# Software Model Checking

## A Case Study: High-Availability Protocol

*Moonzoo Kim*

*CS Dept. KAIST*

# HW#3 is due Oct 31

- 1. Implement a quick sort in Promela and show the correctness of your Promela code
  - Use the algorithm in “Intro. To Algorithms 2nd ed” by T.H.Cormen
    - Use recursion as indicated in 146 pg of the book
    - Hint: do it in a similar way to the sieve of the Eratosthenes
  - Assume the following conditions
    - an input array is a byte array of size 4
    - each element of the array is 0~7
  - Write down statistics of your model
    - # of states, # of transition, and the amount of memory your model consumes
  - Increase the size of the array until the memory of your PC becomes exhausted.
    - Write down the maximum size of the array, # of states, # of with your memory

# HW#3 (cont.)

- 2. Implement the Needham-Schroeder (NS) public-key protocol and show that your design makes handshake between A and B successfully
  - http://en.wikipedia.org/wiki/Needham-Schroeder
- 3. Augmenting your NS protocol with modeling an intruder and show the vulnerability of the protocol
  - Your augmented model should work without an intruder, i.e, A and B can make handshake regardless of the presence of the intruder
  - Assume that A and B are communicating through internet, which means that intruder can
    - see all messages between A and B
    - send arbitrary messages to A or B
- 4. Generalize your NS protocol model with a general intruder so that attack scenario can be obtained through a counter example

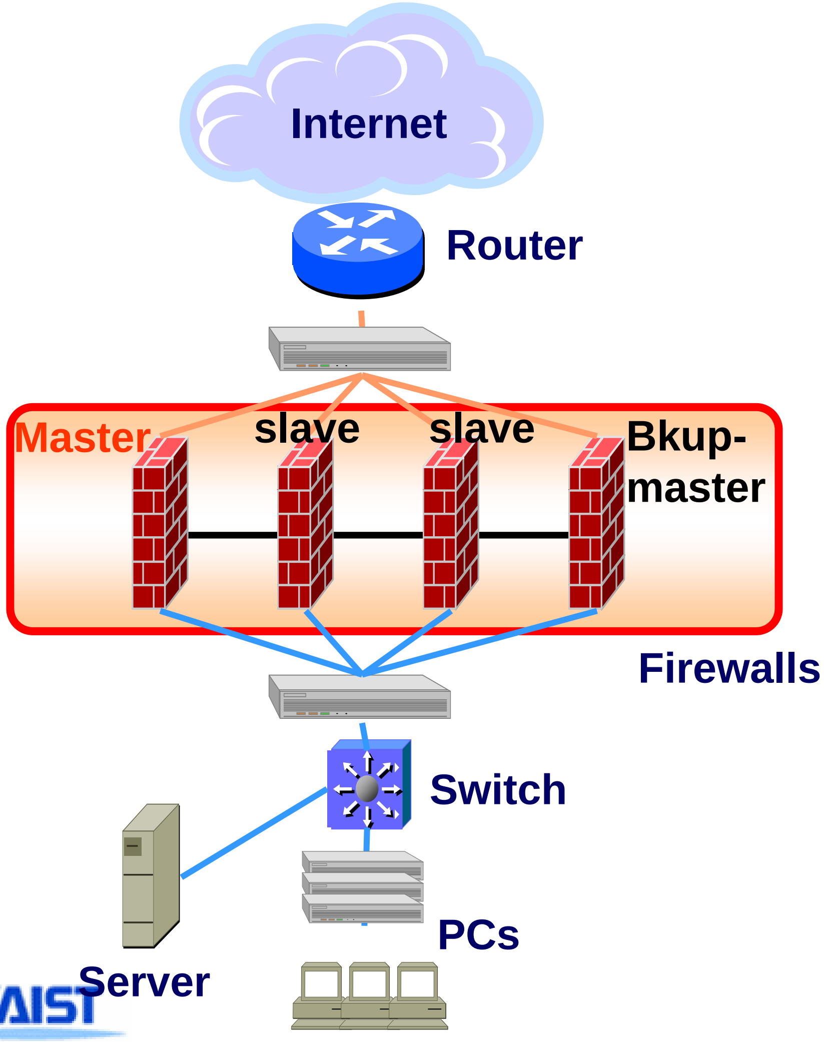
# Last Caution about d_step and atomic

```
byte x;
active proctype A() {

        if
        :: x=1;
        :: x=2;
        fi

}

active proctype B() {
             assert(x!=2);
}
```

```
byte x;
active proctype A() {
        atomic {
        if
        :: x=1;
        :: x=2;
        fi
        }
}

active proctype B() {
             assert(x!=2);
}
```

