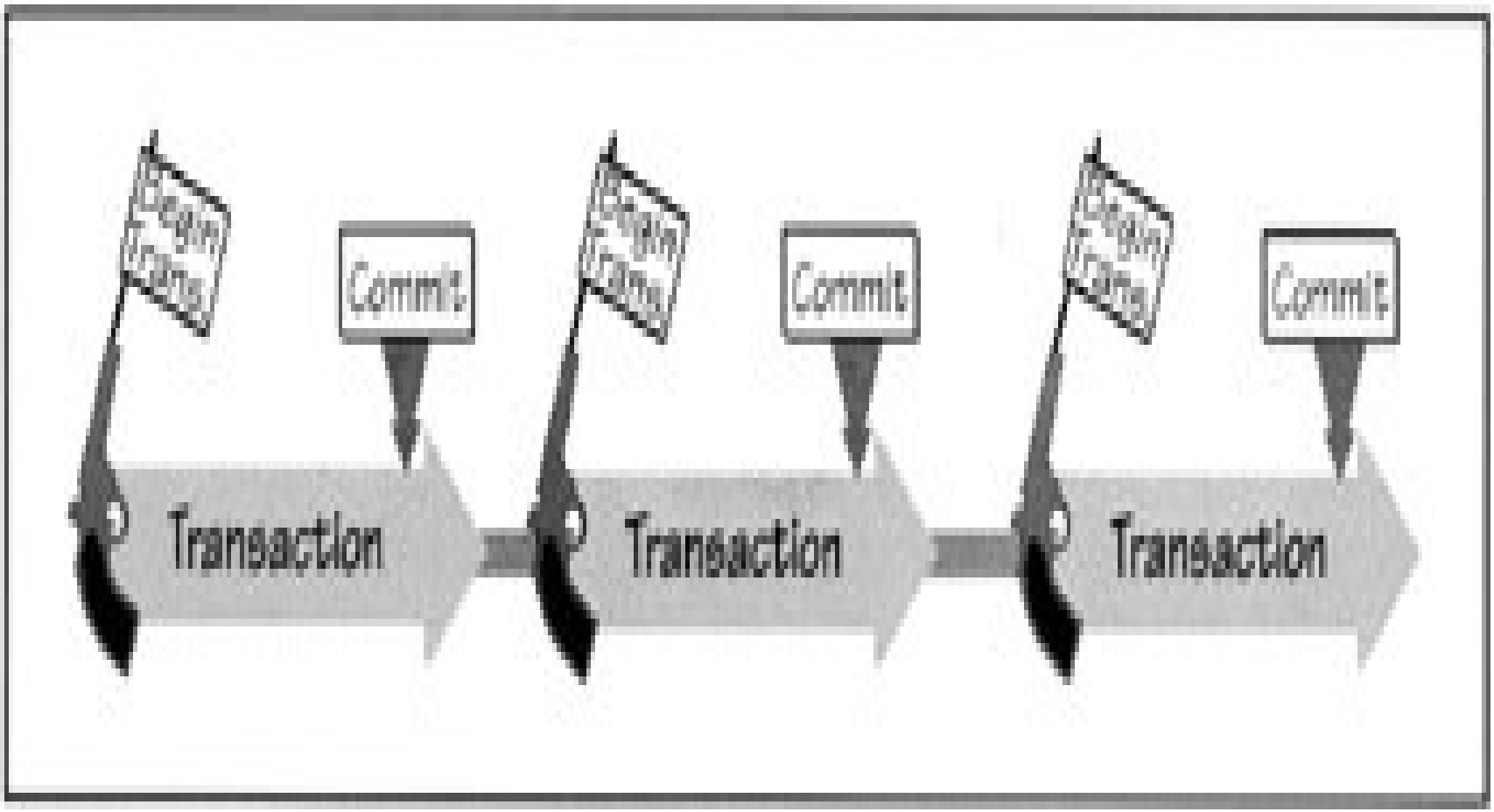


The Flat Transaction: An All-or-Nothing Proposition.

