ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 08 Transaction Log

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 10, 2025

Outline

Transaction Log

Implementing a Transaction Log File

Reading Assignment

- This lecture: 5
- Next lecture: 4

Outline

Transaction Log

Implementing a Transaction Log File

Services as Finite State Machines

- ▶ Computations can be modeled as finite state machines (FSMs)
- ▶ Networked services like microservices
  - ▶ React to requests received via the network.
  - ▶ Update internal data structures and objects as needed.
  - ▶ Generate responses to be sent via the network.
- ▶ Services as FSMs
  - ▶ State: data model stored in data structures and objects.
  - ▶ Initial state: initial values of variables and objects.
  - ▶ Input: requests
  - ▶ Output: responses
  - ▶ State transitions: function and method calls

Persisting Resource State

- ▶ Objective: allow applications and services to start from where they were, after being shutdown.
    - ▶ In particular unexpected shutdown due to faults and failures.
- ▶ Delegate to another service that will be able to handle persistence.
    - ▶ E.g. a database service that supports the data model.
    - ▶ A good choice in practice but doesn't answer the fundamental problem.
- ▶ Make use of persistent storage devices
    - ▶ E.g. hard drives and SSDs where only binary blocks are supported.
    - ▶ A more fundamental problem we need to study today.

Persisting Resource State as Binary Blocks

- ▶ Option 1: Direct State Storage
  - ▶ Encode data structures and objects into a binary format that can be decoded later.
  - ▶ Intuitive but require efforts to design algorithms for individual data structures and objects.
- ▶ Option 2: Transaction Log
  - ▶ Store all requests as binary data in the order of their arrival.
  - ▶ Compute state from the initial state and the stored requests.
  - ▶ To encode requests is usually simple since they are just names of functions and methods plus their arguments.

Performance Considerations

▶ Storage devices are slow.
  ▶ Maximum throughput can only be achieved by sequential reads and writes – storage devices are able to optimize for such cases.
  ▶ Random accesses are limited by latency, resulting in much smaller available throughput.
▶ Direct State Storage
  ▶ Random access to the binary data is required to avoid encoding and saving the whole state every time there is an update.
  ▶ Need to reduce random accesses – not easy.
▶ Transaction Log
  ▶ To store requests as they arrive only requires sequential writes.
  ▶ To compute the state requires only sequential reads.
  ▶ Nevertheless, to store all requests may require a lot of storage, and to read and process them may require a lot of time.

Scalability Considerations

- ▶ Size and throughput of storage services can be improved by horizontal scaling.
  - ▶ <u>Replication</u> improves read throughput by making data available from multiple servers.
  - ▶ <u>Sharding</u> improves write throughput by partitioning data into different servers.
- ▶ Sharding is usually not quite difficult.
- ▶ For replication,
  - ▶ Direct State Storage
    - ▶ Too costly to replicate the whole state frequently.
    - ▶ How to only replicate updates?
  - ▶ Transaction Log
    - ▶ Replicate requests by forwarding them to other servers.
    - ▶ Each server can then compute the state by themselves.

Resilience Considerations

- ▶ Possible faults and failures.
    - ▶ Hardware failure causing loss of data.
    - ▶ Power failure in the middle of saving binary data.
- ▶ Replication helps to resolve issues of loss of data.
    - ▶ But replication won't help if it corrupts data.
- ▶ For power failures,
    - ▶ Direct State Storage
        - ▶ If there is a power failure when updating the binary data, then it is very difficult to tell what data is changed.
        - ▶ This may lead to data corruption that cannot be repaired.
    - ▶ Transaction Log
        - ▶ Storing new requests only requires to append data and will not overwrite existing data for past requests.
        - ▶ If there is power failure, either the new request is stored successfully or there is some extra data at the end that can be detected and removed without much efforts – data corruption can be avoided.

Discussions

▶ Transaction log provides better scalability and resilience.
▶ Transaction log helps troubleshooting.
  ▶ Making it possible to reproduce all system transactions.
▶ Restarting a service using transaction log may take more time than that using direct state storage.
  ▶ Need time to read and process all past requests to compute the current state.
▶ Practical solutions combine the two options to make trade-offs.
  ▶ As we will discuss for distributed database systems.

Outline

Transaction Log File for Key-Value Store

- ▶ To support two operations Put, Delete.
  - ▶ There is no need to record Get as it doesn't change the state.
- ▶ File format
  - ▶ Each request is encoded into a line.
  - ▶ Each line contains four fields delimited by tabs.
  - ▶ Sequence number: monotonically increasing to represent the order of arrival.
  - ▶ Event type: PUT or DELETE
  - ▶ Key
  - ▶ Value: for PUT only.
- ▶ Additional considerations.
  - ▶ Key/Value cannot contain tabs or newline characters.
  - ▶ A line at the end of the file without a newline character indicating a corrupted line that should be removed.

