FEUERLABS

locks

Distributed Scalable Locking

by Ulf Wiger, Co-Founder, Feuerlabs

Why? Isn't locking bad?

- No, locking arbitrates access to shared resources
- Help ensure consistency
- In short:
  When you need locks, you really need them
- Problems with locks:
  - Scalability
  - Complexity (if not made implicit)

Locking challenges

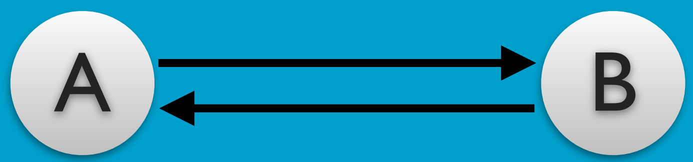
FEUERLABS

- Distribution-related
  - Deadlock/livelock detection/prevention
  - Scalability
  - Fault tolerance (incl netsplits)
- General
  - Read/write locking
  - Hierarchical locks (e.g. table/obj locks)

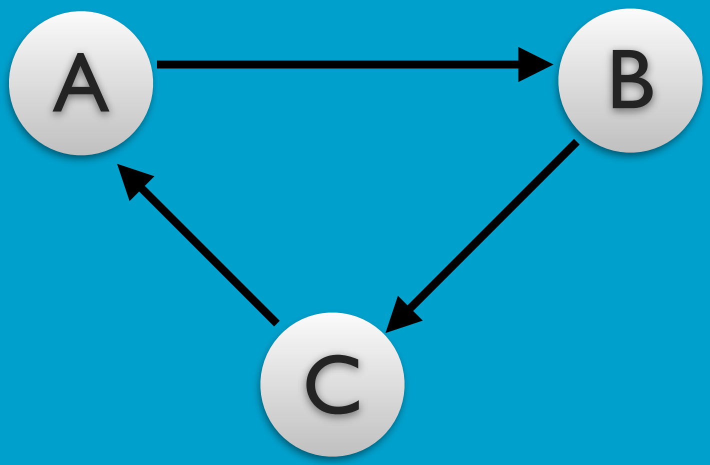
Intro: Dependency graphs

A waits for B

Deadlock



Deadlock



Distributed dependencies

FEUERLABS

- **Central dependency graph**
  - Bad (single point of failure & bottleneck)
- **Deadlock Prevention**—dependencies only one way
  - Gives phantom deadlocks
  - Unnecessary aborts/retries hurt performance
- **Probes**—replicate dependency info
  - (This is basically what we're doing)

The ‘locks’ algorithm

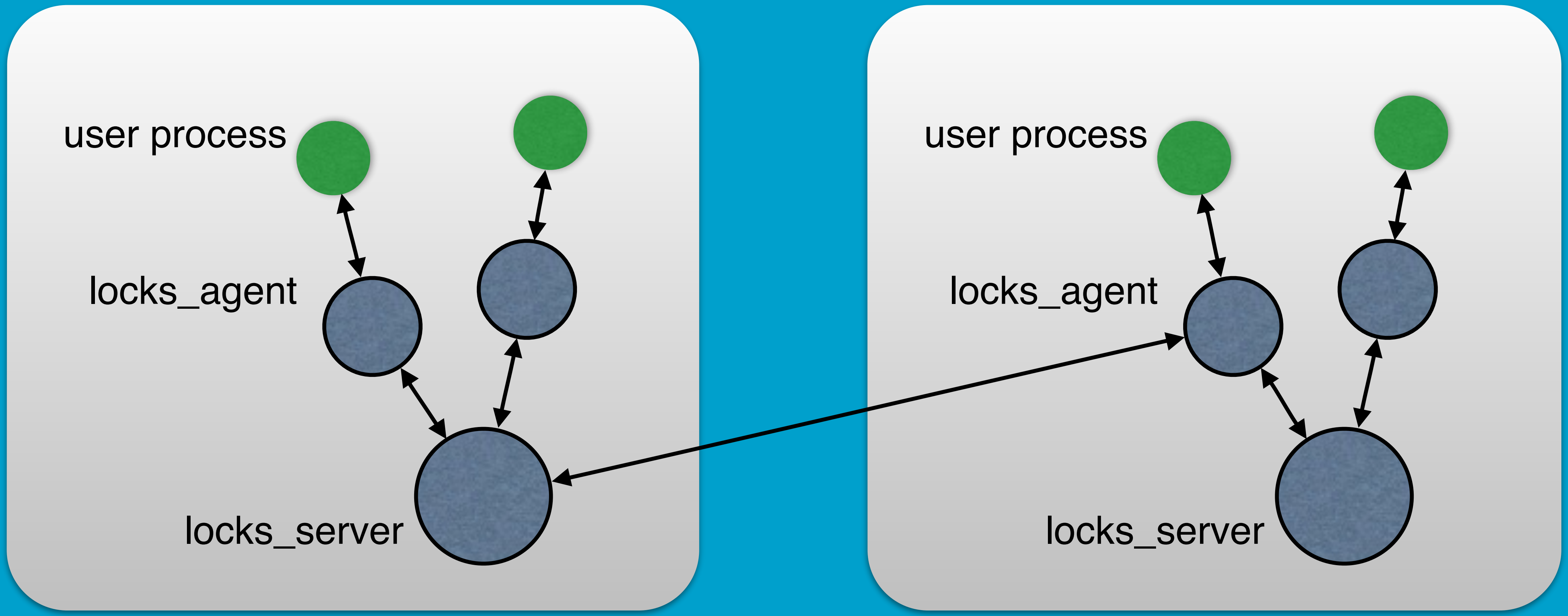
FEUERLABS

- Designed by Wiger in 1993
- Model-checked by Arts & Fredlund 1999-2000
- Extended by Wiger in 2012-13
  - Read+write locks
  - Hierarchical locks
  - Multi-node locks
  - gen_leader-type behavior