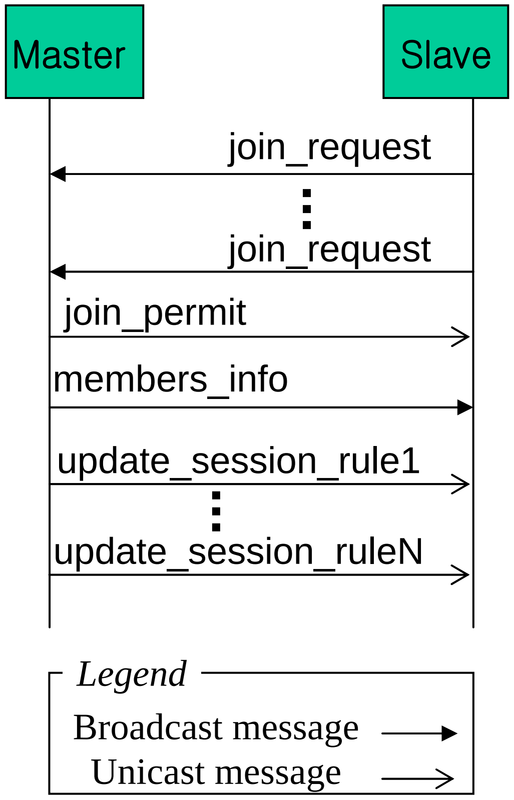
```
byte x;
active proctype A() {
        d_step {
        if
        :: x=1;
        :: x=2;
        fi
        }
}

active proctype B() {
             assert(x!=2);
}
```

# Overview of the High-Availability Protocol



- Multiple firewalls are often deployed in a group for both fault-tolerance and increased throughput
- HA protocol manages a group of firewalls as if there exists single firewall
  - Synchronize information among the group such as session info, etc
  - Elect a master and a bkup-master to coordinate firewalls

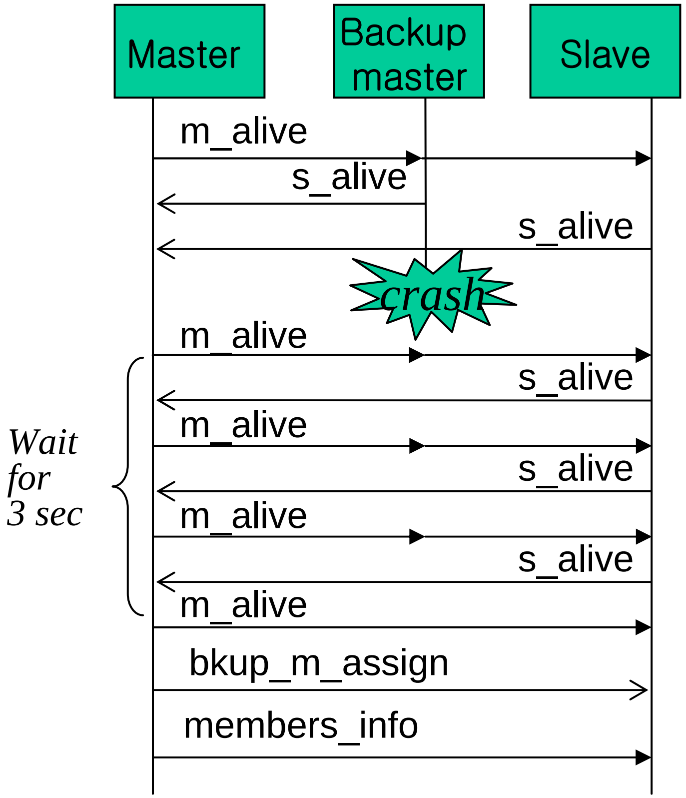
# Specification of the HA Protocol



**a) Message sequence of adding a slave**

- When a slave becomes operational, the slave broadcasts `join_request`
- The master allows the slave to join the group by sending `join_permit`
- Then, the master sends member information via members_info and and session information via `update_session_rule`

