Testing Distributed Systems

Hyperledger Bootcamp Russia

Presented by Sergey Khoroshavin
Agenda

- Why testing
- Common approaches
- Property based testing
- Useful properties
- Questions
Why testing

● Catch bugs before they inflict damage
  ○ Earlier = cheaper
  ○ Easier = cheaper

● Maintainability
  ○ Good tests can become documentation that never lies
  ○ Safety net when improving existing code

● Know limitations
  ○ Performance characteristics
  ○ Memory requirements
  ○ “What if …?”
Distributed systems

- Complex
  - different entities exchanging messages...
  - ...which can arrive out of order, too late, or never
- Easy to make mistakes
- Cost of mistake can be high
- Take a lot of time to develop
- Runtime conditions can vary wildly
Common Approaches
Unit testing

● Test small isolated pieces of code
  ○ easy to write, cheap to run
  ○ easy to understand what went wrong

● Help design clean code

● Can be seen as some form of documentation
  ○ which never lies

● Written by developers
  ○ probably in TDD style
Unit testing: problems

- Lots of tests needed to get decent coverage
  - so actually not so easy to write when looking at the whole picture
- Easy to miss some edge cases
  - especially in complex systems
- Can be hard to write for legacy code
  - refactoring can help
  - "Working effectively with legacy code" by Michael Feathers
- Cannot test whole system
  - other complementary approaches needed
Unit testing: suggestions

● Test interfaces and contracts
  ○ try to refrain from accessing implementation details

● Name and organize tests sensibly
  ○ feel the difference: test_leader_election vs test_leader_election_eventually_completes
  ○ remember that you’ll end up with thousands of unit tests

● Make tests straightforward to read and write
  ○ avoid complex logic - prefer separate test cases
  ○ avoid creating complex frameworks which hide API under test

● Try TDD when writing new code
  ○ don’t follow TDD zealots blindly
  ○ find your own balance
Load testing

● (Stress) test system as a whole
● Relatively easy to write
  ○ actually it can be hard, but usually there is just one tool(set) for running load tests
  ○ as opposed to thousands of unit tests
● Can show lots of insidious problems
  ○ memory leaks
  ○ performance problems at scale
  ○ protocol flaws
● Understand stability and performance characteristics
● Written either by developers or QAAs
Load testing: problems

- Hardware to run tests can be expensive
  - imagine running load test against pool of 25 nodes for days
- Analyzing results can be hard
  - imagine analyzing 10s to 100s Gbs of logs from above mentioned tests
- Doesn’t guarantee to show all problems
- Reproducibility can be a problem
Load testing: suggestions

- **Plan in advance, follow plan**
  - don’t be overwhelmed by results

- **Test different scenarios**
  - stable vs spiky load
  - light vs DoS-like load
  - flaky or even partitioned network

- **Automate as much as possible**
  - infrastructure
  - configuration management
  - running test itself
  - gathering and preliminary analysis of results
  - Ansible + some python scripts help a lot
Integration/system testing

- Test big parts of system or system as a whole
  - can find problems which are hard/impossible to find with unit tests
  - can provide great coverage with relatively low effort
- Relatively easy to write
  - even if code is not of highest quality
- Often cheaper to run than load tests
- Can be written by developers or QAA's
Integration/system testing: problems

- Actually share a lot common with unit tests
  - especially “hard to cover all edge cases”
- Slow to run
  - CI cost can become quite high
- Easy to end up with flaky tests
- Problem analysis can be hard
Integration/system testing: suggestions

● Same as unit tests +
● Avoid “wait for some time” pattern, prefer “wait until some event happens”
● Avoid writing integration tests for every edge case covered in unit tests:
  ○ prefer a couple of integration tests for main scenarios
  ○ add integration tests for scenarios that cannot be properly covered by unit tests
Recap

- No silver bullet, need tests on all levels
Property Based Testing
Main idea

- Developer comes up with properties of system, instead of examples
- Testing framework creates hundreds of random examples and checks that properties hold
Example: Testing key-value storage

- Common approach

```python
storage = Storage()
storage['a'] = 42
assert len(storage) == 1
assert storage['a'] == 42
```

Why 'a'? Why 42?
Example: Testing key-value storage

- Adding randomization

```python
storage = Storage()
key = arbitrary_key()
value = arbitrary_value()

storage[key] = value

assert len(storage) == 1
assert storage[key] == value
```

Generated pseudorandomly, no more “arbitrary” values
Example: Testing key-value storage

- Adding induction

```python
storage = arbitrary_storage()
backup = storage.copy()
key = arbitrary_key()
value = arbitrary_value()
assume(key not in storage)

storage[key] = value

assert len(storage) == len(backup) + 1
assert storage[key] == value
```

Generated pseudorandomly, contains arbitrary number of elements

Good way to avoid too much logic and separate different test cases
Example: Testing key-value storage