The locks implementation

FEUERLABS

- locks_agent represents a transaction context
- Asynchronous messaging, reactive design
- Locks automatically released if process dies

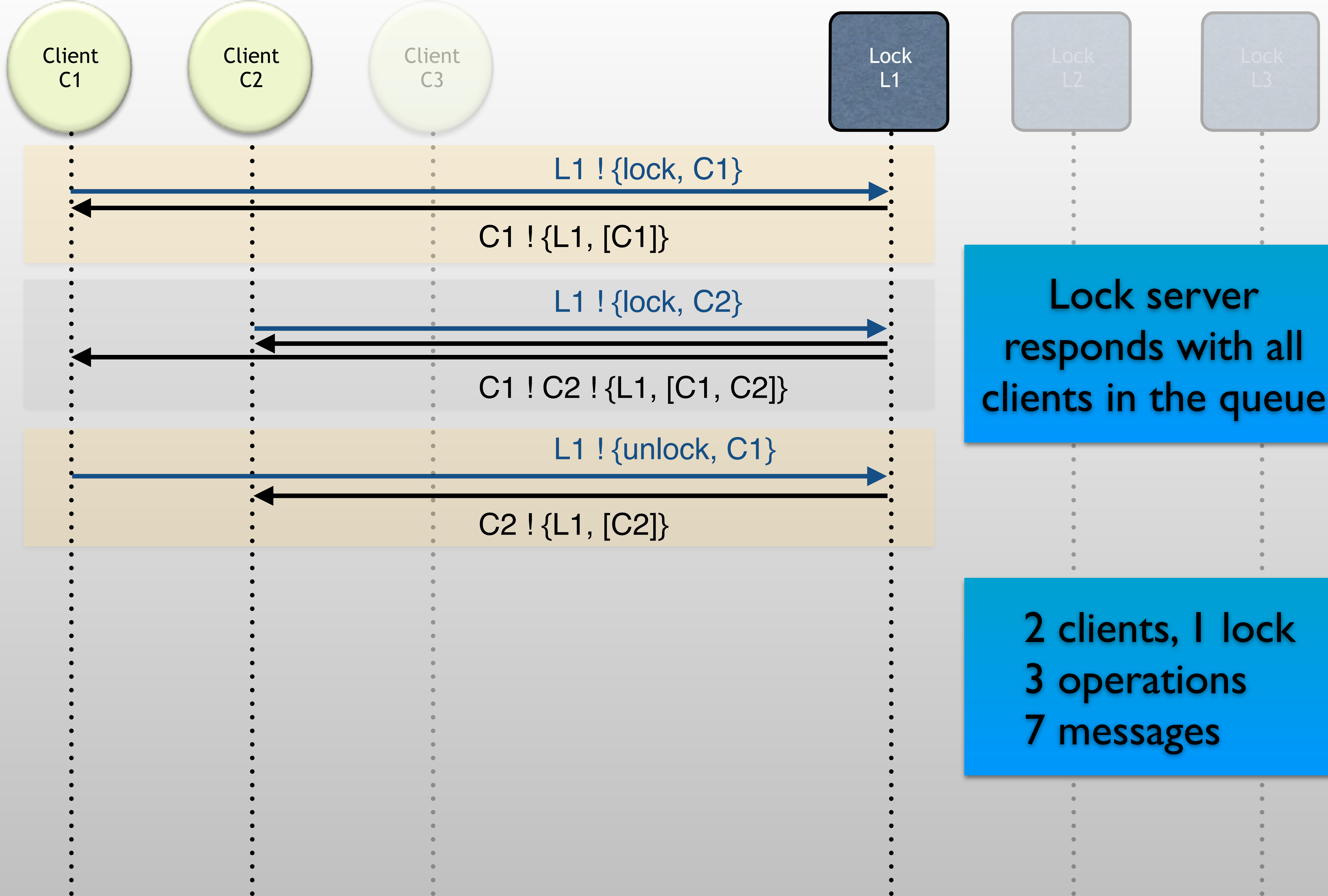


Erlang-style locking

- The lock itself is a process
- Transaction context is a process
- Asynchronous message passing
- Distributed dependency analysis

