SC250
Computer Networking I

Finite State Machines

Prof. Matthias Grossglauser

School of Computer and Communication Sciences
EPFL

http://lcawww.epfl.ch

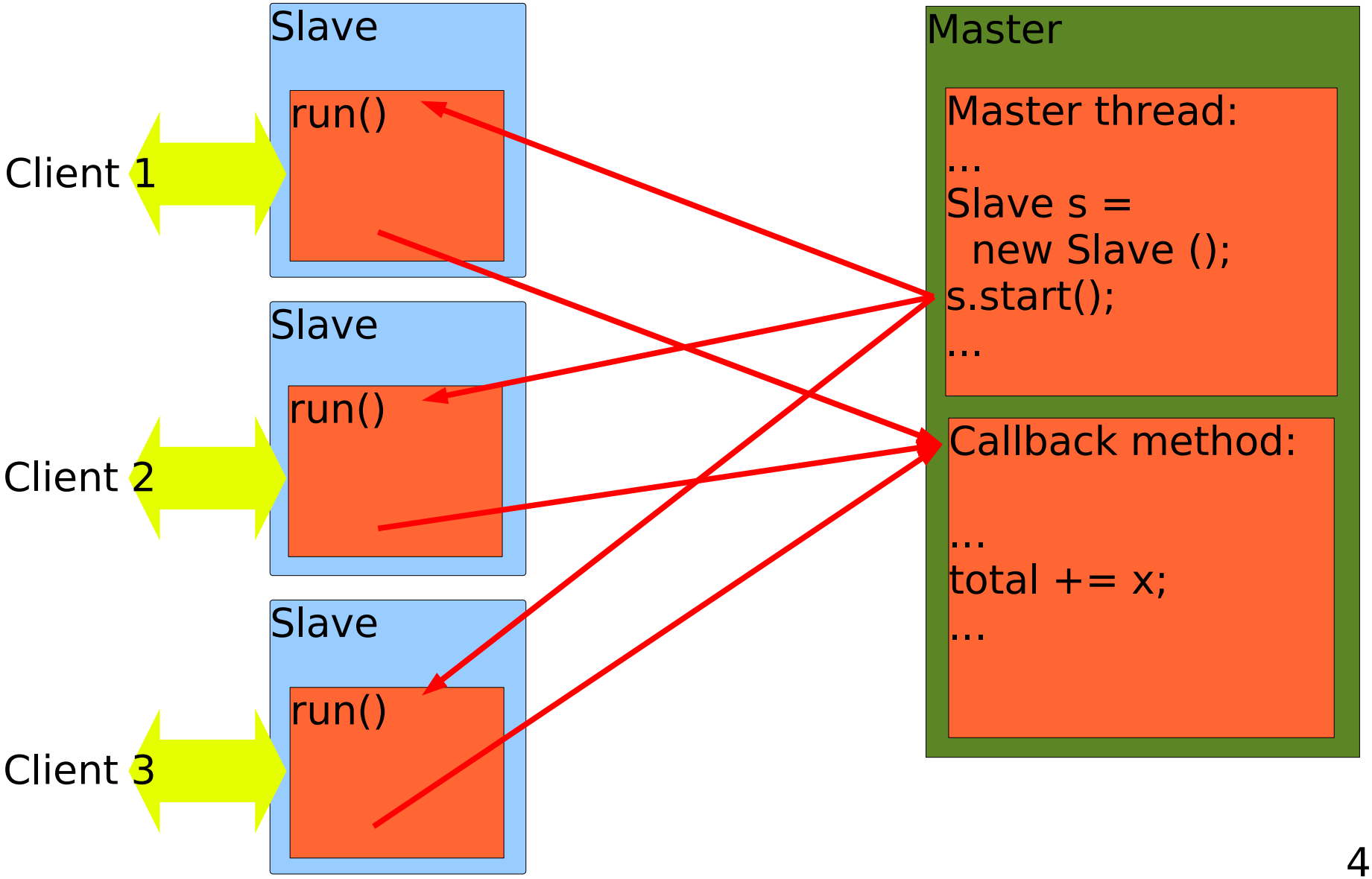
Today's Objectives

- More on java multithreading
  - Callback
- Finite state machines (FSMs)
  - Language to specify and analyze protocols
  - How to use FSMs in protocol development
  - Aggregate state machine
  - Checking properties

Java Threads: Notion of Callback Method

- Multithreading example last week:
  - Main threads listens on welcomeSocket
  - Separate thread launched to serve each client
  - Threads independent, simply go away after work done
- More complex scenarios:
  - Dependences between threads
  - Need for coordination & information exchange
  - Concurrent programming, handling asynchrony
- Common case: callback from thread to invoker
  - Networking, GUIs,...
  - Implements publish-subscribe

Callback: Main Idea



Server with Callback Method

Callback method

Synchronized block, only one thread can execute (mutex)

```
public class HiscoreServer {

    int hiscore = -1;

    public void receiveScore (int hiscore) {

        synchronized (this) {
            if (hiscore > this.hiscore) {
                this.hiscore = hiscore;
                System.out.println ("new highscore: " + hiscore);
            }
        }
    }

...
```

Server with Callback Method (cont.)

Starting thread, passing self-ref for callback

```
...

    private void listen (int socket) throws Exception {
        ServerSocket serverSocket = new ServerSocket(socket);

        while (true) {
            HiscoreServerThread thread = new
                HiscoreServerThread(serverSocket.accept(), this);
            thread.start();
        }
    }

    public static void main(String[] args) throws Exception {

        HiscoreServer hs = new HiscoreServer ();
        hs.listen (4444);
    }
}
```

Server Thread

Thread receives reference to callback method

```
public class HiscoreServerThread extends Thread {

    private int hiscore;
    private Socket socket = null;
    private HiscoreServer callback;

    public HiscoreServerThread(Socket socket, HiscoreServer callback) {
        super("HiscoreServerThread");
        this.socket = socket;
        this.callback = callback;
    }
```

Server Thread (cont.)

```
...
    public void run() {
        try {
            DataInputStream inFromClient =
                new DataInputStream(
                        new BufferedInputStream(socket.getInputStream()));
            hiscore = inFromClient.readInt();
            callback.receiveScore(hiscore);
            socket.close();
        } catch (IOException ex) {
            System.err.println (ex);
        }
    }
}
```

Callback to server object

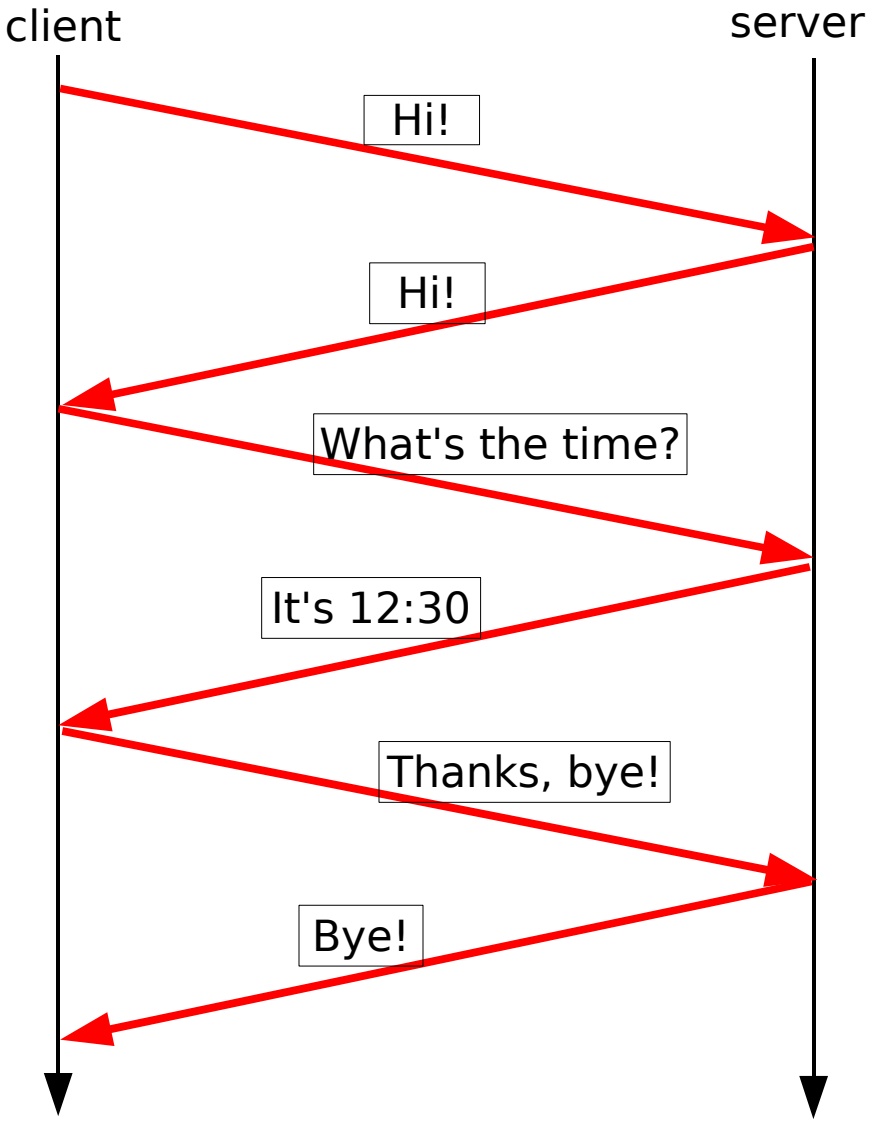
Callback for Publish-Subscribe

- Callback implements publish-subscribe
  - Some threads (subscribers) are interested in signals from other threads (publishers)
  - Here: HiscoreServer=subscriber, HiscoreServerThread=publisher
- Can be generalized
  - Multiple subscribers for a publisher
  - One technique: publisher keeps list of subscribers
  - Java:
    - Each subscriber has to implement a callback interface
    - Subscriber offers “subscribe” method
    - Publisher calls callback of every registered subscriber
    - Similar technique used for GUI