Transaction Logger

```
type TransactionLogger interface {
  WriteDelete(key string)
  WritePut(key, value string)
  Err() <-chan error
  ReadEvents() (<-chan Event, <-chan error)
  Run()
}
```

- ▶ An interface to support transaction log.
- ▶ `WriteDelete` and `WritePut` record requests.
- ▶ `ReadEvents` reads past requests when the service restarts.
  - ▶ Communication through channels: `<-chan Event` is a channel of `Event`s where past requests can be read out.
  - ▶ Reduce memory usage by <u>not</u> reading and storing all past requests at the same time.
- ▶ Run the logger in its own threads with channels.
  - ▶ Avoid racing conditions from multiple RESTful requests without using locks.

Implementing File Based Transaction Logger

```go
type FileTransactionLogger struct {
  events chan<- Event // Write-only channel for sending events
  errors <-chan error // Read-only channel for receiving errors
  lastSequence uint64 // The last used event sequence number
  file *os.File // The location of the transaction log
}
func NewFileTransactionLogger(filename string) (TransactionLogger, error) {
  file, err := os.OpenFile(filename, os.O_RDWR|os.O_APPEND|os.O_CREATE, 0755)
  ...
  return &FileTransactionLogger{file: file}, nil
}
```

- ▶ Implement `FileTransactionLogger` to store requests in file
    - ▶ Lowercase members are private.
    - ▶ Members not explictly initialized are set to `nil` or `0`.
    - ▶ Need to implement the 5 methods from the
      `TransactionLogger` interface.
- ▶ We will omit error handling to focus on functionalities when
  necessary.

Read Past Requests

```go
func (l *FileTransactionLogger) ReadEvents() (<-chan Event, <-chan error) {
  scanner := bufio.NewScanner(l.file) // Create a Scanner for l.file
  outEvent := make(chan Event) // An unbuffered Event channel
  outError := make(chan error, 1) // A buffered error channel
  go func() {
    var e Event
    defer close(outEvent) // Close the channels when the
    defer close(outError) // goroutine ends
    for scanner.Scan() {
      line := scanner.Text()
      if err := fmt.Sscanf(line, "%d\t%d\t%s\t%s",
        &e.Sequence, &e.EventType, &e.Key, &e.Value); err != nil {
        outError <- fmt.Errorf("input parse error: %w", err)
        return
      }
      l.lastSequence = e.Sequence // Update last used sequence #
      outEvent <- e // Send the event along, block if channel is full
    }
    ...
  }()
  return outEvent, outError
}
```

▶ Send event e to channel outEvent by outEvent <- e.

Write Requests to File

```go
func (l *FileTransactionLogger) WritePut(key, value string) {
  l.events <- Event{EventType: EventPut, Key: key, Value: value}
}
func (l *FileTransactionLogger) WriteDelete(key string) {
  l.events <- Event{EventType: EventDelete, Key: key}
}
func (l *FileTransactionLogger) Run() {
  l.events = make(chan Event, 16) // Make an events channel
  l.errors = make(chan error, 1) // Make an errors channel
  go func() { // start a goroutine that runs in a single thread
    for e := range l.events { // Retrieve the next Event
      l.lastSequence++ // Increment sequence number
      _, err := fmt.Fprintf(l.file, "%d\t%d\t%s\t%s\n",
        l.lastSequence, e.EventType, e.Key, e.Value)
      ...
      }
    }
  }()
}
```

▶ Thread confinement: multiple threads may call `WritePut` and
   `WriteDelete` but only a single thread will handle them.
   ▶ Synchronization via a channel without a lock.

Initialization

```go
var logger TransactionLogger
func initializeTransactionLog() error {
  logger, err := NewFileTransactionLogger("transaction.log")
  ...
  events, errors := logger.ReadEvents()
  e, ok := Event{}, true
  for ok && err == nil {
    select { // use select to read from multiple channels
    case err, ok = <-errors: // Retrieve any errors
    case e, ok = <-events:
      switch e.EventType {
      case EventDelete: // Got a DELETE event!
        err = Delete(e.Key)
      case EventPut: // Got a PUT event!
        err = Put(e.Key, e.Value)
      }
    }
  }
  logger.Run()
  return err
}
func main() {
  err := initializeTransactionLog()
  ...
```

ECE 473/573 – Cloud Computing and Cloud Native Systems, Dept. of ECE, IIT

Recording PUT Requests

```go
func keyValuePutHandler(w http.ResponseWriter, r *http.Request) {
  vars := mux.Vars(r)
  key := vars["key"]
  value, err := ioutil.ReadAll(r.Body)
  defer r.Body.Close()
  ...

  err = Put(key, string(value))
  ...

  logger.WritePut(key, string(value))
  w.WriteHeader(http.StatusCreated)
  log.Printf("PUT key=%s value=%s\n", key, string(value))
}
```

▶ DELETE is recorded in a similar way.

Summary

- Use transaction logs to store states indirectly for better scalability and resilience.