FEUERLABS

Example: simple lock

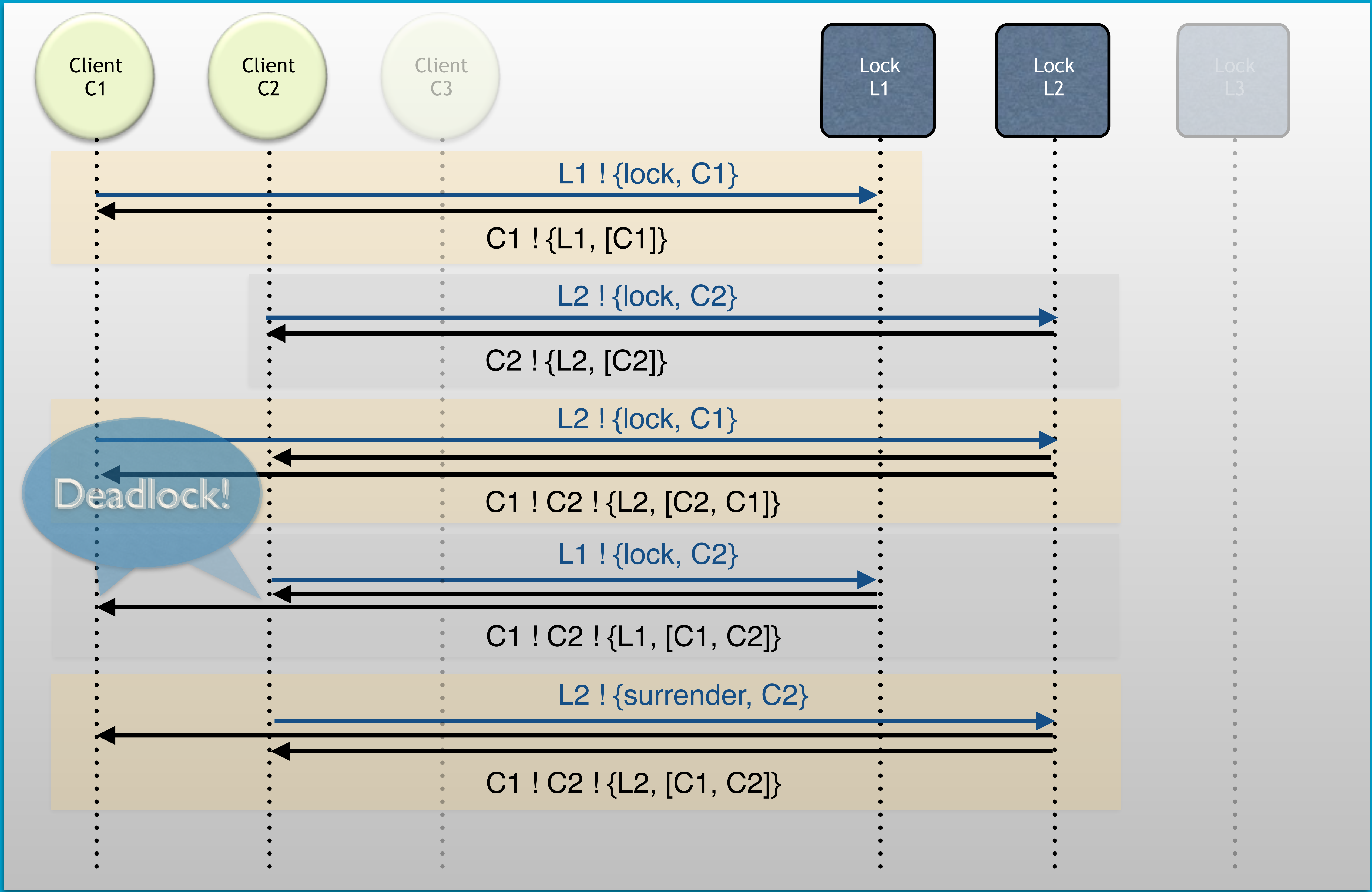


Lock server responds with all clients in the queue

2 clients, 1 lock
3 operations
7 messages

Simple deadlock

FEUERLABS



Complexity

FEUERLABS

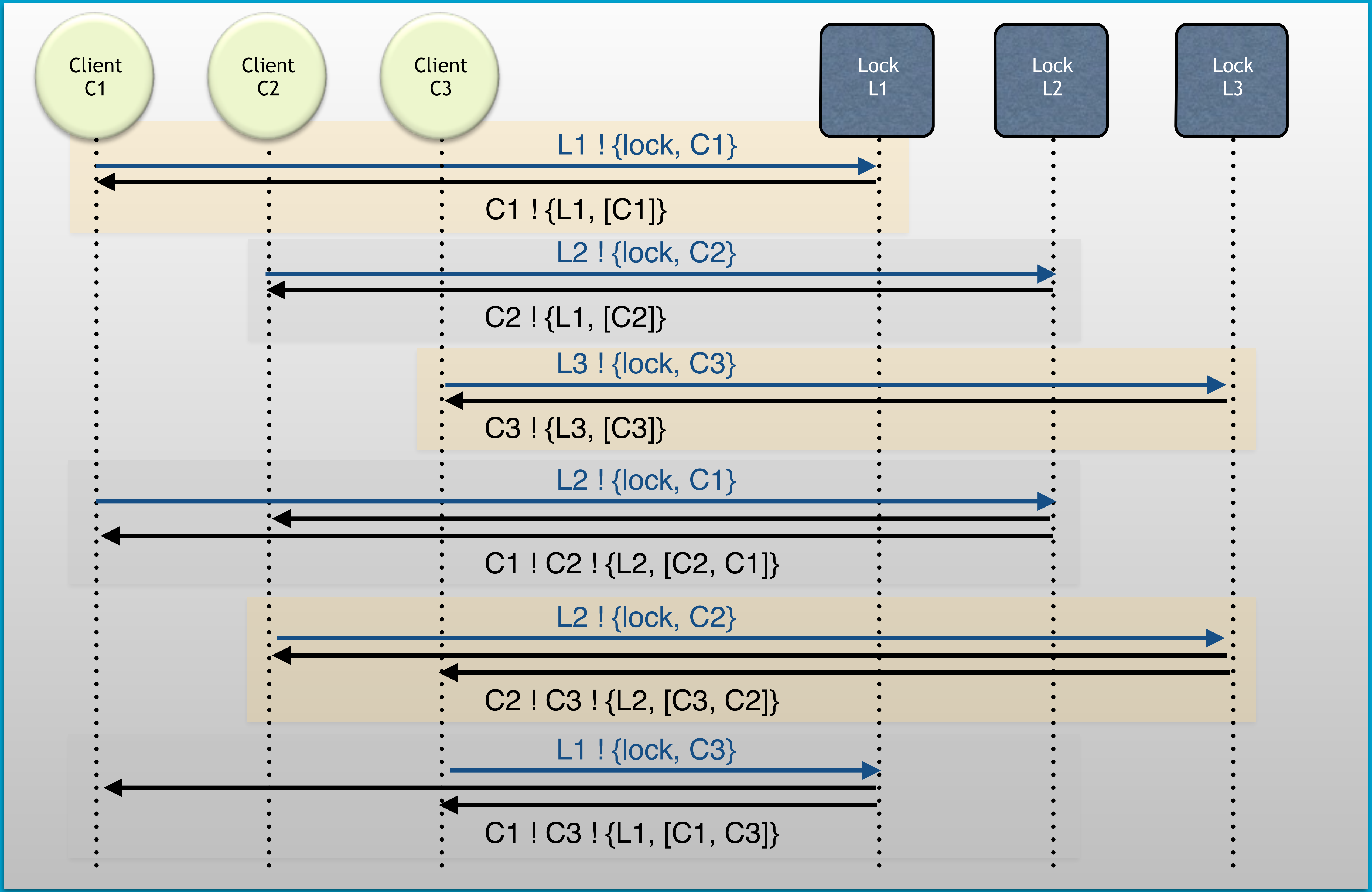
- 2 clients
- 2 locks
- 4 operations [1]
- 2 dependencies [2]
- 1 deadlock resolution [3]