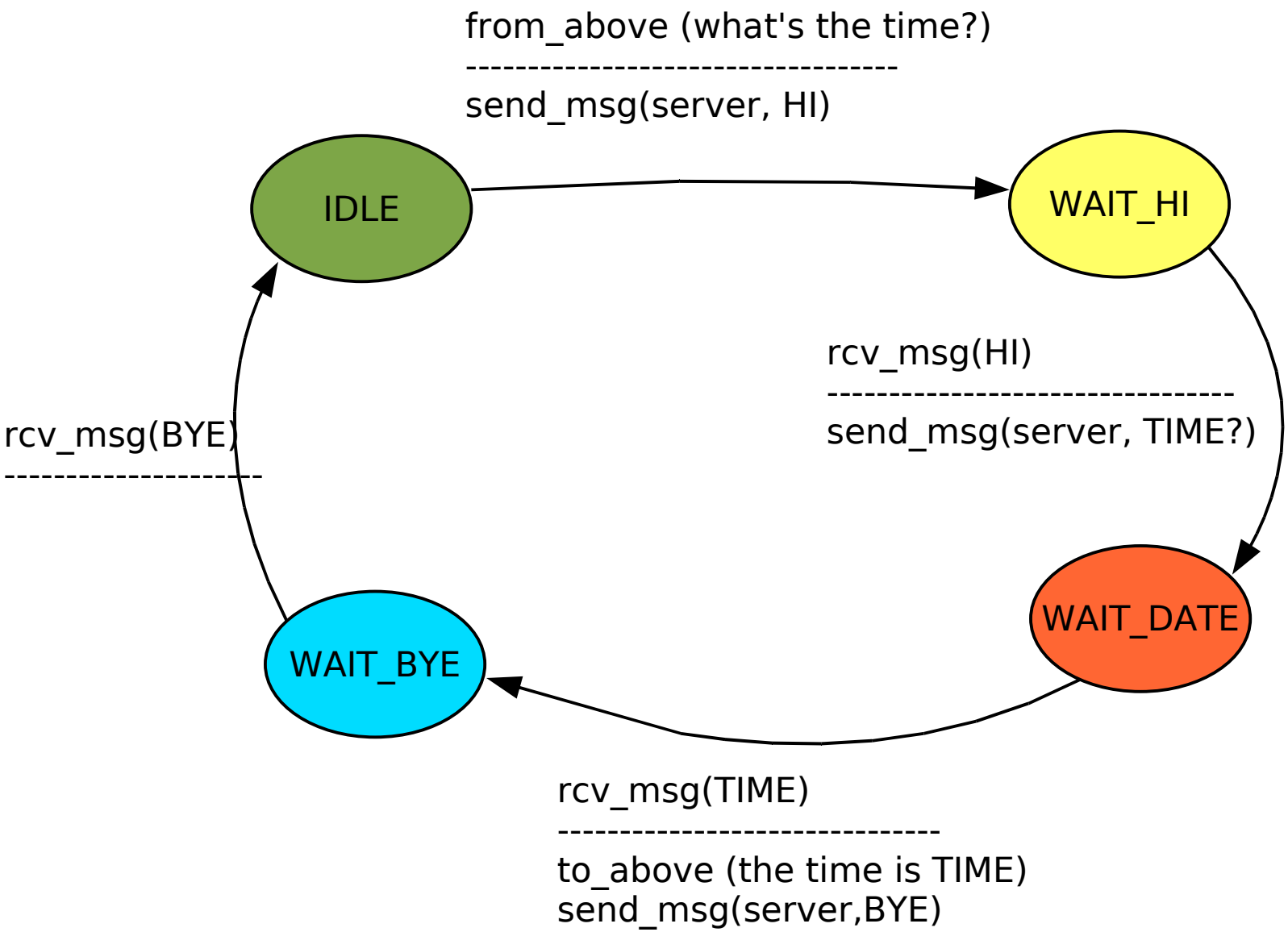
FSM: Get the Time from a Stranger

- Specification:
  - Implement a protocol that obtains the time from a stranger
  - Constraints: you have to be polite: a stranger will not respond unless you precede any question by a polite exchange of “hi's”, and thank the stranger in the end
- Developing a protocol:
  - Identify a typical exchange, draw on a “timeline diagram”
  - Identify:
    - Types of messages
    - States for both sender and receiver
    - Transitions between these states
  - Draw full FSM
  - Convince yourself that the protocol is ok

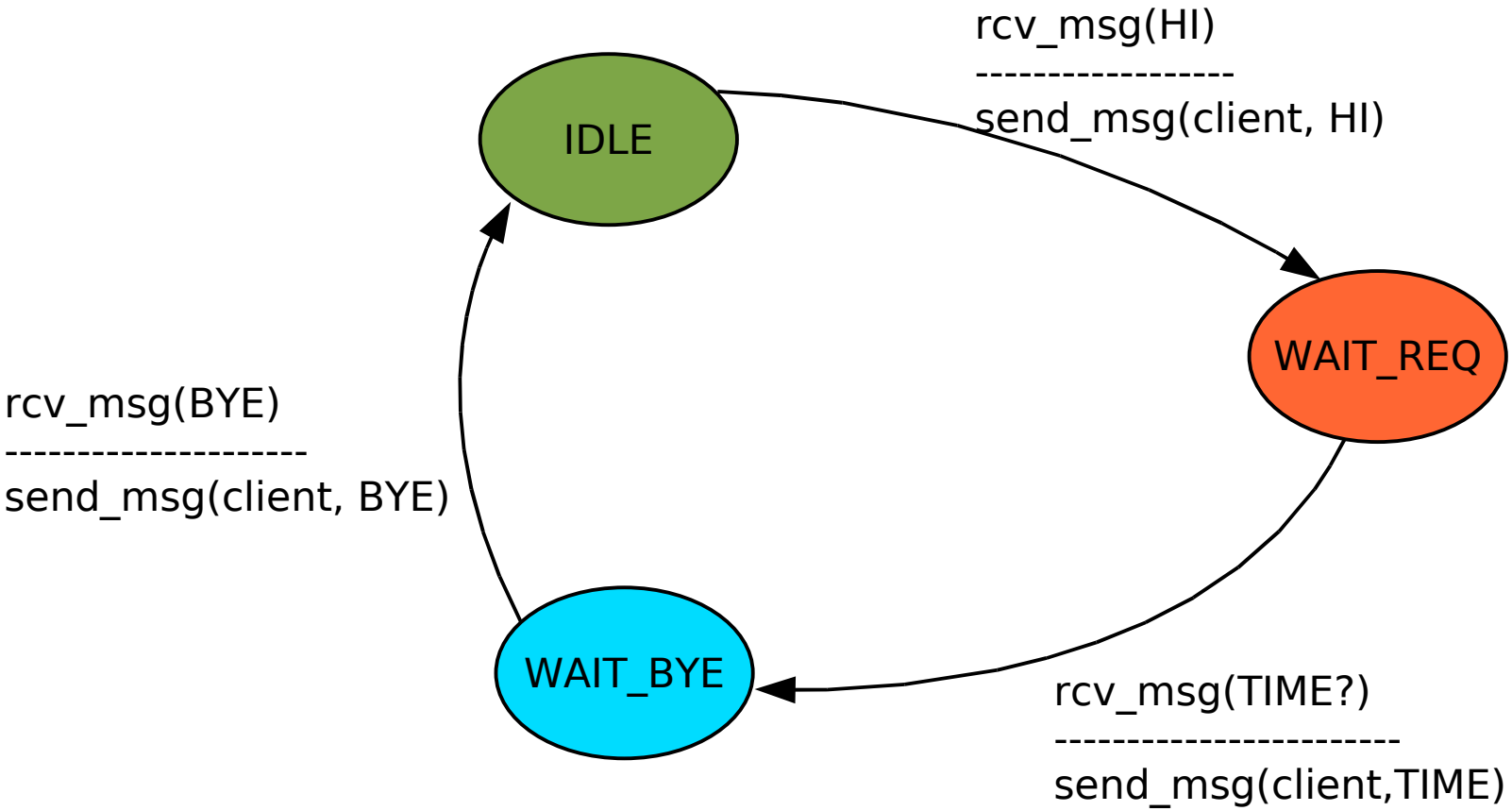
Example on Timeline



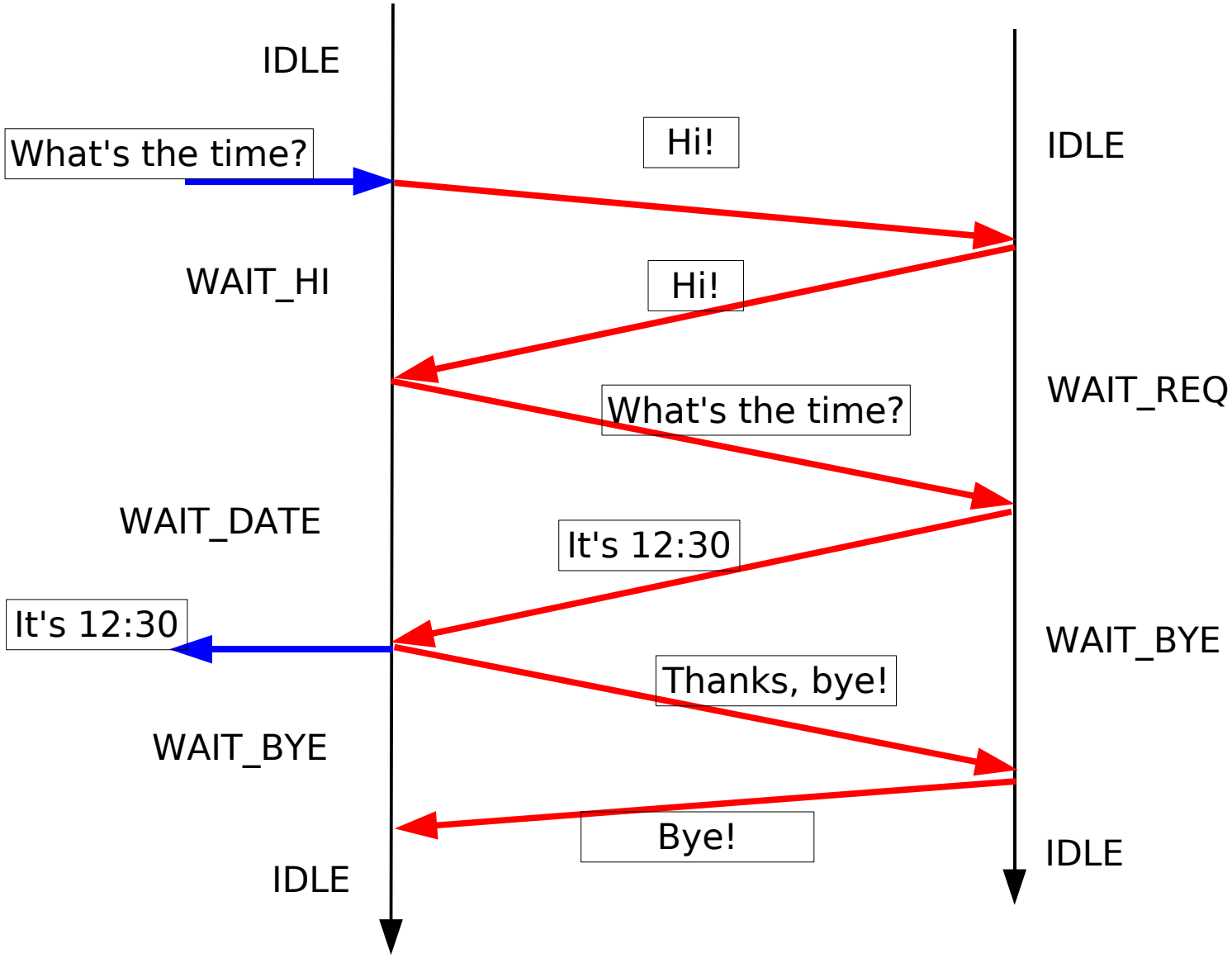
Protocol 1: Client FSM for get_time()



Protocol 1: Server FSM for get_time()