- all the work done within a transaction's boundaries is at the same level
- The transaction starts with begin_transaction and ends with either a commit_transaction or abort_transaction
- it provides an excellent fit for modeling short activities.

- The major virtue of the flat transaction is its **simplicity** and the ease with which it provides the ACID features.
- flat transaction model **does not provide the best fit in all environments**
- flat transactions using two-phase commits are usually not allowed to cross intercorporate boundaries
- A typical flat transaction does not last more than two or three seconds to avoid monopolizing critical system resources such as database locks
- OLTP client/server programs are divided into short transactions that execute back-to-back to produce results

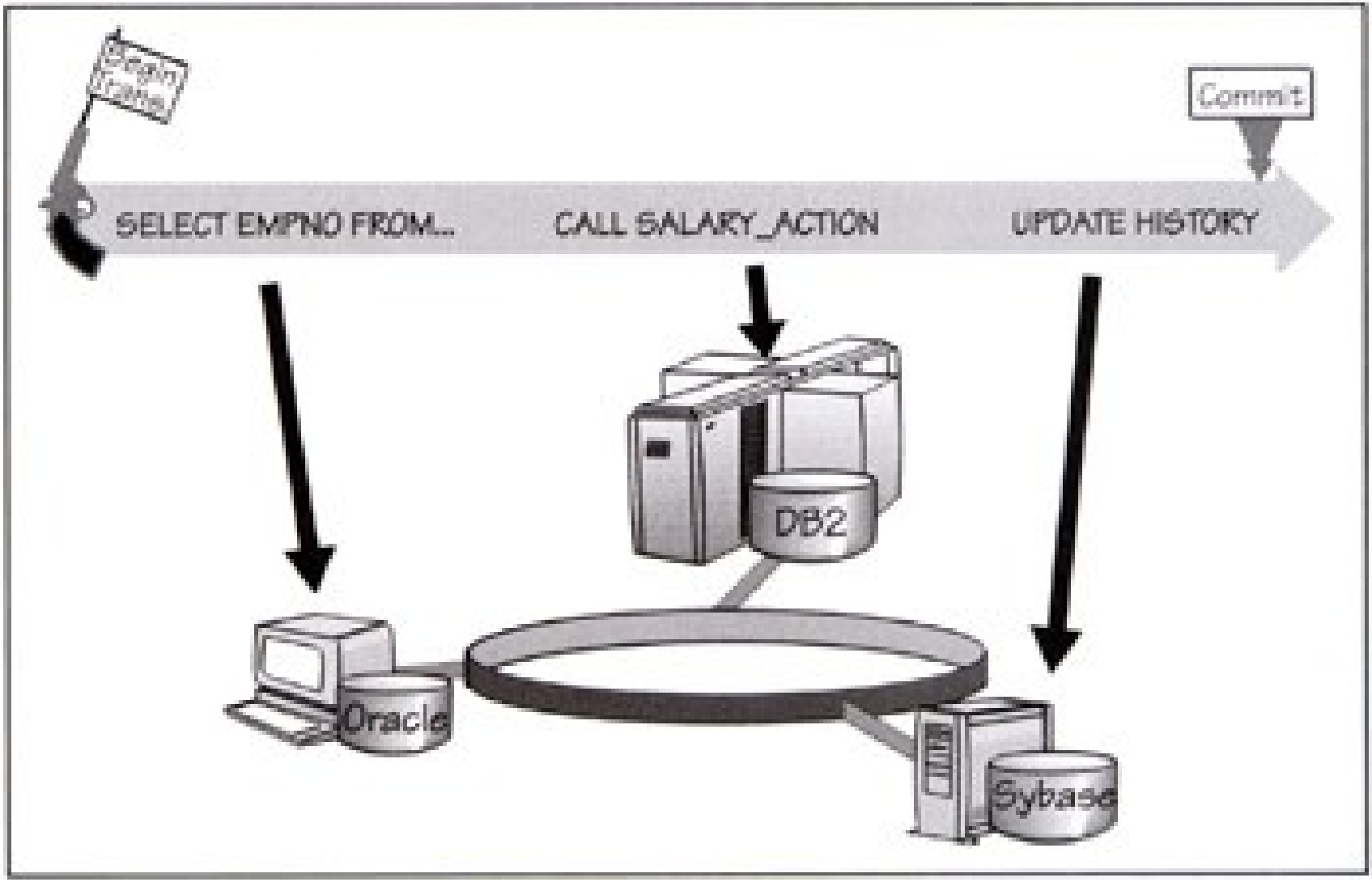
Back-to-back flat transaction



The Distributed Flat Transaction

- a **flat transaction run on multiple sites** and update resources located within multiple resource managers
- The programmer is not aware of the considerable amount of "under-the-cover" activity that's required to make the multisite transaction appear flat
- **The transaction must travel across multiple sites to get to the resources it needs**.
- Each site's TP Monitor must manage its local piece of the transaction
- Within a site, the TP Monitor coordinates the transactions with the local ACID subsystems and resource managers—including database managers, queue managers, persistent objects, and message transports