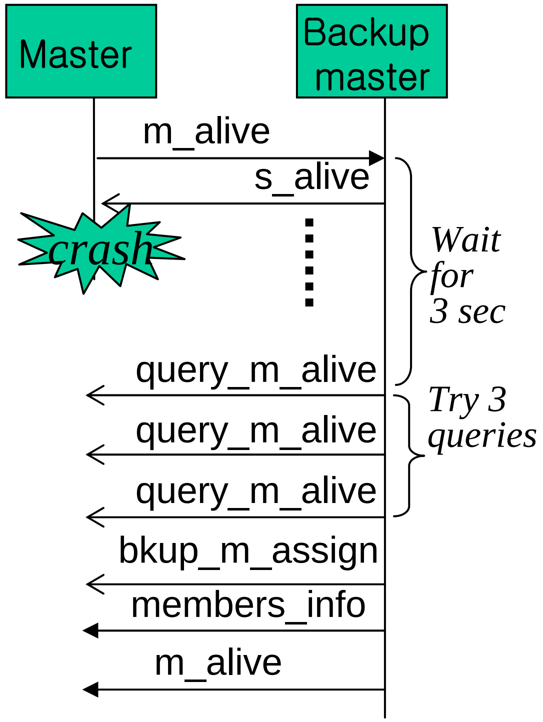
# Specification of the HA Protocol



**b) Message sequence of deleting a backup master**

- A master assigns a slave as a backup master to prepare a case of master crash
- A master broadcasts `m_alive` heartbeat messages to the slaves in the group. Similarly each slave sends a `s_alive` heartbeat message to the master
- If a master does not receive `s_alive` for 3 seconds, corresponding slave is removed
- A master sends a backup master assignment message `bkup_m_assign` to a slave if a backup master is dead

# Specification of the HA Protocol



**c) Message sequence of changing a master**

- When a backup master does not receive `m_alive` for three seconds, the backup master sends three queries to the master to confirm whether it really crashes
- Then, the backup master becomes a master and assigns a new backup master and broadcasts new `members_info`
- When a firewall recovers from a crash, it starts as a slave
  - Firewall 0 starts as a master after recovery if there exists no master

# Requirement Properties

- Deadlock-free property
  - Can be checked with spin's default option
- Single master property
  - [] assert(# of master <= 1)
- Fault-tolerant property
  - $\phi$ = [] $\exists$ i $\in$ Group. working(i)
  - For N=3,
    - [] (working[0] || working[1] || working[2])

# Requirement Properties (cont.)

- But the HA protocol cannot satisfy the fault-tolerant property due to physical constraints
  - A machine may crash for several reasons which are out of our control
    - Ex. Power failure, network line failure, etc
- We need more refined/weakened fault-tolerant property which our model can satisfy
  - $\phi' =$ $[](\forall\, i \in G.(\, \neg alive(i) \rightarrow <>(\exists\, j \in G.\ working(j))))$

# Requirement Properties (cont.)

- But still $\phi'$ is not fully satisfactory because
  - $\phi'$ does not require recovery of crashed machine
    - i.e., a machine does not have to join the group after recovery from crash
    - This is not desirable for the HA protocol because it pursuits increased network throughput by recovering crashed machine as well as fault-tolerance
- Final requirement property $\phi''$
  - $[] (\forall\ i \in G.(alive(i) \rightarrow <>(working(i) \lor \neg alive(i)))))$
- What is still missing?