- (4*2 + 2*1 + 1*(2+1) = 13 messages)

  [1] [2] [3]

Indirect deadlock (1)

FEUERLABS

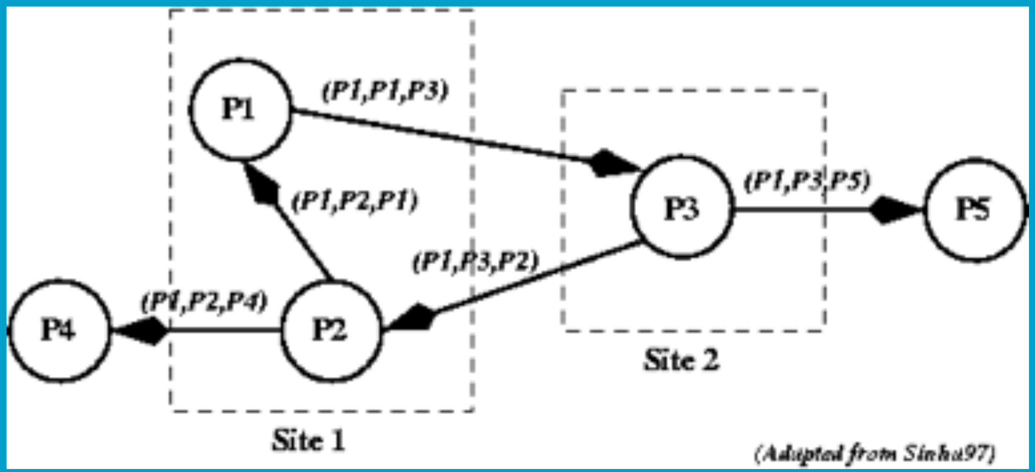


Fill-in-the-blanks

- Share lock dependency D with
  - Greater client C, which holds a lock
  - If C is not involved in D

Chandy-Misra-Hass Detection Algorithm (1983)

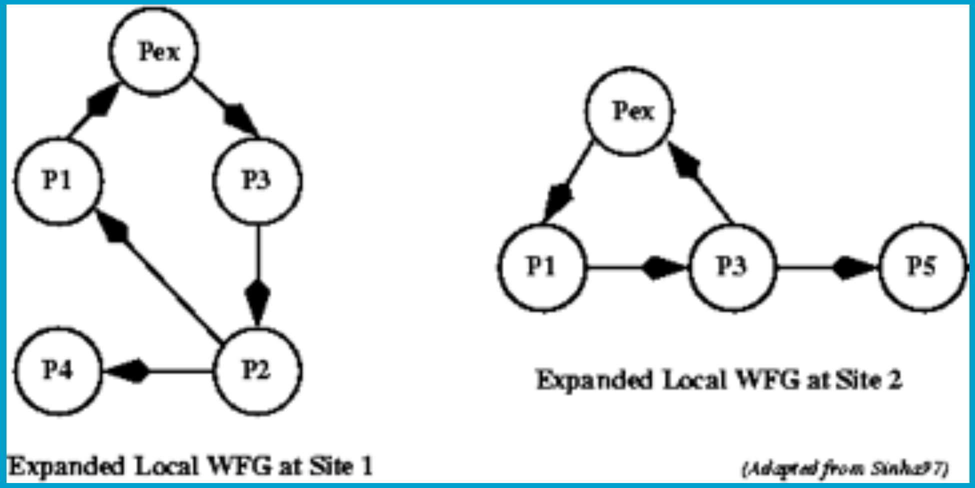
- Each waiting process sends probe to each lock holder it waits for
- Each probe receiver passes it on to lock holders it waits for



http://www.cs.colostate.edu/~cs551/CourseNotes/Deadlock/DDCMHAlg.html

Silberschatz-Galvin Detection Algorithm (1993)