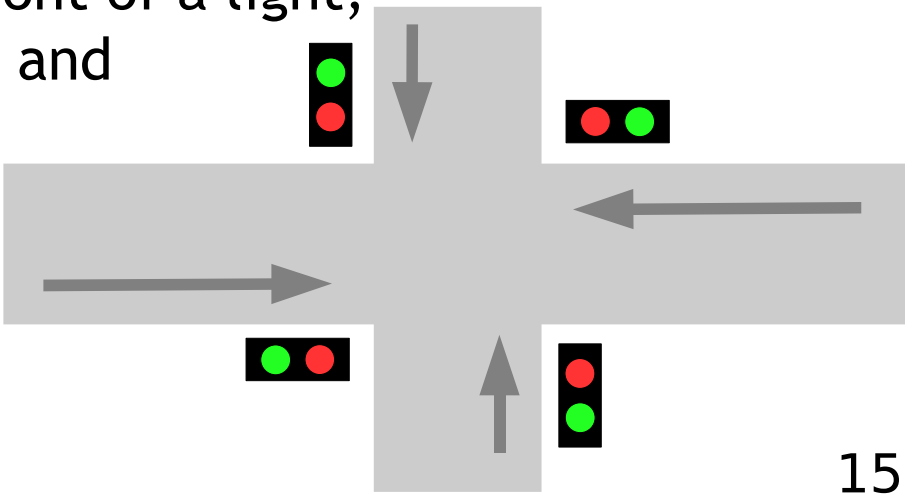


Example on Timeline

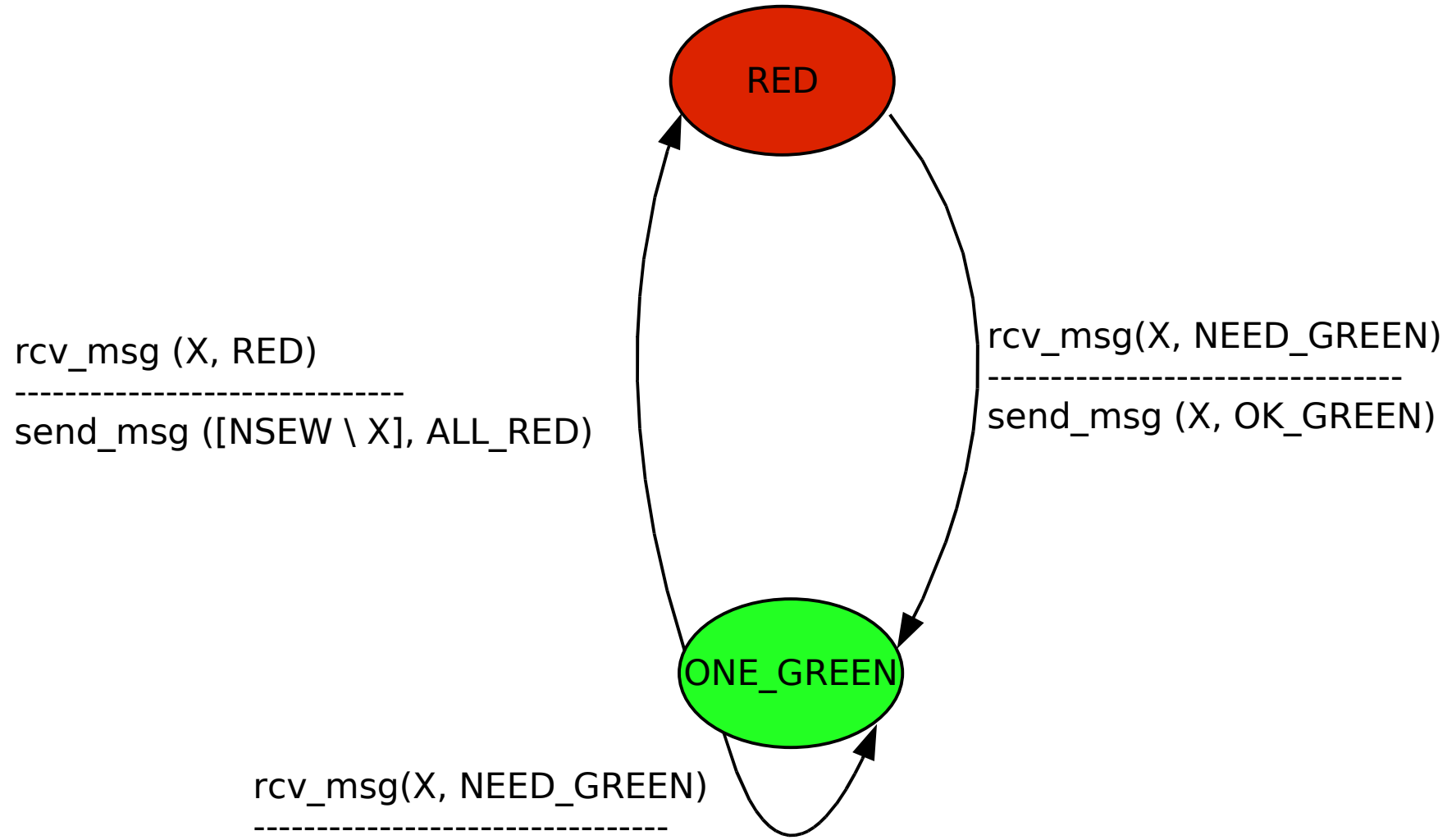


Example 2: Distributed Traffic Light

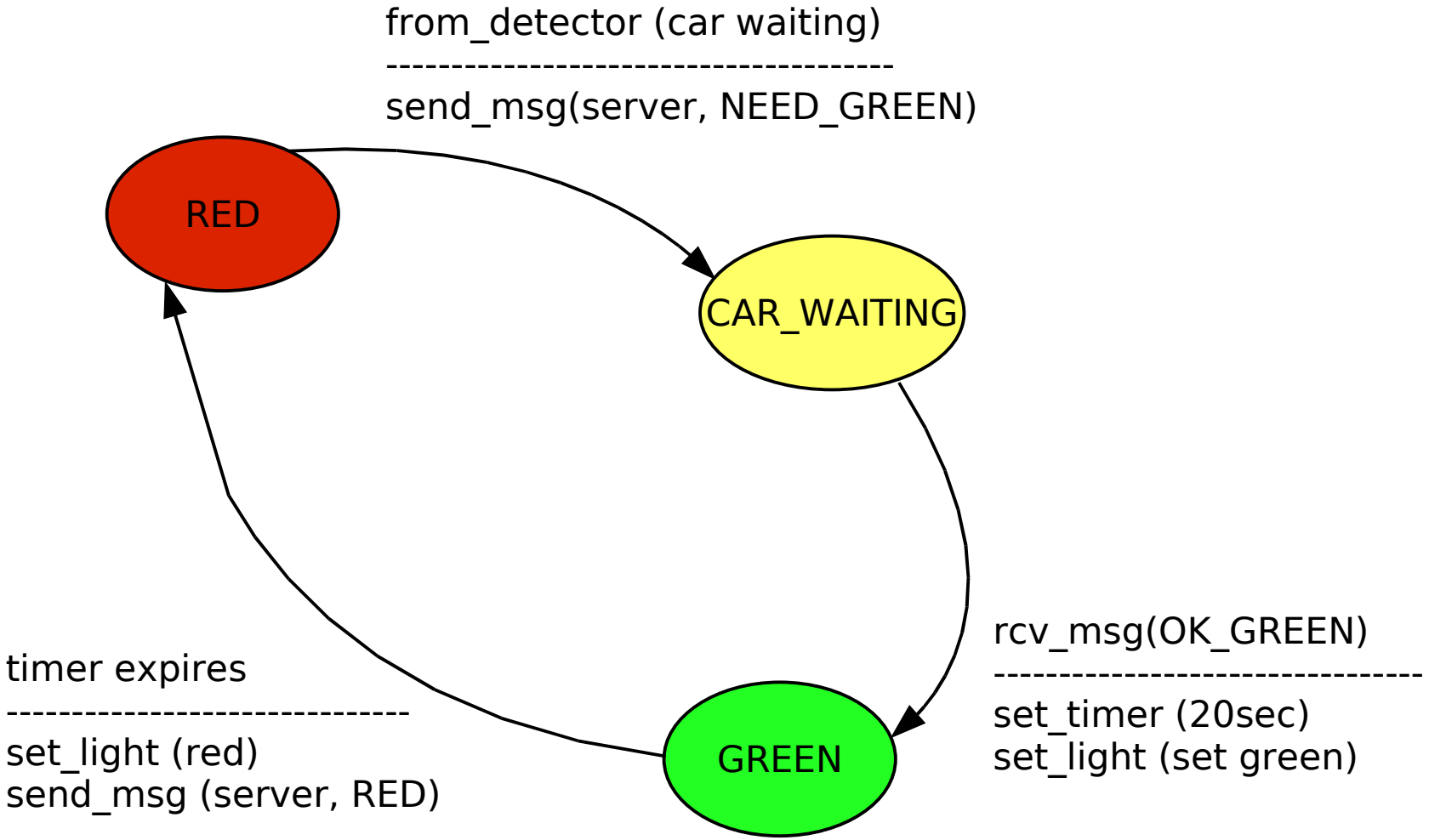
- Specification:
  - We need to control the traffic lights at a four-way intersection
  - Each direction (NSEW) is equipped with a traffic light and a detector for cars waiting; this is controlled by a computer (Client_NSEW)
  - The four clients are connected to a server
  - Your task: Write a protocol to ensure that
    (i) when a car is waiting in front of a light, it will eventually turn green, and
    (ii) at any given time, only one light is green