# Abstractions of the HA Design

- We have to simplify the HA model in order to get a useful result with reasonable computing resource
  - Abstraction of general crash behaviors
    - We limited possible crash scenarios
  - Abstracted heartbeat messages
    - Use a global variable live[N] instead
  - Abstracted channel communications
    - We add a special channel (ch2mst) to make join activity simpler
    - We reduced a possible types of messages, and thus, reduce necessary size of buffer

# Abstraction of General Crashes

- Do we model a general/random crash?
  - A general crash (finest granularity of a crash) can be modeled using unless statement

```
bool crash[N];
active proctype firewall() {
     machine_init:
     { …} unless {crash[_pid]; crash_behavior(); goto machine_init}
}
active proctype  random_crash() {
    do
    :: atomic{crash[0]=false->crash[0]=true}
    :: crash[0]=false
    …
    od
}
```

  - We should be careful about every possible crash behavior in order to prevent deadlock due to the crash
    - ex. flushing buffer, timeout of communicating party, etc
- Instead, we allow a firewall to crash at only special states

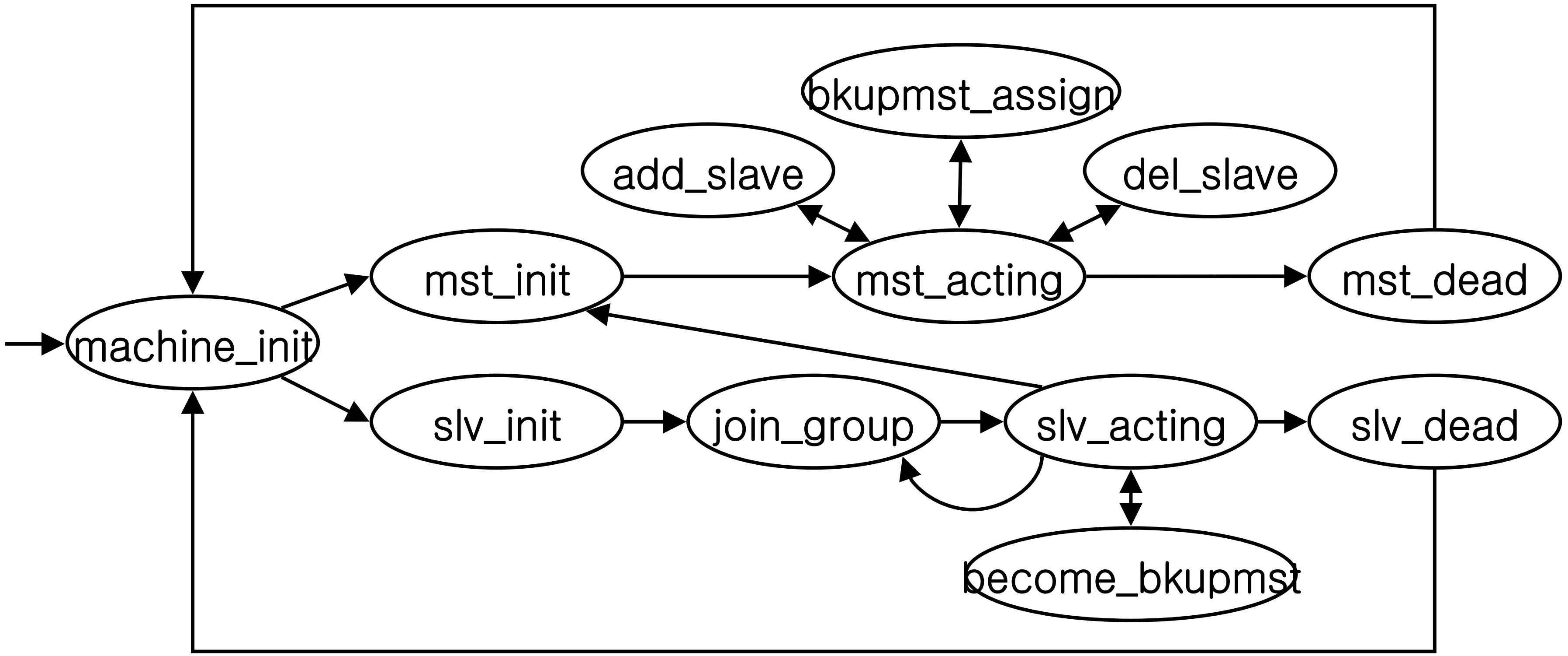
# Abstracted Heartbeat Messages

- To model a real-time behavior is a complex task, especially using a modeling system which does not support real-time with its primitive operators
- For general heartbeat messages, we need to model a synchronization among processes to simulate time advance
- A firewall must handle heartbeat messages in time (within “1 sec”). And a firewall must handle heartbeat message concurrently with other messages (extra concurrency required)
- Channels between a master and slaves should be flushed appropriately when a firewall is dead in order to prevent unnecessary deadlock due to full channel buffer
- We decided to model heartbeat messages using global boolean variables alive[N]