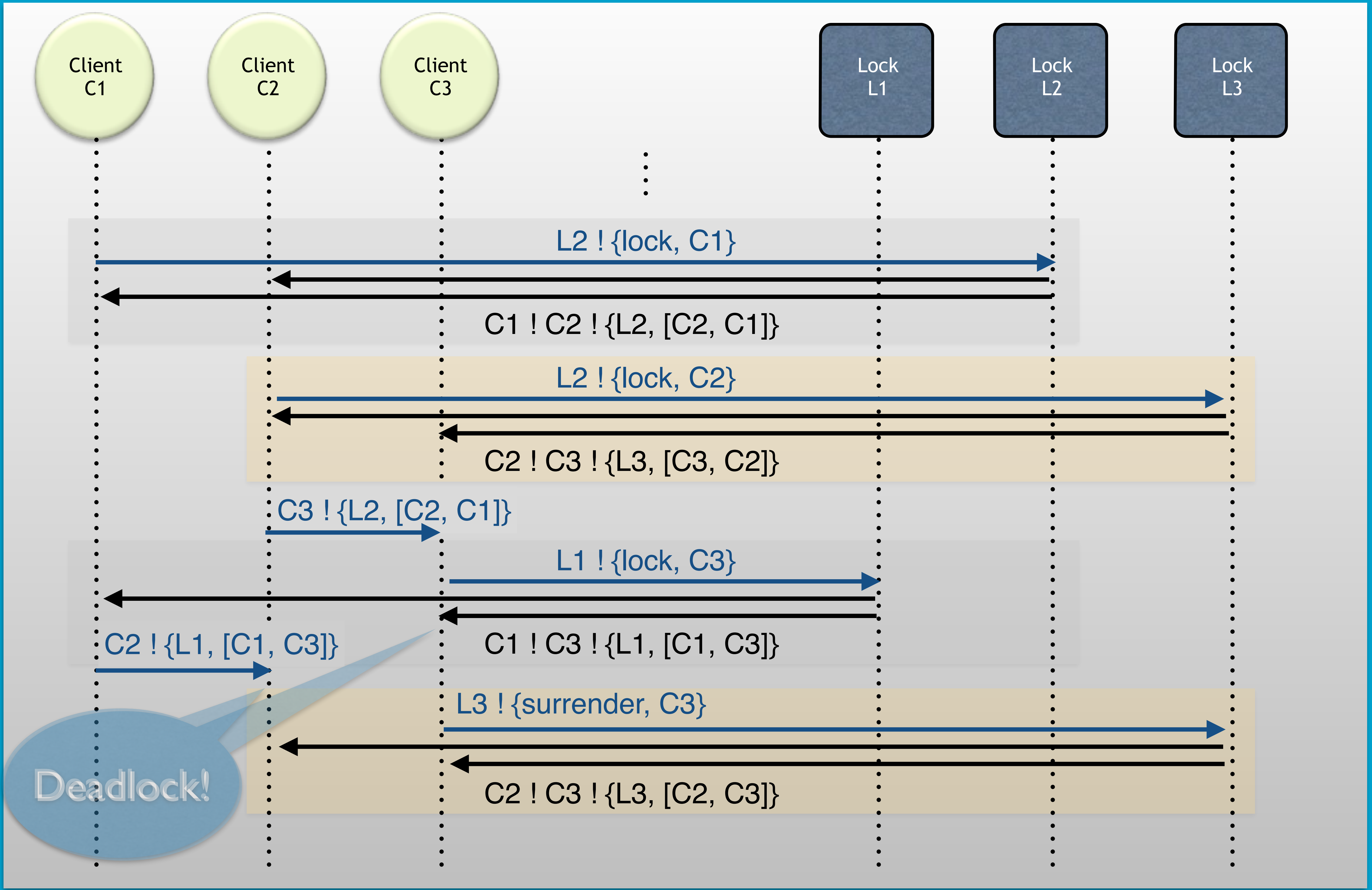
- Mark external dependencies in WFG
- Send complementary info to other site



http://www.cs.colostate.edu/~cs551/CourseNotes/Deadlock/DDSilGal94.html

Indirect deadlock (2)

FEUERLABS



Complexity

FEUERLABS

- 3 clients
- 3 locks
- 6 operations [1]
- 3 direct dependencies [2]
- 2 indirect dependencies [3]
- 1 deadlock resolution [4]

- (6*2 + 3*1 + 2*1 + 1*(2+1) = 20 messages)
  [1] [2] [3] [4]

Always surrender?

FEUERLABS

- Problematic if client has already acted on the lock
- {abort_on_deadlock, true}, will
    - Surrender lock iff the client has not yet been informed of the lock
    - Otherwise, abort

Multi-node locks

- Each {Obj,Node} pair is a separate lock
- Transaction agent keeps track of how many nodes are needed for request to be served
  - All requested
  - A majority of all requested
  - All/majority nodes that are alive