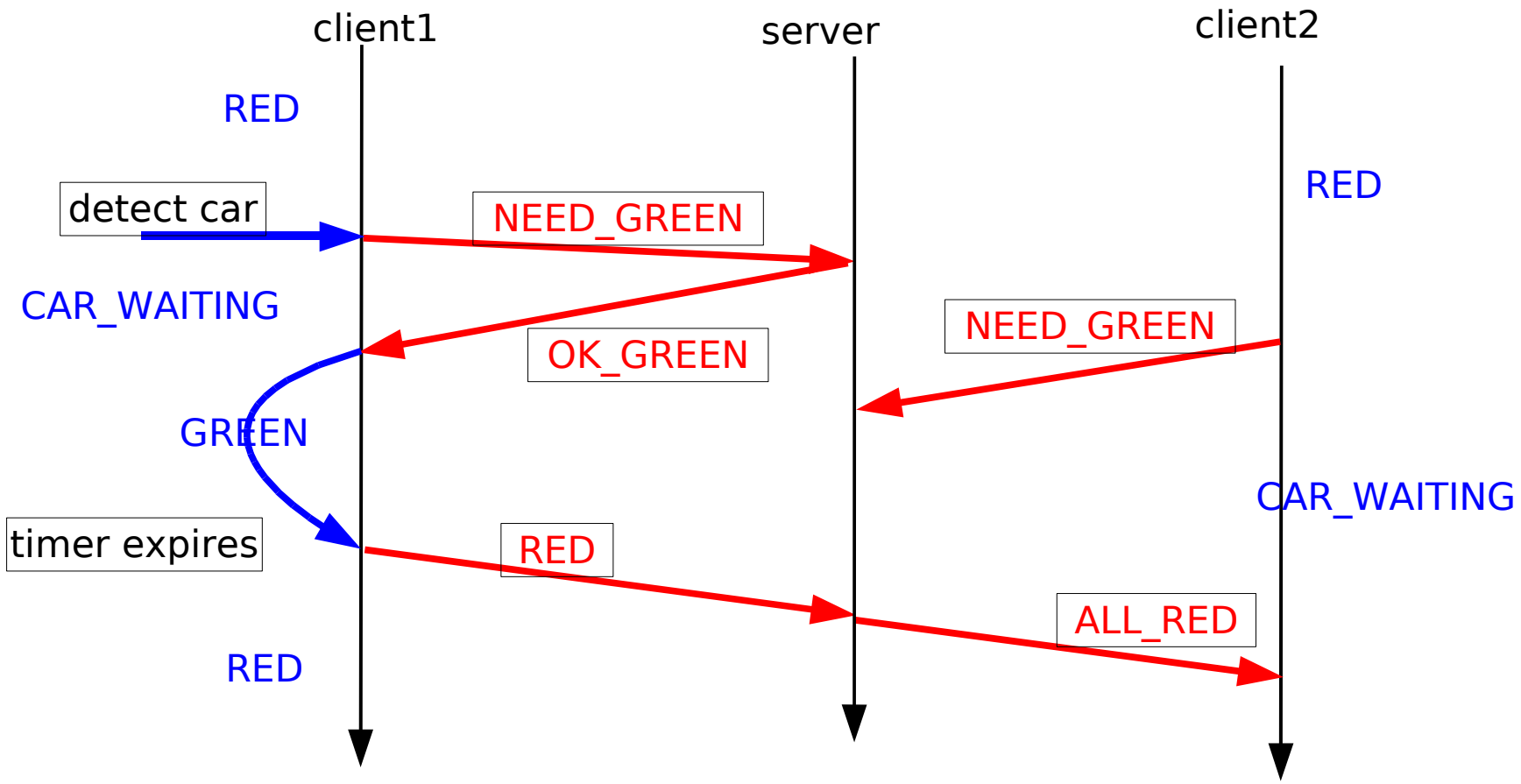
Protocol 2: Server FSM for Intersection



Protocol 2: Client FSM for Intersection

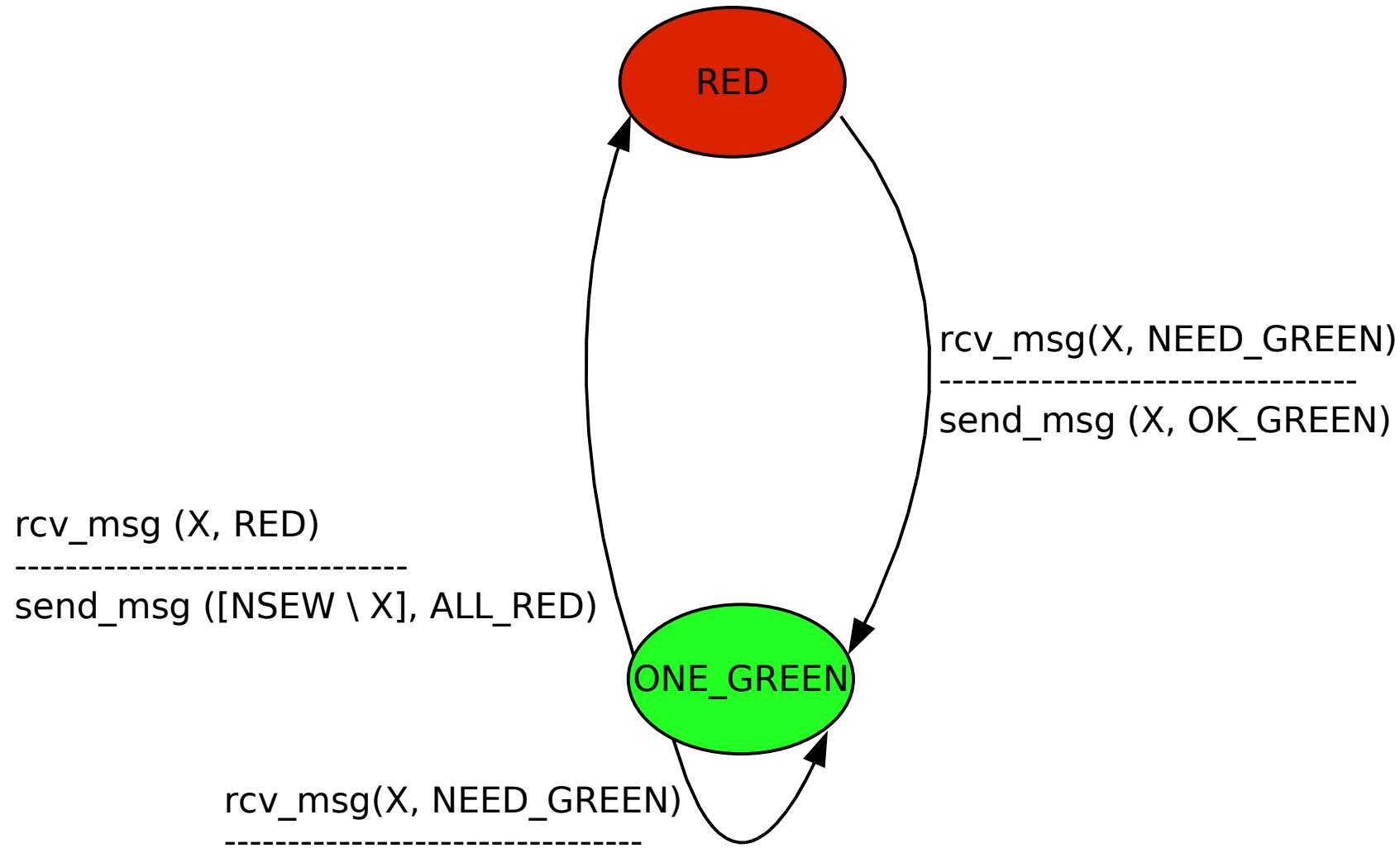


Problem: Blocked Client

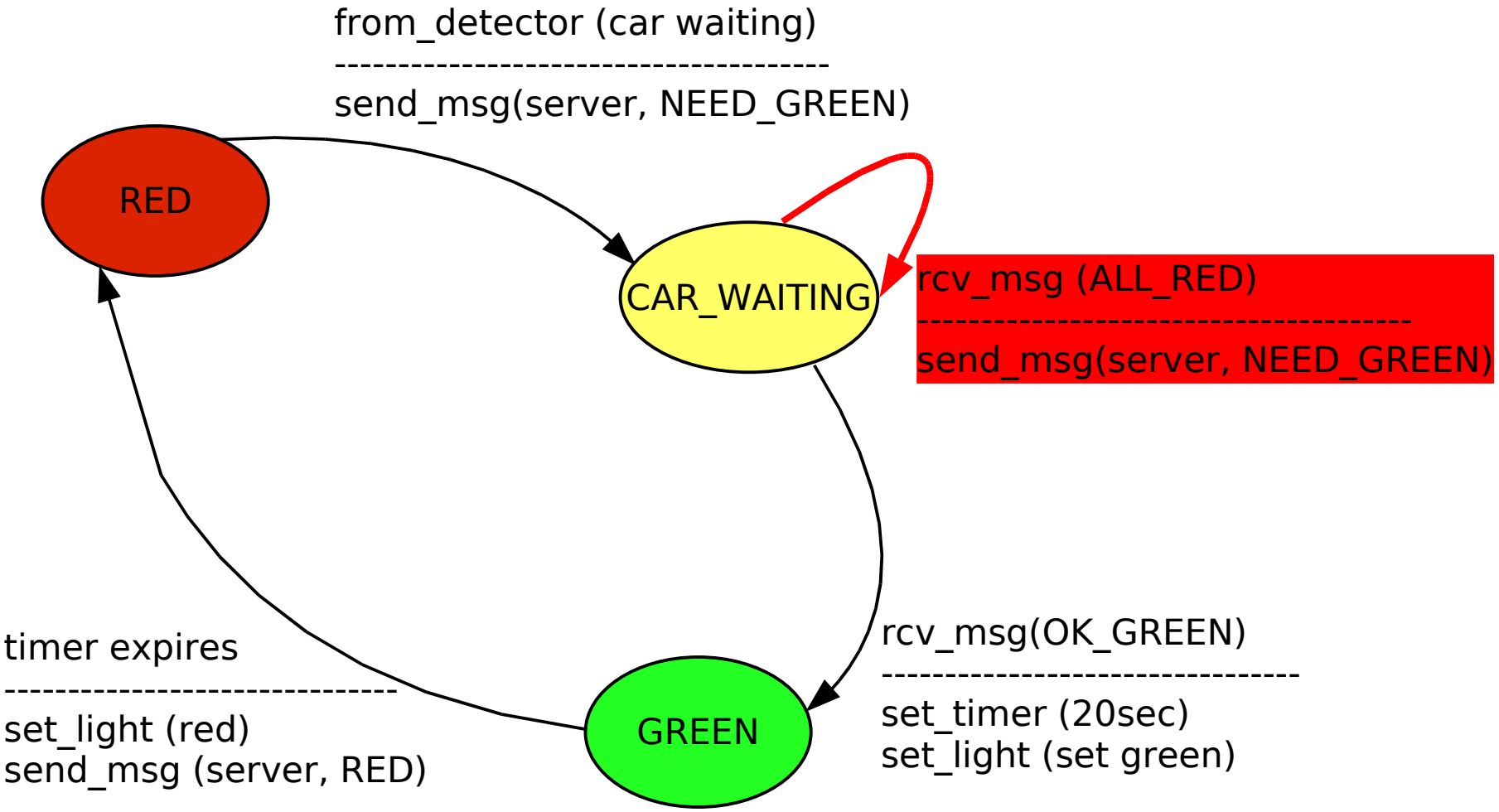


- Problem: client 2 remains stuck in state CAR_WAITING forever
- Solution?

