ST24 Distributed Algorithms (COMP60009 @ Imperial College London) by ck21

Broadcast

Broadcast Classes One Shot (each message considered separately from others). Unreliable Best-Effort Broadcast (BEB), Regular Reliable Broadcast (RB), Uniform Reliable Broadcast (URB). Multi Shot (involves all messages that are broadcast). FIFO Message Delivery, Causal Order Message Delivery, Total Order Message Delivery.

PL: Perfect P2P Links. Distributed processes communicate with each other using Pefect P2P Links. Reliable Delivery (L) - If A and B are correct processes then every message sent by A to B is eventually delivered by B. No Duplication (S) - No message is delivered to a process more than once. No Creation (S) - No message delivered unless it was sent.

BEB: Best Effort Broadcast. Given a list of all nodes and a msg, a process could broadcast the msg with multiple sends - for p <- processes do PL.send(p, message) end. If msg sending reliable, then for BEB, the message will be delivered to every correct process. If the broadcasting process crashes during sending, then some arbitrary subset of processes will receive the message (no delivery guarantee). Validity (L) - If a correct process broadcasts a message then every correct process eventually delivers it. No Duplication (S) - No message delivered to a process more than once. No Creation (S) - No message is delivered unless it was broadcast. BEB Basic Broadcast (fail-silent) - send message to each process using PL. Works because PL will ensure all correct processes will deliver the message if the sender of the message does not crash. Algo is fail-silent if process crashes can never be reliably detected. Performance - $O(N)$, where N is the number of processes.

RB: Reliable Broadcast. For BEB, if the sending process crashes during a broadcast, then some arb subset of processes will receive the msg. BEB no delivery agreement guarantee - correct processes do not agree on the delivery of the message. RB algos have this - reg RB all correct processes will agree on the messages they deliver, even if the broadcasting process crashes while sending. Validity, No Duplication and No Creation all same as BEB. Agreement (L) - If a correct process delivers message M then every correct process also delivers M. V and A together provide a Termination Property for broadcast. Only correct processes are required to deliver the message - faulty processes could deliver messages not delivered by correct processes.

Eager RB Fail-Silent. Every process re-broadcasts every message it delivers. So, if the broadcasting process crashes, the msg will be forwarded by other processes using BEB. Performance - $O(N)$ BEB Broadcasts, $O(N^2)$ messages.

Lazy RB Fail-Stop. Uses BEB with failure detector to detect processes that have failed. Agreement derived from validity property of BEB, that every correct process BEB-broadcasts every message that it deleivered from a crashed process and properties of PFD.

Failure Detectors. PFD P - Provides processes with a list of suspected (detected) processes that have crashed. Makes timing assumptions (system no longer asynchronous). Never changes its view - suspected processes remain suspected forever. <> P - Eventually PFD. May make mistakes but will eventually accurately detect a crashed process. Strong Completeness (L), Strong Accuracy (S), Eventually strong accuracy (L). PFD Exclude on Timeout - Uses PL to exchange heartbeat messages (req repl) and uses a timeout period for replies to suspect a crashed process. PL performs reliable sending for correct processes. Timeout period needs to be large enough to send a heartbeat message to all processes, processing at the receiving processes and getting replies back (*synchrony* assumption).

PFD Safety and Liveness. Strong Completeness (L) - every prcoess that crashes will eventually be permanently suspected by every correct prcoess. If a process crashes it stops replying to heartbeat messages and no process will deliver its reply. PL ensures that no message is delivered unless sent. Every correct process will thus detect the crash. Strong Accuracy (S) - no process is suspected before it crashes. A process is suspected only if no heartbeat reply is delievered before timeout. Can only happen if hte process has crashed under our timing assumption. Reply delivered before timeout.

URB: Uniform Reliable Broadcast. Validity, No Duplication and No Creation same as BEB and RB. Uniform Agreement (L) - if a process delivers message M then every correct process will also deliver M. Implies the set of messages delivered by a faulty process is always a subset of messages delivered by a correct process (stronger guarantee).

Majority-Ack URB. urb_deliver message only after the message has been beb_delivered by a majority of correct processes. Fail-silent algorithm where process crashes are not reliably detected. Does not use a FD. Assumes that majority of processes are correct. If f processes might crash then we need at least $2f+!$ processes. - i.e. we need to have a majority of at least $f+1$ correct processes. For correct process P and message M, if P beb_delivers M, then eventually P urb_delivers M. If P beb_broadcasts M then all correct prcoesses $beb_broadcast\ M$. Given the majority assumption, P will eventually beb_deliver M from the majority of processes and will then urb_deliver M. Correctness - no creation (S) follows from BEB. No duplication (S) because algo keeps track of messages that have been urb_delivered. Performance - best case 2 steps. 1st step requires N messages. 2nd rebro step requires $N(N-1)$ messages, i.e. we need $N + N(N-1)$ messages. Validity (L) because P urb_broadcasts $M \rightarrow P$ eventually beb_delivers $M \rightarrow P$ urb_delivers M. Uniform agreement (L) because if Q is any process that urb_delivers M. Implies Q beb_delivered M from the majority of processes (which we assume is a correct majority) so at least 1 correct process must have beb_broadcast M. Hence all correct processes will eventually beb_deliver M (validity of BEB) and will also eventually urb_deliver M.