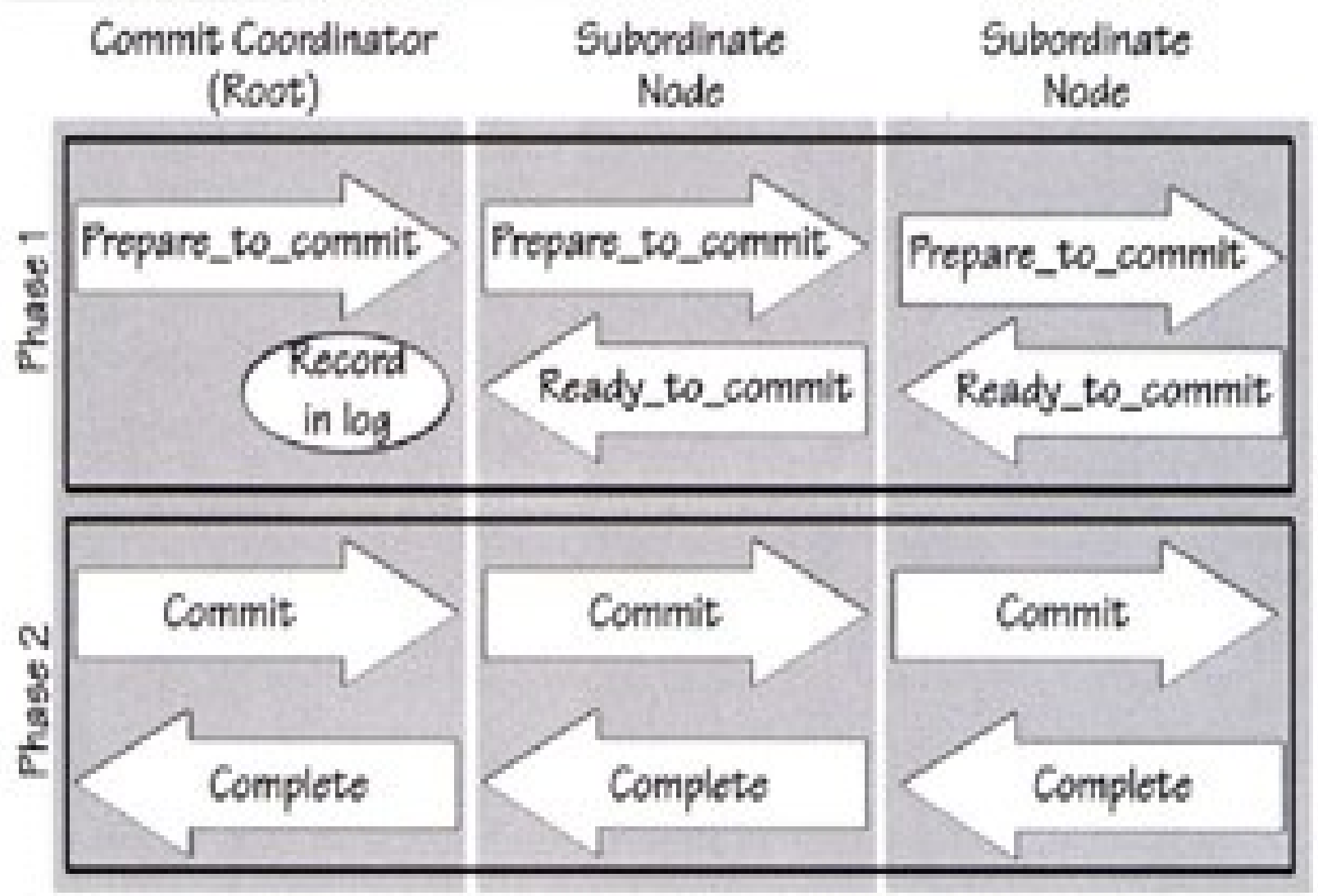
The Distributed Flat Transaction



Two-Phase Commit Protocol:

- The two-phase commit protocol is used to synchronize updates on different programs and machines so that they either all fail or all succeed.
- done by centralizing the decision to commit but giving each participant the right of veto
- If none of the parties present object, the transaction takes place
- In 1993 ISO published its OSI TP standard that defines very rigidly how a two-phase commit is to be implemented

Two-Phase Commit Protocol:



First phase of a commit

- The **commit manager node**—also known as the **root node** or the **transaction coordinator**—sends ***prepare-to-commit*** commands to all the subordinate nodes that were directly asked to participate in the transaction
- **The first phase of the commit terminates**
- root node receives ***ready-to-commit*** signals from all its direct subordinate nodes that participate in the transaction
  - This means that the **transaction has executed successfully** so far on all the nodes, and they're now ready to do a final commit

The second phase of the commit starts

- The root node makes the decision to commit the transaction—based on the unanimous yes vote. It tells its subordinates to commit.
- **The second phase of the commit terminates**
- When all the nodes involved have safely committed their part of the transaction and made the **transaction durable.**
  - The root receives all the confirmations and can tell its client that the transaction completed.

The two-phase commit aborts

- if any of the participants return a refuse indication, meaning that their part of the transaction failed
- **Limitations of two phase commit**
  - **Performance overhead**
  - **Hazard windows**, where certain failures can be a problem. For example, if the root node crashes after the first phase of the commit, the subordinates may be left in disarray.