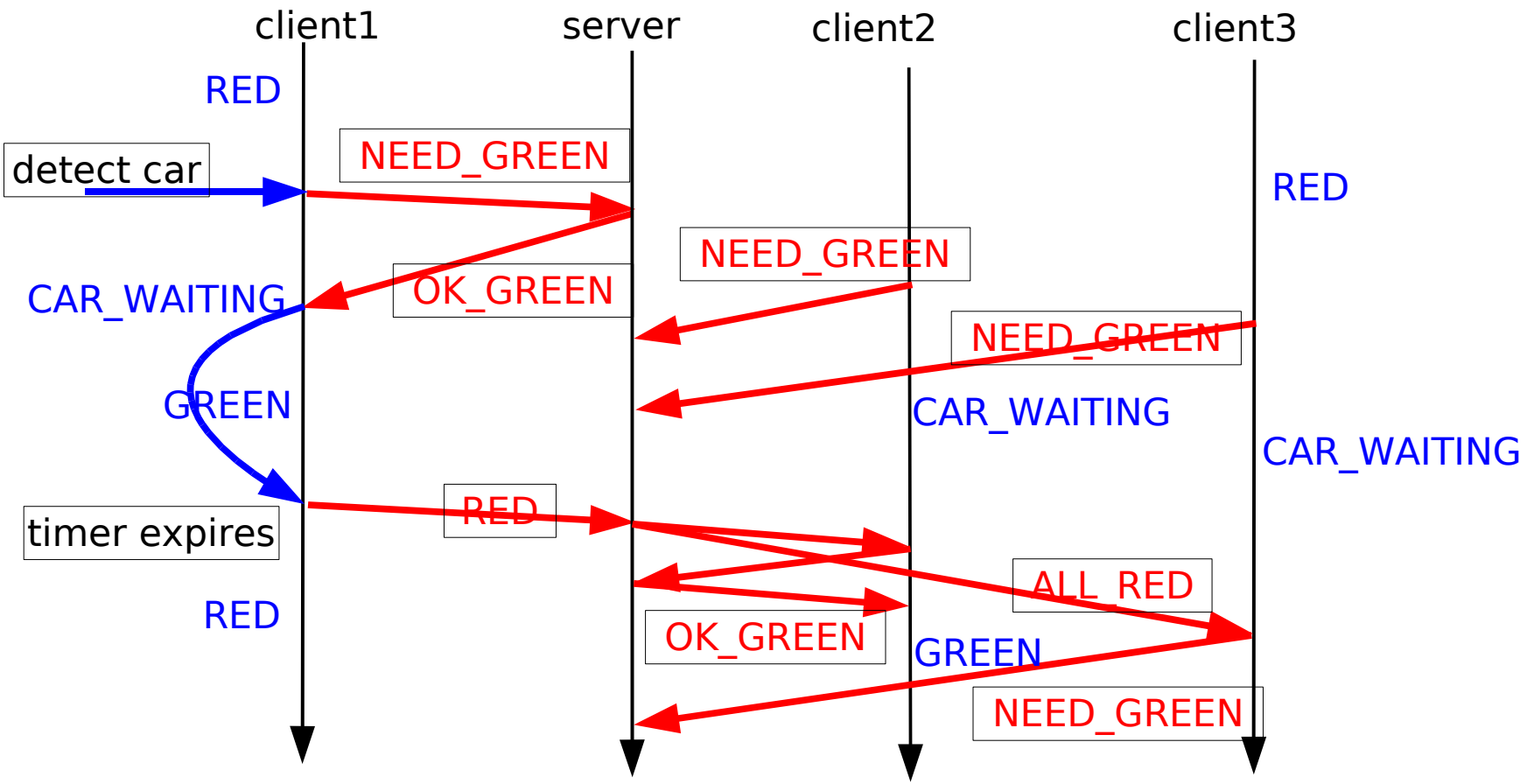
Protocol 3: Server FSM, no Blocking



Protocol 3: Client FSM, no Blocking

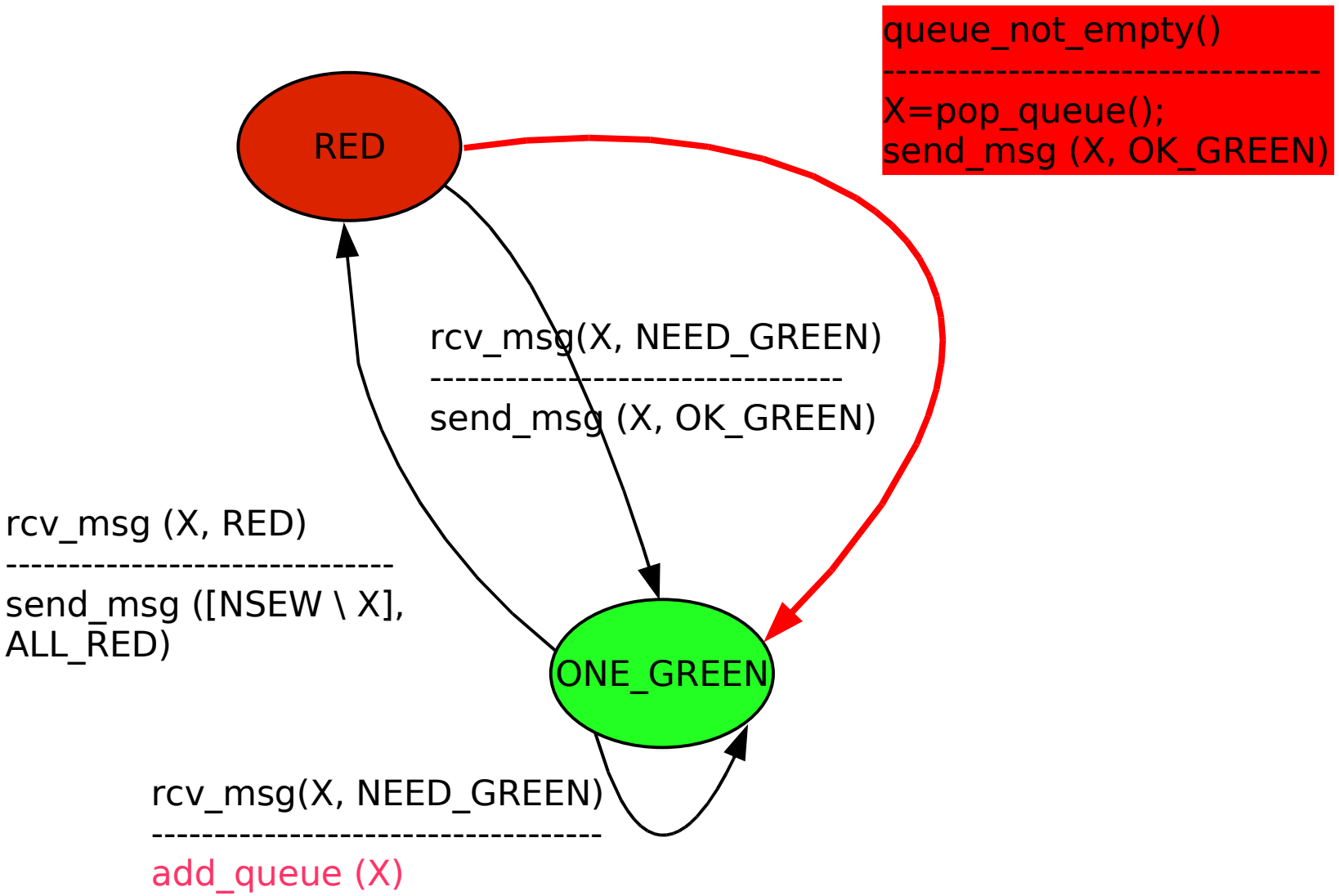


Problem: Starvation

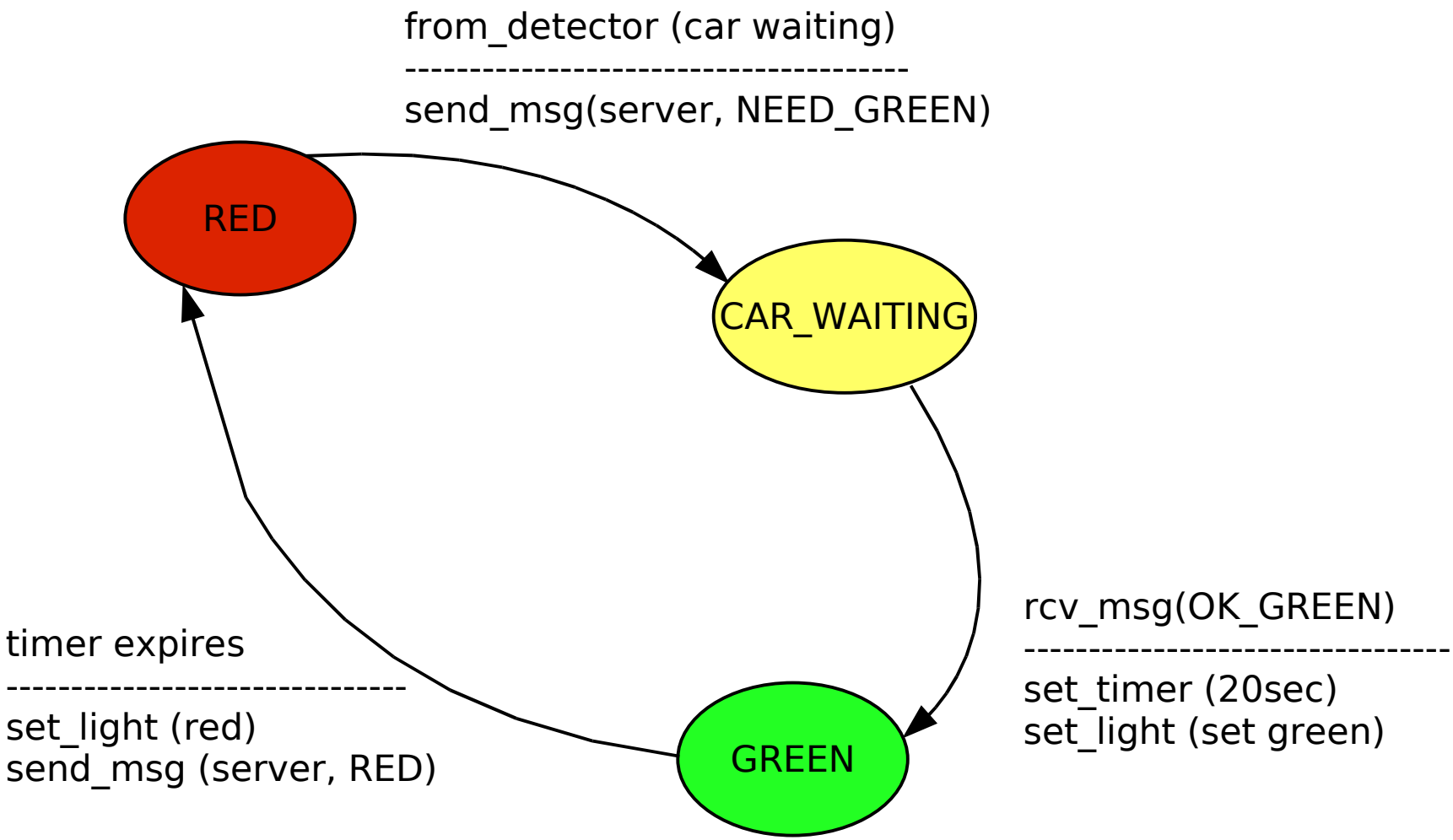


- Problem: if client2 and 3 are such that delay(cl2->server) < delay(cl3->server) always, then client2 can “starve” client 3
- Solution?