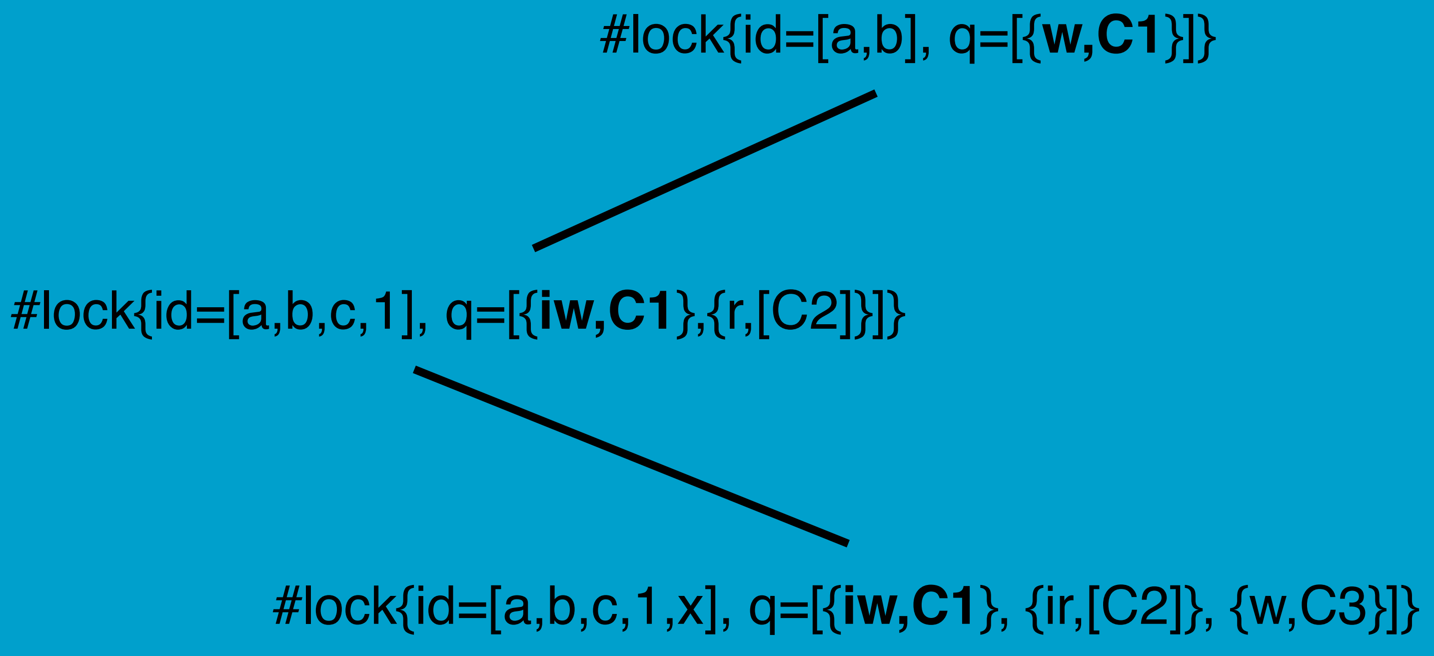
Read/write locks

- Write locks = exclusive
- Read locks = shared
- The only key aspect for dependency analysis is who waits for whom:
  - Write locks wait for read and write locks
  - Read locks wait for write, but not read, locks
- Queue: `#lock{queue = [{r,[C1,C2]}, {w,C3}, {r,[C4]}]}`

FEUERLABS

Hierarchical locks

- Lock ID is a list: [kvdb, my_db, my_tab, obj1]
- Key enabler: implicit locks
- Dependency analysis sees no difference



Scalability: Large transactions

FEUERLABS

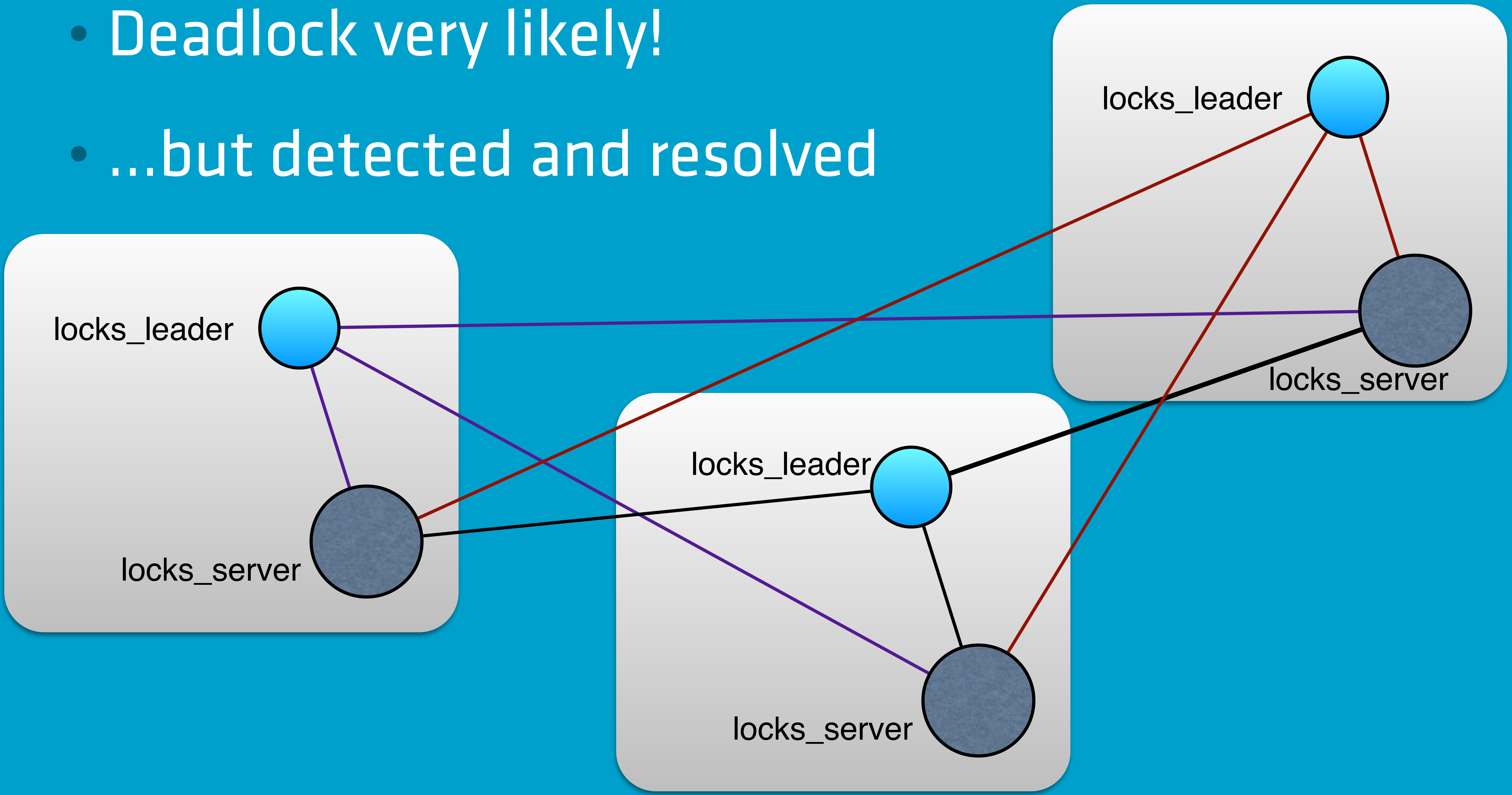
- Test: claim N independent locks within one transaction (measure: latency)
- Roughly constant cost per lock request, even with 1000s of locks
- Starting cost:
  - ~ 100 us (locks:begin_transaction/0)
  - ~ 20 us + ~50 us (locks: spawn_agent/1)

```
Eshell V5.9.2  (abort with ^G)
1> bench:simple_locks(1).
[{1,174.2}]
2> bench:simple_locks(1000,1010).
[{1000,229.7},
 {1001,244.6},
 {1002,239.9},
 {1003,212.6},
 {1004,183.6},
 …]
3> bench:simple_locks(3000,3010).
[{3000,255.7},
 {3001,266.5},
 {3002,251.5},
 {3003,206.5},
 {3004,183.0},
 …]
4> bench:simple_locks(5000,5010).
[{5000,283.1},
 {5001,282.3},
 {5002,260.5},
 {5003,232.0},
 {5004,192.9},
 …]
```