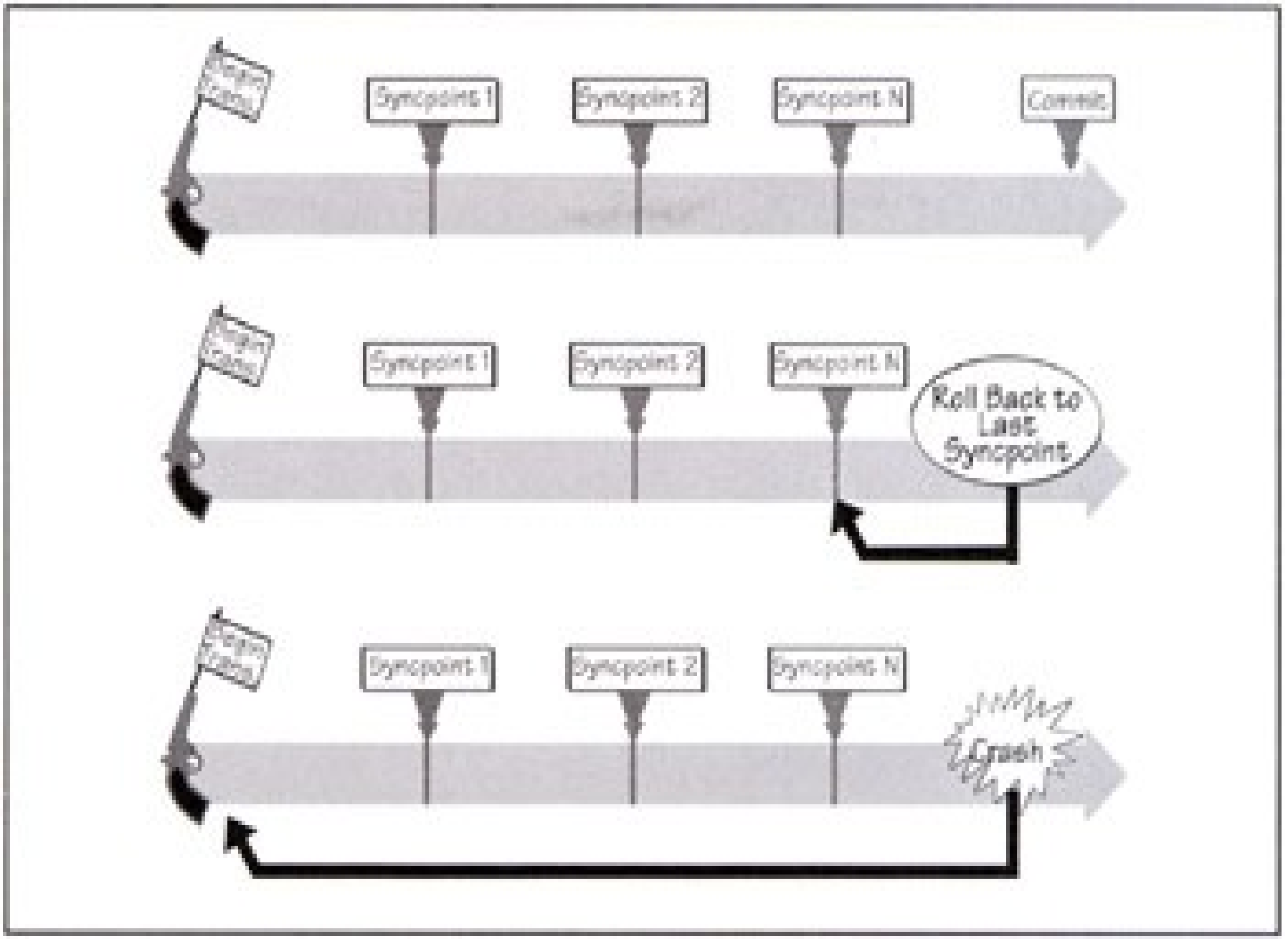
Limitations of flat transaction:

- Compound business transactions that need to be partially rolled back
- Business transactions with humans in the loop
- Business transactions with a lot of bulk
- Business transactions that span across companies or the Internet.