Protocol 4: Server FSM, no Starvation



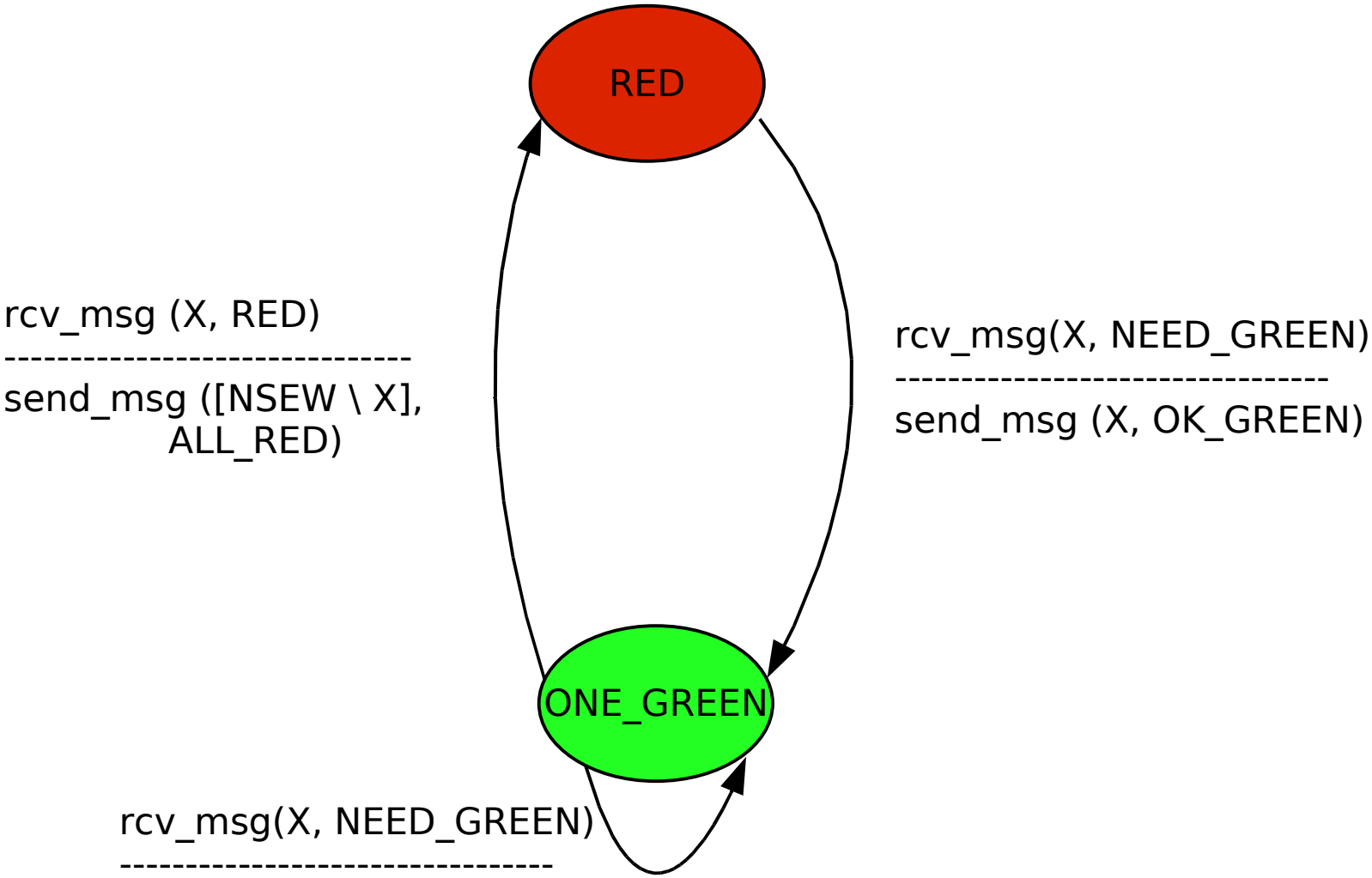
Protocol 4: Client FSM, no Starvation



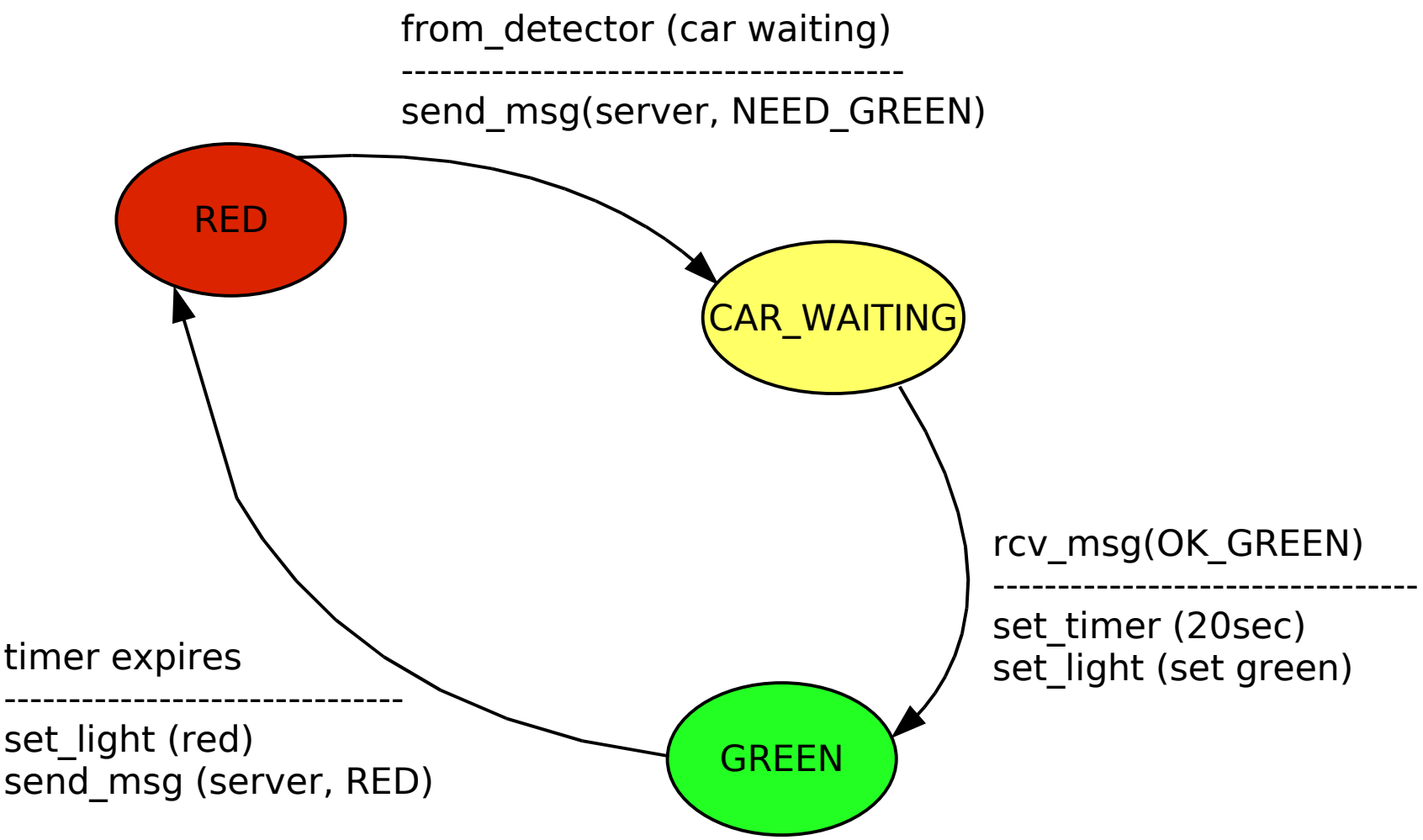
Joint FSM

- Joint FSM:
  - An FSM that captures the evolution of the system as a whole
  - Each state S of the joint FSM corresponds to the set of states each component FSM 1,...,n is in:
    - S=(S1,S2,...,Sn)
  - Every transition (S1,S2,...,Sn) -> (S1',S2',...,Sn') corresponds to one or several transitions in component FSMs
    - If several transitions, they have to be concurrent