- Another test case

```python
storage = arbitrary_storage(min_size=1)
backup = storage.copy()
key = choose_arbitrary(storage.keys)
value = arbitrary_value()

storage[key] = value

assert storage == backup
```

Chosen pseudorandomly from existing keys
Property based testing strengths

- Relatively small number of tests can provide good coverage
- Applicable at unit, integration and system levels
- Framework can come up with unexpected edge cases
  - and find bugs before expensive load test is performed
  - ...or before these bugs slip into release
Property based testing problems

- Harder to write (especially initially)
  - need a mind shift to think about properties instead of examples
- Can be harder to read
  - thinking about examples is easier and more natural, than thinking about properties
- Can give false feeling of confidence
  - output of `sort` should be sorted, what can go wrong?
  - `def sort(input): return [1, 2, 3]`
- Specialized framework is very desirable
Why frameworks?

- **Pseudorandom generators**
  - randomly sample from possible input space
  - with some emphasis on edge cases
  - repeatable (seed is controllable)
  - composable (create generators for complex custom types)

- **Shrinking**
  - after coming up with failing example…
  - ...try to “minimize” input so that test still fails
  - very useful when analyzing failures
Popular frameworks

- Python: **Hypothesis**
- Rust: **PropTest**
- Scala: **ScalaCheck**
- C++: **RapidCheck**
- JavaScript: **FastCheck**
- Go: **Gopter**
Stateful property based testing

- Instead of input values to functions...
- ...generate random sequences of operations
- Coming back to key-value storage example operations are “add new element” and “add existing element”
- Failure case is trace of such operations
- Shrinking still applies to help get minimal trace!
Simulation testing

- **Control all external inputs to system**
- **Abstract time through scheduler**
  - get_current_time()
  - schedule(interval, callable)
- **Use special implementation in tests**
  - run_for(duration)
    - advance time and process events until “now+duration”
  - wait_for(condition)
    - advance time and process events until condition becomes true
  - production code can think it waited for hours
    - while test run took only fraction of a second
- **Network can be simulated through scheduler**
  - send just schedules receive call with random delays
Simulation test example

- **Leader election test**
- **Arrange**
  - schedule at random interleaved times:
  - reception of incoming requests
  - leader change events
- **Act**
  - run scheduler until there are either:
    - no new events (so no further progress is possible)
    - all requests processed by all nodes
- **Assert**
  - all nodes changed leader
  - all nodes elected same leader
  - all requests are processed by all nodes in same order
Useful properties
Fuzzing

- Check that system doesn’t crash or do unexpected things no matter what input is given
- Can be seen as special case of property based testing
- So special that there are specialized tools for this job:
  - most notable AFL (American Fuzzy Lop)
  - coverage guided fuzzer
  - security oriented
  - has long list of found vulnerabilities
Test oracle

- Different implementations of same algorithm
- Optimized vs naive
  - quicksort vs bubble sort
  - asm/low-level vs plain
- Common functionality
  - persistent kv storage vs dictionary
  - merkle patricia trie vs dictionary
- Numeric vs analytic
  - test numeric solver against cases where analytical solutions exists
  - helps understand numeric limitations
Direct requirements

- Some random examples
  - setter-getter
  - store-fetch
  - elements of dictionary are unique
  - sort returns sorted list
  - base58 encoding contains only symbols from base58 alphabet
  - signature checker should accept genuine signature, reject random

- Also applicable to math
  - vector translation and rotation doesn’t affect its length
  - normalized vector should have unit length
Reverse

- Serialize-deserialize
- Encode-decode
- Encrypt-decrypt
- Invert-invert
  - matrix inverse or transpose
- Math transforms
  - coordinate
  - domain (for example Fourier)
Idempotence

- Repeated application is identical to single application
  - Sorting
  - canonical representation (filesystem path, JSON encoding, etc)
  - adding existing element to dictionary or set
  - vector normalization

- Trick with double reverse
  - normalize == encode-decode
  - check idempotence of normalize
  - can be useful when representation is ambiguous
Different paths, same result

- \( x + y = y + x \)
  - non-standard number representations (bigint, rational, etc)
  - cryptographic objects like elliptic curve points
  - set/dictionary union/intersection

- \( F(x+y) = F(x) * F(y) \)
  - linear operators
    - coordinate transforms
    - convolution
    - domain transforms
  - cross-domain operations
    - product of fourier transforms is fourier transform of convolution
    - set/dictionary merge vs adding elements one by one
Conclusion

● There is no silver bullet, need tests on all levels
● Treat tests as first class code
  ○ follow good practices
  ○ avoid anti-patterns
● Find right balance for your project
  ○ right = maximizes return of investment
● Property based testing can help a lot
  ○ approach useful both in unit and system tests
  ○ improves coverage with less effort
  ○ especially when applied to complex projects
Questions?
Useful resources

- Introduction to property based testing
- Choosing properties for property-based testing