Chained and Nested Transactions

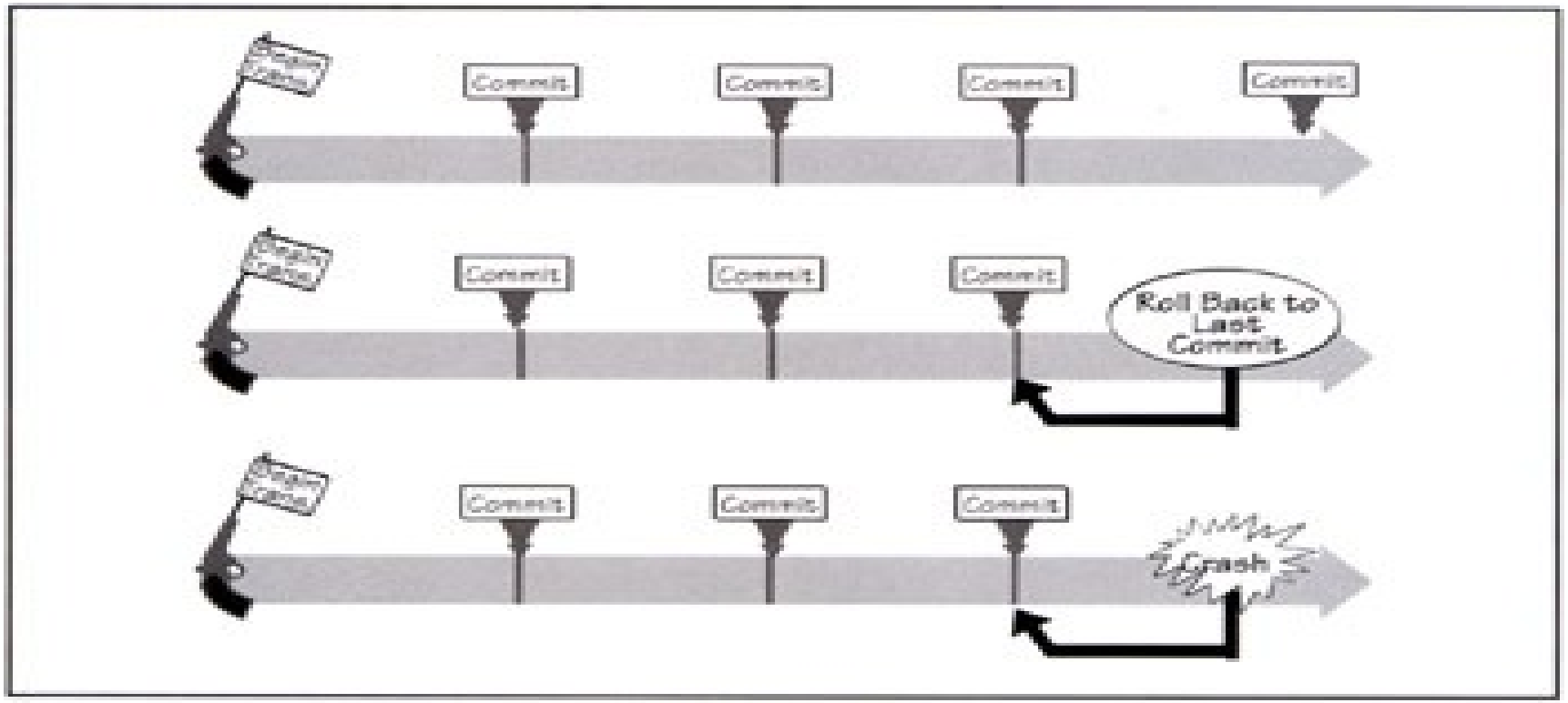
Syncpoints

- The simplest form of chaining is to use syncpoints—also known as savepoints—within a flat transaction that allow periodic saves of accumulated work
- The syncpoint lets you roll back work and still maintain a live transaction
- In contrast, a commit ends a transaction
- Syncpoints also give you better granularity of control over what you save and undo.
- But the big difference is that the commit is durable while the syncpoint is volatile
- If the system crashes during a transaction, all data accumulated in syncpoints is lost


Syncpoint 1
Syncpoint 2
Syncpoint N
Commit
Syncpoint 1
Syncpoint 2
Syncpoint N
Roll Back to Last Syncpoint
Syncpoint 1
Syncpoint 2
Syncpoint N
Crash

Chained transactions

- **Chained transactions** are a variation of syncpoints that make the accumulated work durable