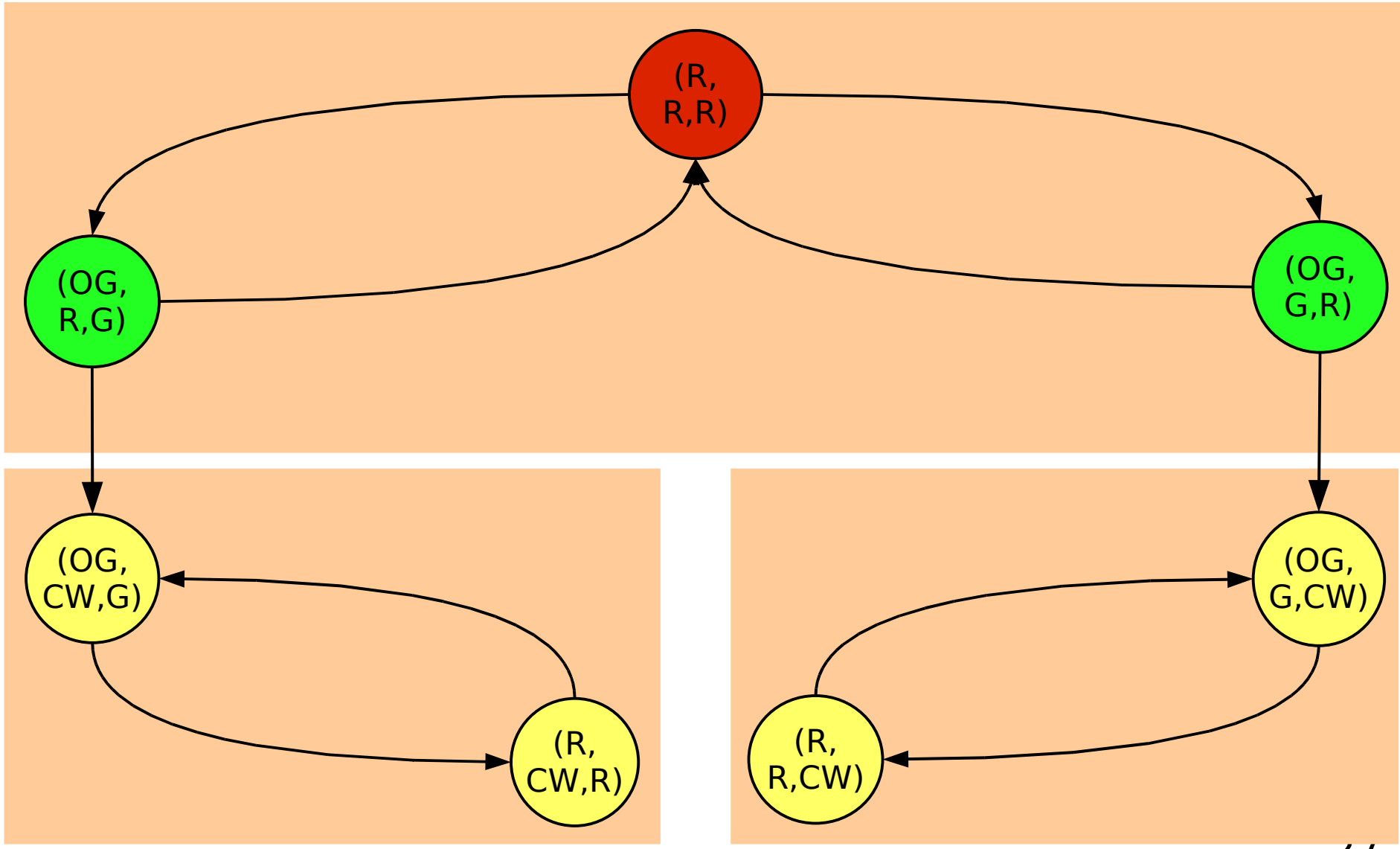
Back to Protocol: Server



Back to Protocol 2: Client

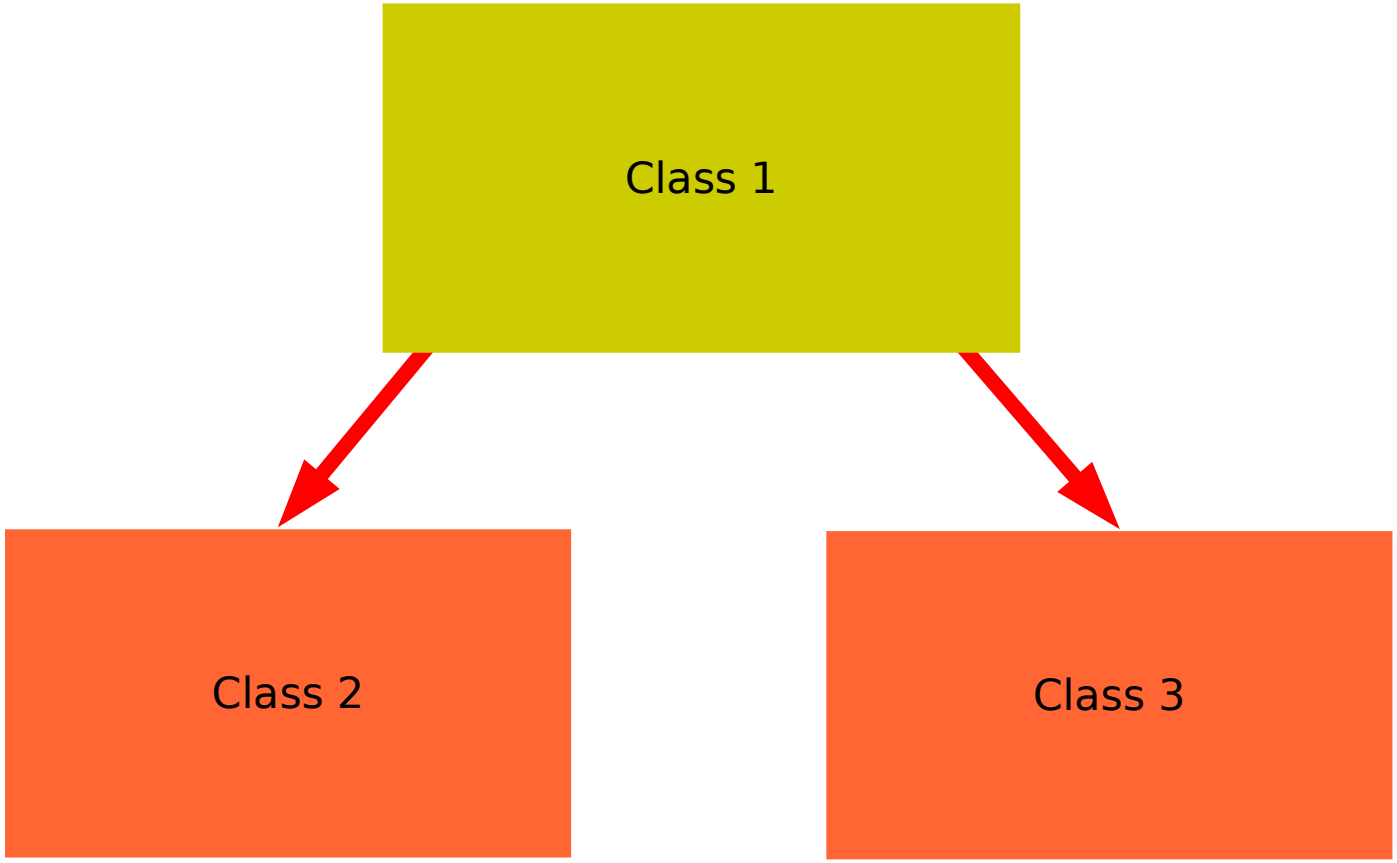


Joint FSM for Protocol 2, 2 Clients, Zero Delay



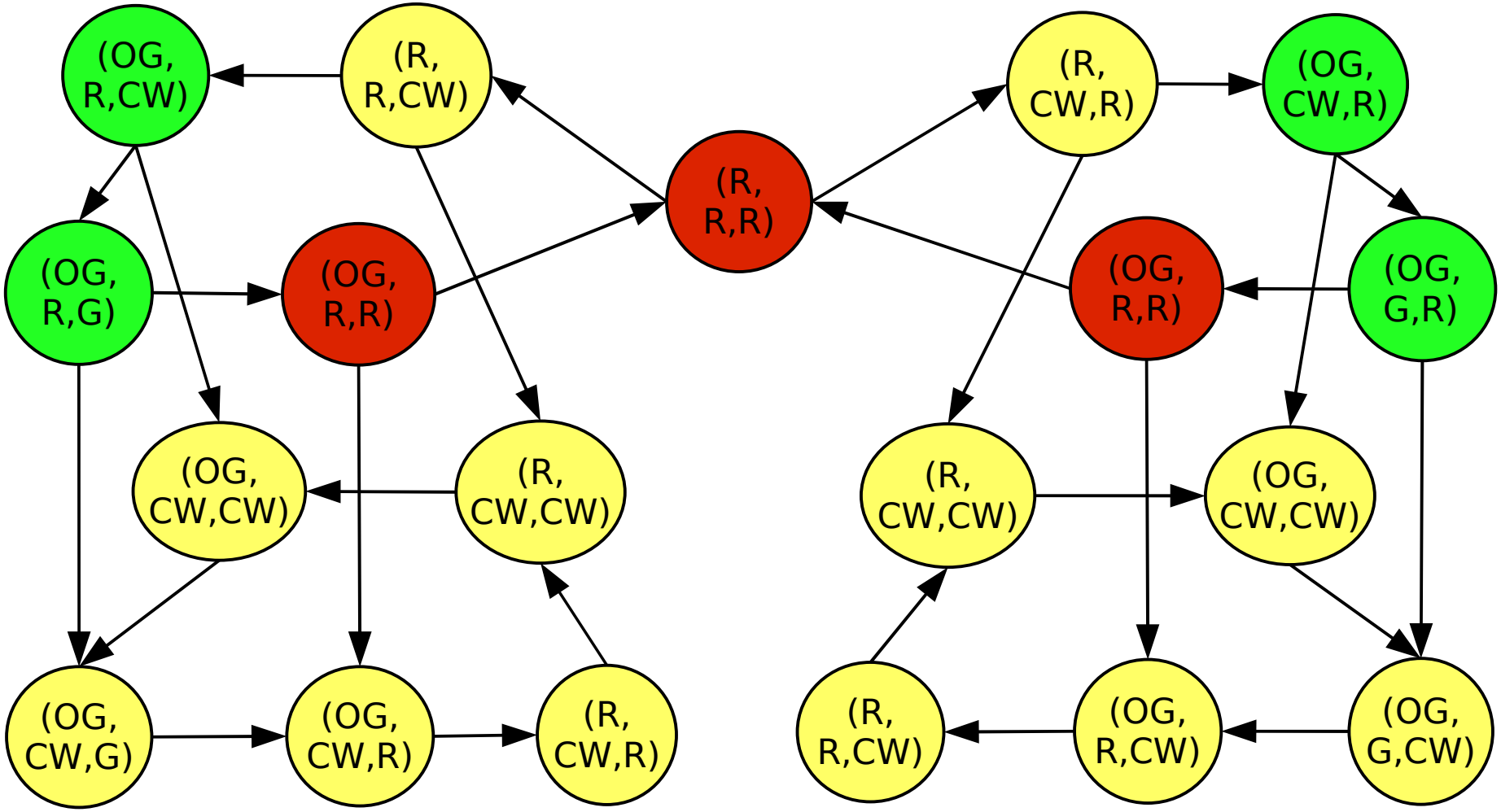
Joint FSM for Protocol 2

- Classes of states
  - We can leave class 1 to either class 2 or 3 and never come back!

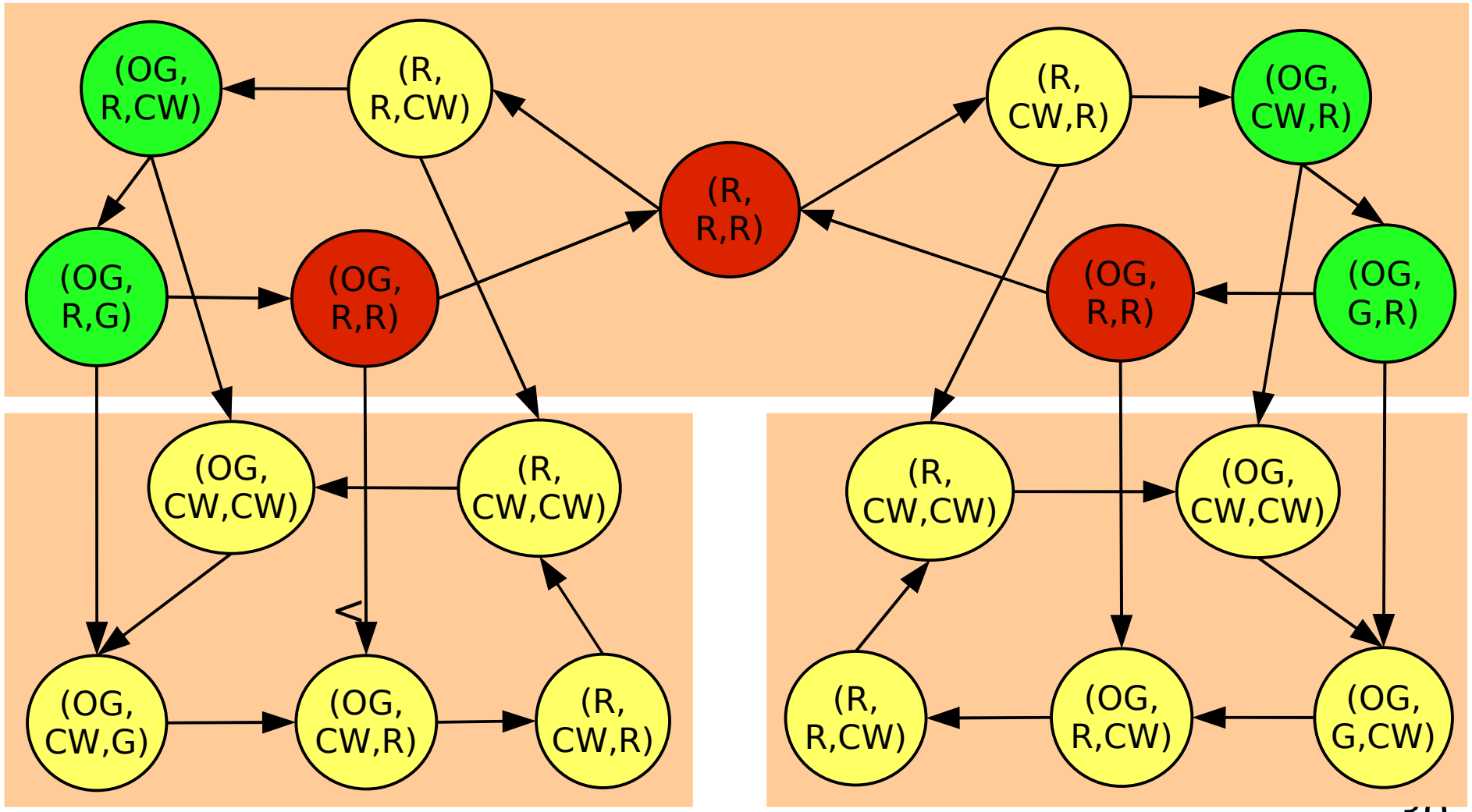


Joint FSM for Protocol 2 with Non-Zero Delay

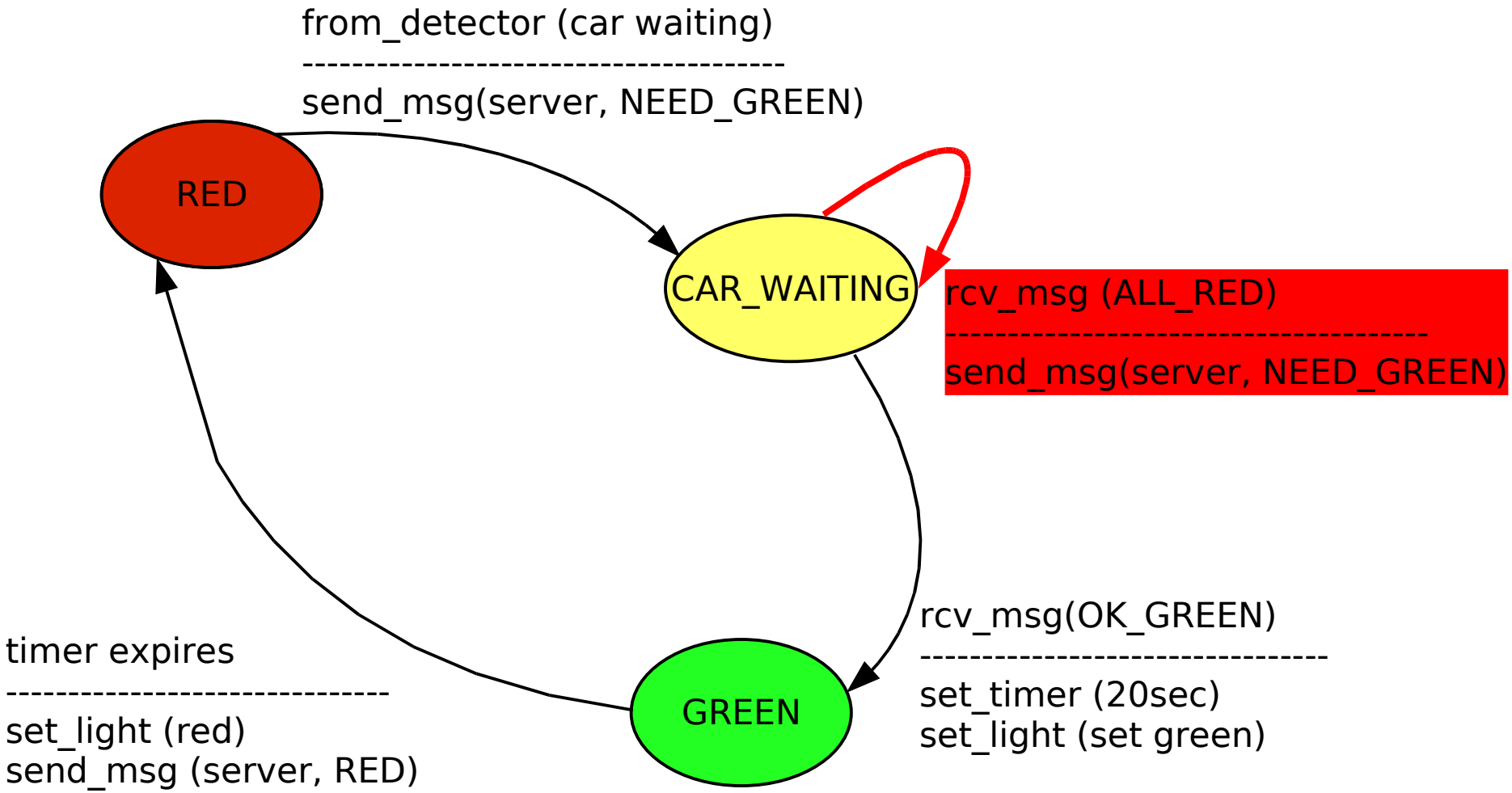
- State = (server, state_client1, state_client2, [channel])