Leader Election

FEUERLABS

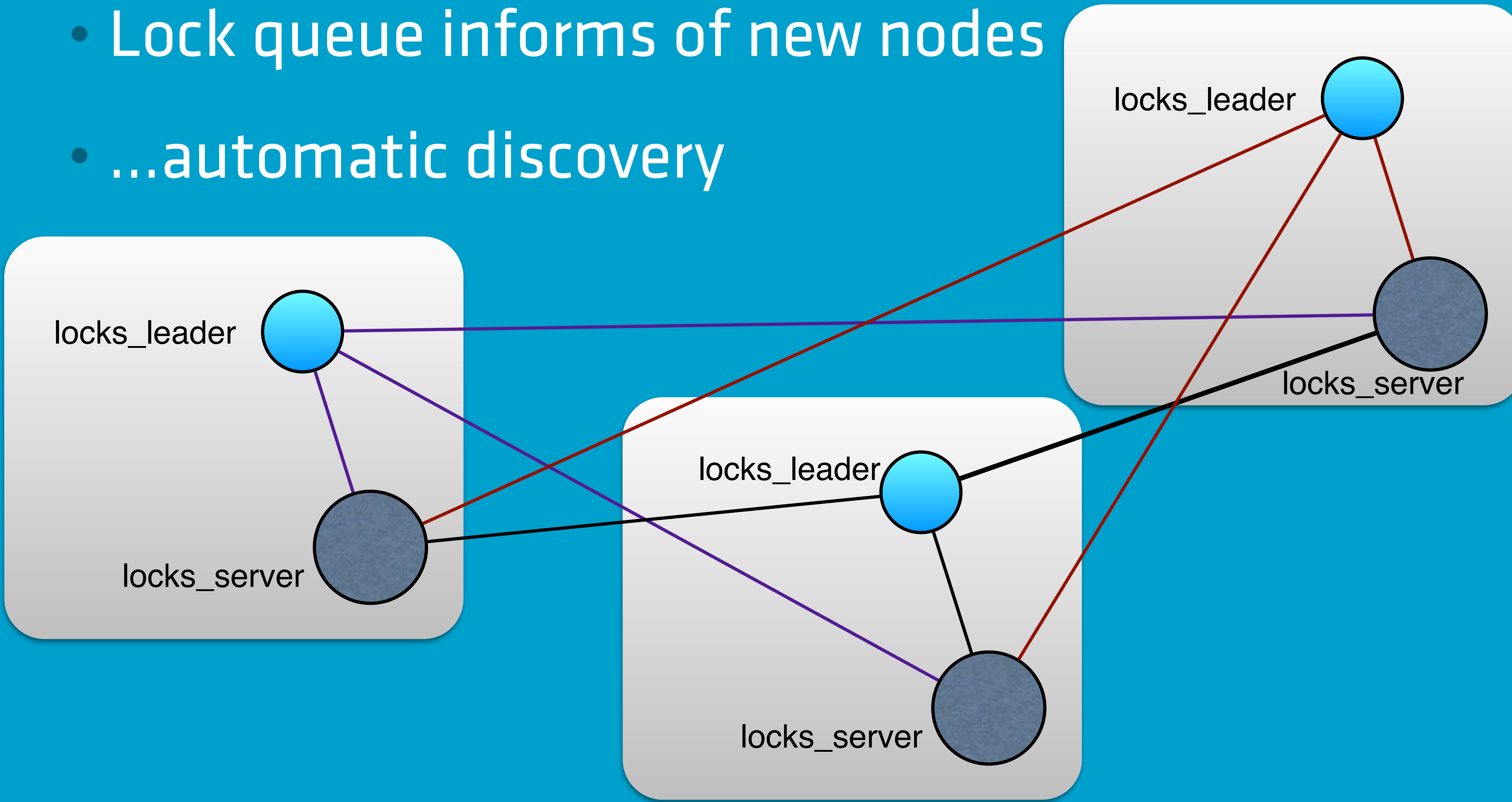
- All candidate try to lock Resource on all nodes
- Deadlock very likely!
- ...but detected and resolved