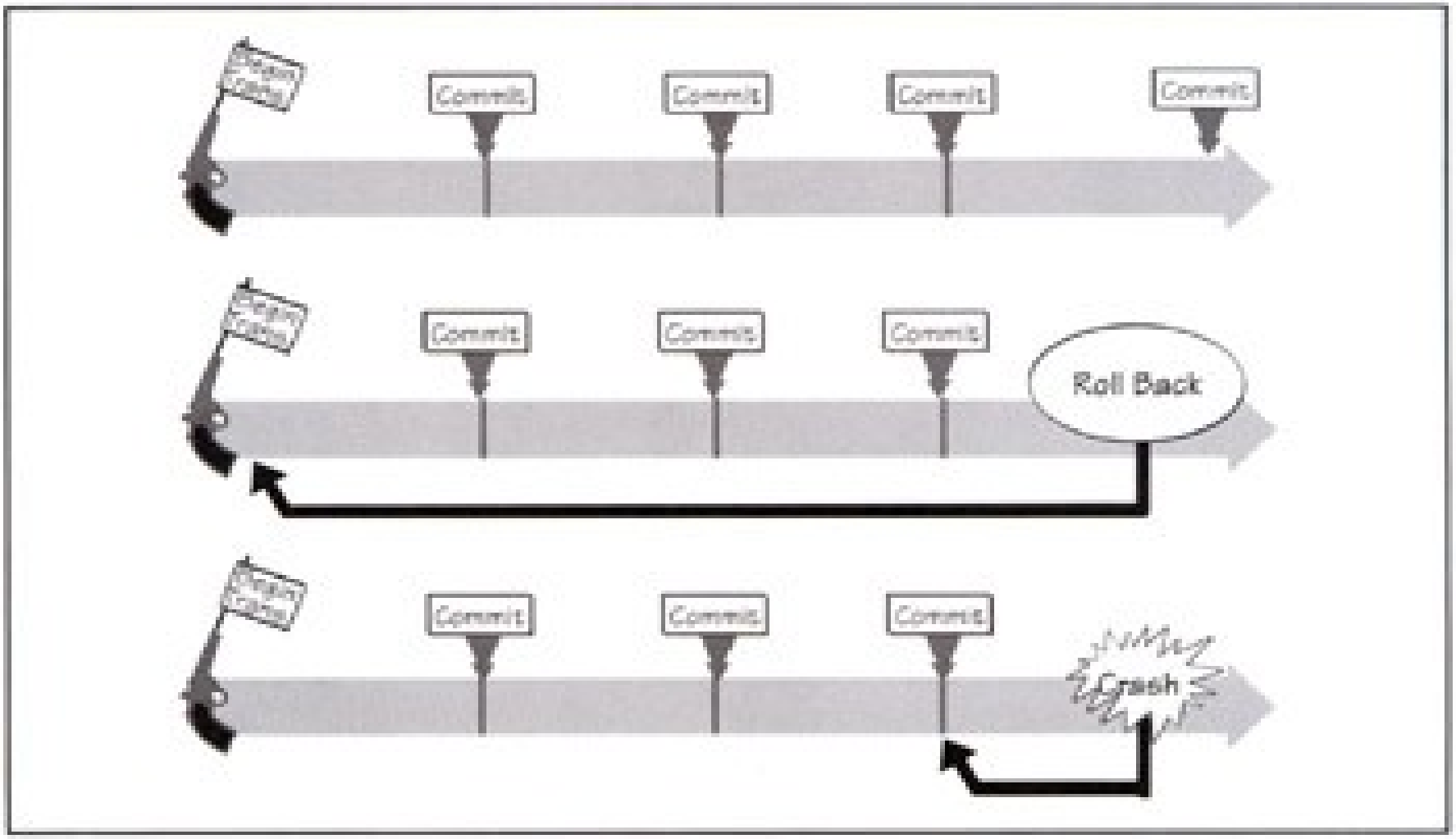
Chained transactions

- They allow you to commit work while staying within the transaction
- But the ability to roll back an entire chain's worth of work is lost

Sagas

- **Sagas** extend the chained transactions to let you roll back the entire chain
- They do that by maintaining a chain of compensating transactions
- You still get the crash resistance of the intermediate commits,
- you have the choice of rolling back the entire chain under program control
- treat the entire chain as an atomic unit of work.