# Abstracted Channel Communications

- Originally, a slave broadcasts join_request messages repeatedly until it receives join_permit. We created a special channel (ch2mst) designated to a current master
  - A slave needs to send only one join_request message to the channel
  - This abstraction models livelock into deadlock, which can be detected more efficiently
- We also use a global variable instead of using bkup_m_assign
- We do not model update_session_rule<N>, members_info, etc. In other words, our model is not detailed enough to to show session-over behavior
- As a result, we have only two messages join_request and join_permit which reduces necessary buffer size as
  - chan ch2mst = [N] of {mtype,byte};
  - chan ch2s[N] = [1] of {mtype,byte};

# Modeling the HA Protocol



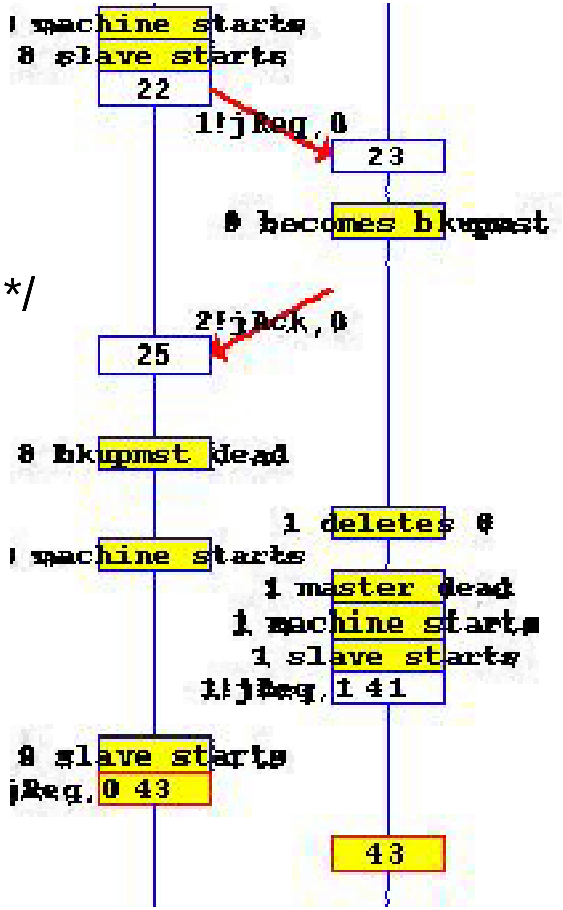
- Each firewall is modeled as a process starting at `machine_init` state
- Depending on its context, a firewall is configured as a master (`mst_init`) or a slave (`slv_init`)
- A slave becomes a master through a transition from `slv_acting` to `mst_init` via `become_mst`
- A fault can occur at only slv_dead and mst_dead states

# Modeling the HA Protocol in PROMELA

```
#define MACHINE_INIT 1
#define NULL 255

bool alive[N];
bool working[n];
byte mst=NULL, bkupmst=NULL;
mtype = {jReq,jAck}
chan ch2mst = [N] of {mtype,byte};
chan ch2s[N] = [1] of {mtype,byte};
…
Inline machine_init() { …}
active [N] proctype firewall() {
  byte current=MACHINE_INIT, next=MACHINE_INIT;
  ...
  do
      /* normal behavior */
      :: atomic{ next==MACHINE_INIT -> current=MACHINE_INIT; machine_init();}
      :: atomic{ next==MST_INIT -> current=MST_INIT; mst_init();}
      …
      :: atomic{ next==BECOME_MST -> current=BECOME_MST;become_mst();}
od
}
```