Joint FSM for Protocol 2

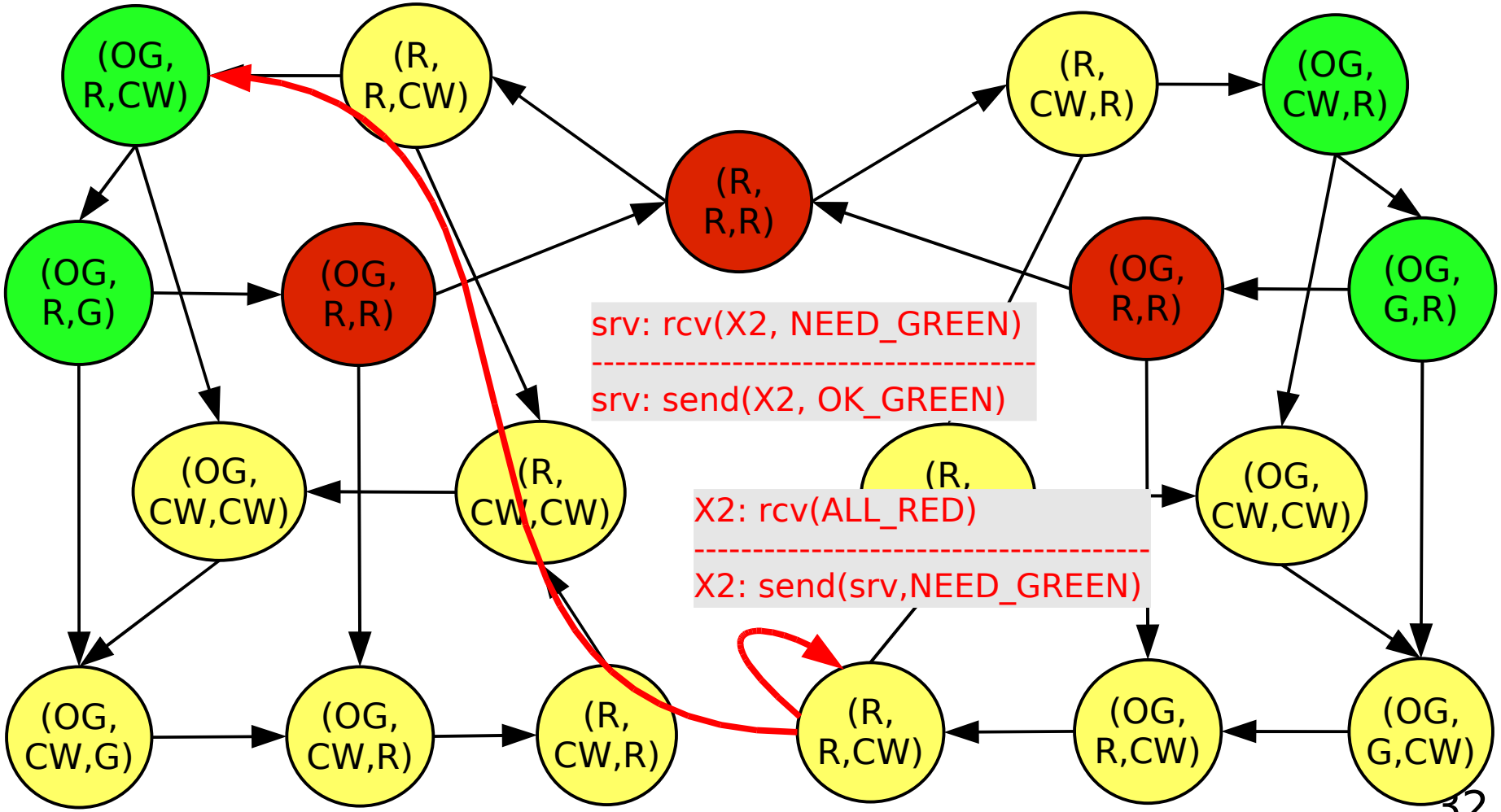


Intersection Client Protocol 3, no Blocking



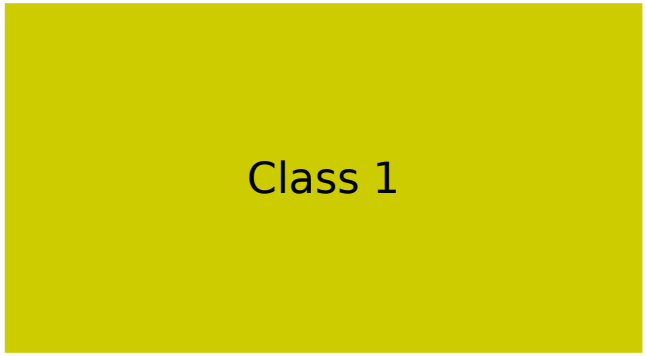
Joint FSM for Protocol 3

- State = (server, state_client1, state_client2)

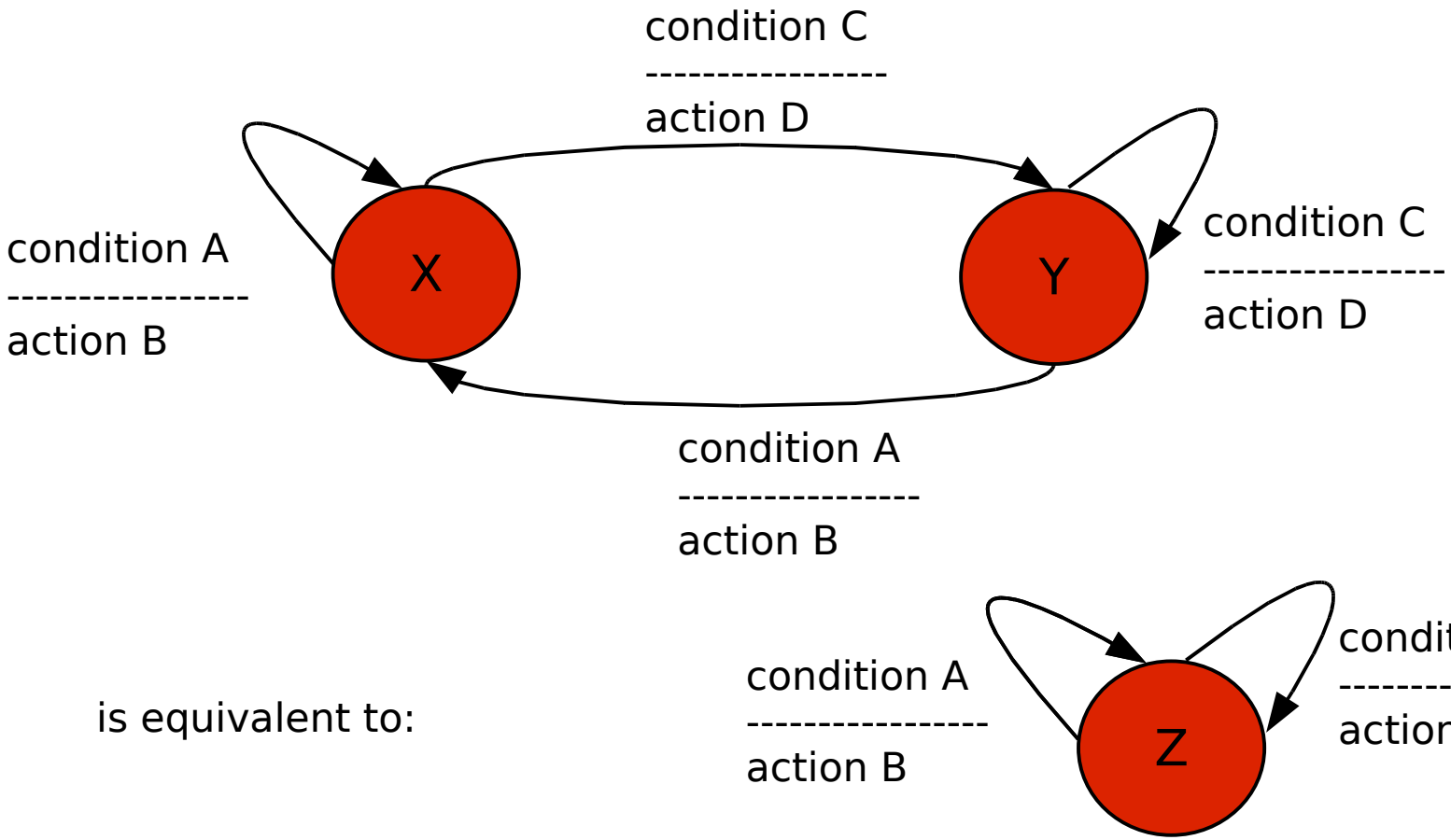


Joint FSM for Protocol 3

- Single class of states
  - We can go from anywhere to anywhere, given the right "input"



Another Example of FSM-Based Analysis



- All that matters is input->output
  - Can we get rid of internal states?
  - Automatic simplification

FSM: Summary

- Individual FSM:
  - Describes individual protocol entity
- Joint FSM:
  - Describes communicating set of entities
  - Can be generated automatically
  - Protocol verification: check properties such as “no absorbing classes of states”, etc.
  - Advantage:
    - Allows mathematical proof of these properties
    - Software tools, code generators
  - Disadvantage:
    - Computational complexity quickly becomes large as FSMs become more complex -> number of joint states explodes

Implementing FSMs

```
case state=A
      case signal=timeout
            state:=B
            send (server, ACK)
            ...
      case signal=request_from_above (new_connection)
            state:=C
      case signal=receive (server, RESPONSE)
            state :=B (self-loop)
            start_timer (20ms)
            to_above (response)
            ...
case state=B
      ...
case state=C
      ...
```

Lessons

- Protocols are hard to get right!
  - Combinatorial explosion of execution paths – or: “so many things can happen!” How to be sure it's ALWAYS right?
  - How to specify a protocol so that different implementations will do exactly the same thing?
  - How to analyze what a protocol does?
- Finite State Machines:
  - Natural “language” to specify and analyze protocols
  - Mathematical theory and software tools
    - Check properties such as no deadlock, starvation, etc.