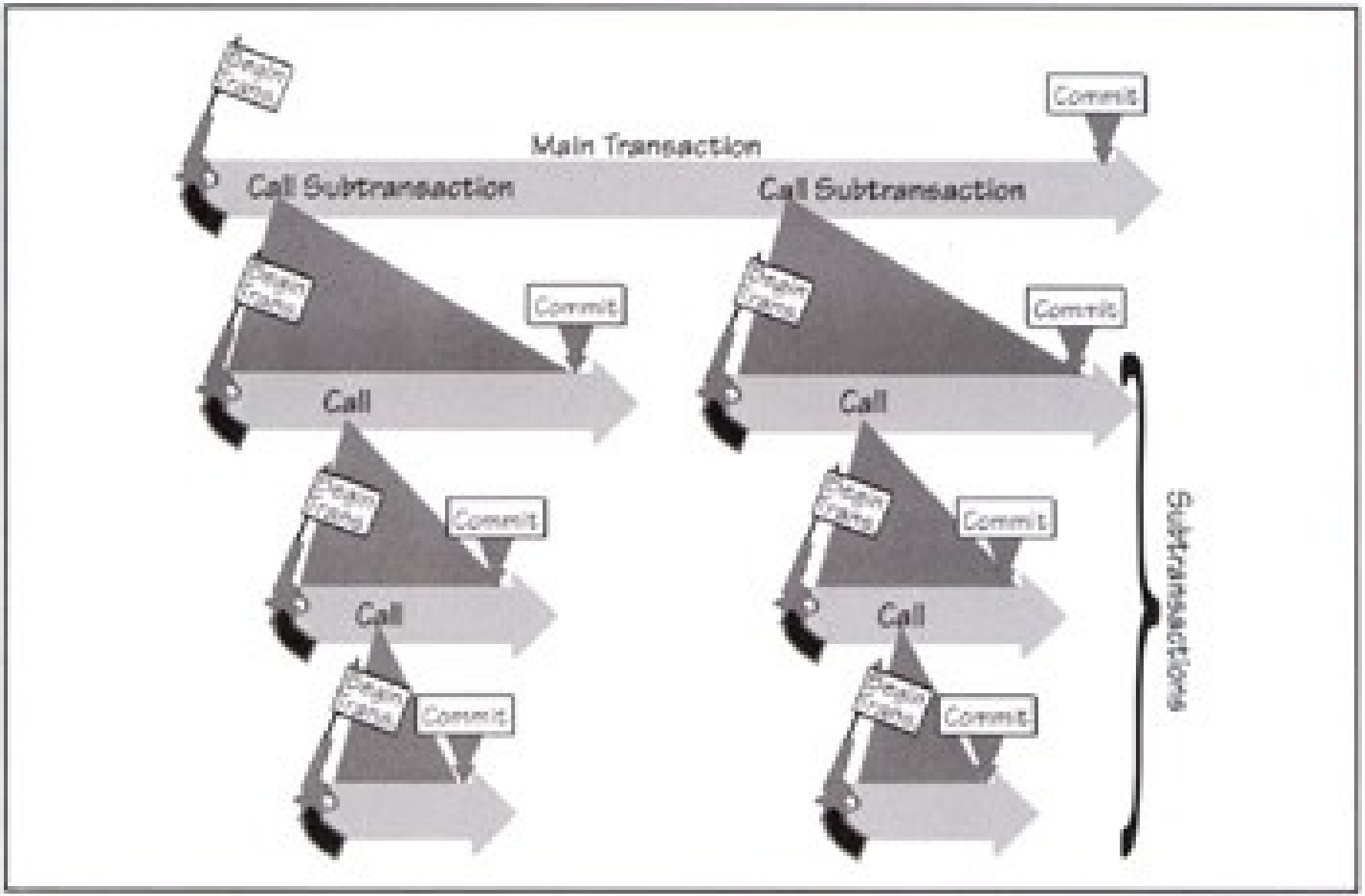
Sagas


Commit
Commit
Commit
Commit
Commit
Commit
Commit
Roll Back
Commit
Commit
Commit
Crash

Nested Transactions

- Nested Transactions provide the ability to define transactions within other transactions.
- by breaking a transaction into hierarchies of "subtransactions"
- The main transaction starts the subtransactions, which behave as dependent transactions
- A subtransaction can also start its own subtransactions, thus making the entire structure very recursive

Nested Transactions



THANK YOU