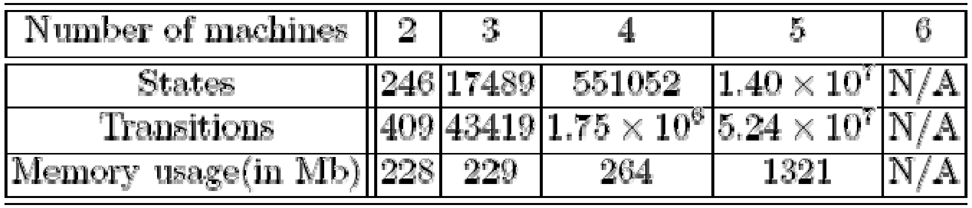
# Modeling the HA Protocol in PROMELA

```
inline machine_init() {
  d_step{
  printf("MSC: %d machine starts\n",_pid);
  if /* If this machine is a statically configured master, and
    there exists no master, the machine starts as a master */
  :: mst == NULL && _pid == 0 ->
     mst = 0;
     printf("MSC: %d master starts\n",_pid);
     next=MST_INIT
  :: else ->
     printf("MSC: %d slave starts\n",_pid);
     next=SLV_INIT
  fi;
  }
}
```



# Verification Results

- We could generate state space upto N=5
- Single master property is satisfied
  - we need to verify the property 4 times for N=2,3,4,5
- We found that the model has a deadlock

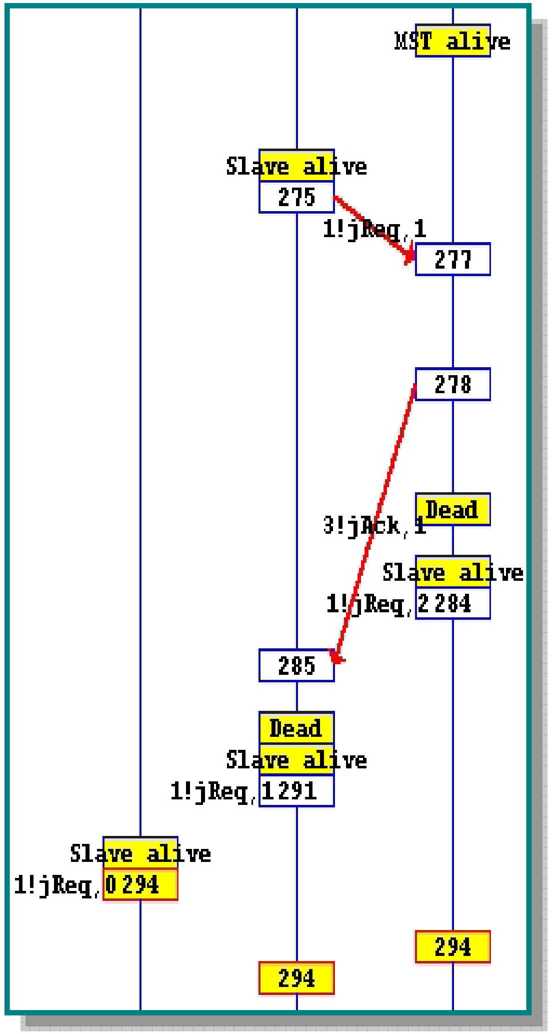


| Number of machines | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| States | 246 | 17489 | 551052 | $1.40 \times 10^7$ | N/A |
| Transitions | 409 | 43419 | $1.75 \times 10^6$ | $5.24 \times 10^7$ | N/A |
| Memory usage(in Mb) | 228 | 229 | 264 | 1321 | N/A |

**Table 2.** Statistics on the HA protocol model

# Identification of an Bugs Causing Deadlock

- The counter example shows that all machines are slaves at **join_group state.** Thus, no master exists to accept new slaves and progress is blocked
  - Could we conclude that this is the only cause for deadlock?
- We analyzed all counter examples and found that all machines are slaves. Thus, we can conclude that master election has a problem
- Thus, it is clear that our HA model does not satisfy $\phi$”



# Identification of Bugs Causing the Deadlock

- Bug $B_1$
  - A master (machine 1) died immediately after a backup master (machine 0) had died and revived as a slave. Then, machine 1 revived as a slave and all machines became slaves.
- Bug $B_2$
  - A master elected a machine that was dead, as a backup master without knowing that the machine was dead. Then, the master died and it happened that there existed no master.
- Bug $B_3$
  - A backup master died immediately after a master had died and revived as a slave.  Then, the backup master revived as a slave and all machines became slaves