Leader Election (2)

FEUERLABS

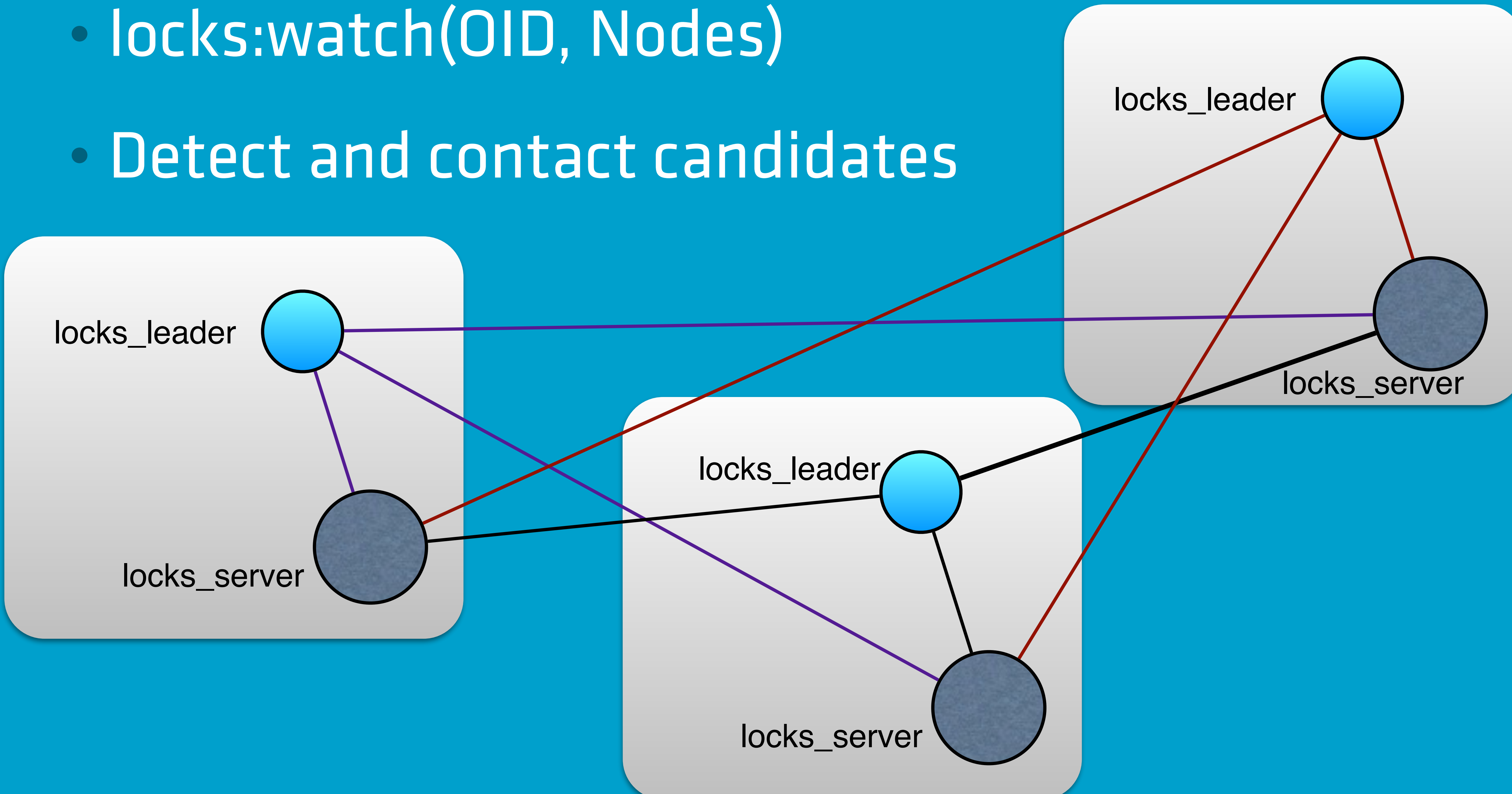
- Asynchronous lock requests
- Lock queue informs of new nodes
- ...automatic discovery



Leader Election (3)

FEUERLABS

- Workers must not attempt to lock!
- locks:watch(OID, Nodes)
- Detect and contact candidates



A better gen_leader?

FEUERLABS

- Handles dynamic (Erlang-style) networks
- Can have multiple candidates on the same node
- Candidates don’t have to be registered
- Netsplit handling with conflict resolution
  - Extended API with e.g. ask_candidates/2 (allows for state merging upon election

Status

FEUERLABS

- Currently integrating into the kvdb DBMS
  - Feuerlabs Exosense test suites pass using 'locks'
- The gproc 'uw-locks_leader' branch uses 'locks' for global properties
- Unit test exercises various weird locking scenarios

- https://github.com/uwiger/locks