Ordering. FIFO, Causal Order (CO), Total Order (TO) - all correct processes deliver the same global seq of msg. Impossible in purely async system. FIFO delivery - proc broadcasts M1 before M2 then all correct processes will deliver M1 before M2. Perf same no of messages as RB broadcast + extra memory. CO RB - msgs delivered in a way that respects causal order. Causal order relation $m1 \rightarrow m2$ states that message $m1$ may have caused $m2$ if: (1) FIFO order (A broadcasts $m1$ and later $m2$), (2) Local Order (A delivers $m1$ and later broadcasts $m2$), (3) Transitivity ($m3$ such that $m1 \rightarrow m3$ and $m3 \rightarrow m2$). Causal Delivery Property: if any process delivers $m2$ then it must previously have delivered every mesage $m1$ such that $m1 \rightarrow m2$. Causal delivery is ensured by every message carrying its causally past messages and ensuring they are crb_delivered before the message. No duplication, no creation, validity and uniform agreement are derived from the use and properties of URB. Performance - no additional messages over URB but message size grows linearly. Past grows very large. We can improve algo by GC messages from past.

CRB with Vector Clocks. Each CRB process keeps array of seq numbers indexed by integer process num 1 to N. VC entry for process P indicates num of messages from process P that have been crb_delivered. Each CRB process also maintains a count of the num of msgs that it has rb_broadcast. (=crb_broadcasts received). When CRB rb_broadcasts a message, it includes its VC with its own entry in the VC set to the num of crb_broadcasts it has previously made. When message M is rb_delivered, CRB compares the VC sent with the message with its own VC. Any difference in VC entries for the receiving process and the sending process indicates the number of messages that must be delivered before M can be crb_delivered. CRB must defer delivering if necessary - done with <= element wise comparison between VCs. Process P2 RB_delivers m1 and CRB_delivers m1 since its VC P1 $\leq$ VC P2 (VC1[0,0,0] $\leq$ VC2[0,0,0]). After CRB_deliver P2 updates VC to VC2[1,0,0]. P3 RB_delivers m2 first. However it cannot CRB_deliver m2 since VC2[1,0,0] $\leq$ VC3[0,0,0]. However when m1 is RB_delivered we have VC1[0,0,0] $\leq$ VC3[0,0,0] so P3 can CRB_deliver m1, which will update VC3 to VC3[1,0,0]. P3 can then CRB_deliver m2 since VC2[1,0,0] $\leq$ VC3[1,0,0] and update VC3 to VC3[1,1,0] ensuring CO is maintained. Agreement property of RB will ensure that if a correct process delivers message M then every correct process will also deliver M. This ensures that RB eventually delivers every message that causally preceded M, allowing the VC comparison to succeed and M to be crb_delivered. Causal delivery property ensured (correctness).

```
defmodule CRB_broadcast_vc
def start() do
  receive do
    { :bind, c, rb, pnum, processes }
      -> %{c: c, rb: rb, pnum: pnum, rb_broadcasts: 0,
    pending: empty_set(), vc: init_map(processes,0)}
    |> next()
  end
end
defp next(this) do
  receive
    { :crb_broadcast, msg } ->
      vc2 = Map.put(this.vc, this.pnum, this.rb_broadcasts)
      this |> rb_broadcasts(this.rb_braodcasts+1) |> next()
    { :rb_deliver, sender, { :crb_data, s_msg, s_vc }} ->
    this |> pending_put({sender, s_msg, s_vc })
    |> check_pending_and_deliver()
    |> next()
  end
end
defp check_pending_and_deliver(this) do
  case Enum.find(this.pending, fn {_, _, s_vc} -> s_vc <=
  this.vc end)
  do
    {sender, s_msg, s_vc} = data ->
      send this.c, {:crb_deliver, sender, s_msg}
      this |> vc_elem_put(sender,this.vc[sender]+1)
      |> pending.delete(data)
      |> check_pending_and_deliver()
    _otherwise ->
      this
  end
end
end
```

TO/Atomic Broadcast. All correct processes must deliver all messages in same order. TO does not need to respect causality or FIFO. Essential when replicas need to treat requests in the same order to preserve consistency. Pubsub service. TO URB: Additional property. If a process (correct or crashed) delivers message M1 without previously delivering message M2, then no correct process delivers M2 before M1. TOB equivalent to consensus.

Consensus

Intro. Each process proposes a value. All correct processes agree on one of the proposed values in finite time. Values to be agreed are often actions to be carried out by replicated servers.

RAFT. Leader-based, decomposes problem (normal op, leader changes). Simplifies normal operation - no conflicts. More efficent than leader-less approaches. Leader - handles all client interactions, log rep. At most 1 viable leader at a time. Follower - completely passive (issues no RPCs, responds to incoming RPCs). Candidate - used to elect a new leader. Normal op - 1 leader, $N-1$ followers. Time divided into terms - election, normal op. Failed election - terms with no leader. Each server maintains current term value. Terms identify obsolete info. Persistent server state - currentTerm, votedFor, log[], state, leader, commitIndex, nextIndex[], matchIndex[].

RPCs. AppendEntries - add an entry to the log. Empty messages used as heartbeats. Vote - Message used by candidates to ask for votes and win elections. Heartbeats - servers start up as followers. Followers expect to receive RPCs from leaders or candidates. Leaders must send empty AppendEntries RPCs to maintain authority. If $\Delta_{election}$ time units elapse with no RPCs - follower assumes leader crasshed. Follower starts new election. Timeouts 100-500ms. **Election Start**. Set new timeout in range $[\Delta_{election}, 2 \cdot \Delta_{election}]$. Increment current term. Change to candidate state. Vote for self. Send Vote RPCs to all other servers, retry until either - (1) receive votes from majority of servers, become leader and send AppendEntries heartbeats to all other servers, (2) receive AppendEntries heartbeats from valid leader and return to follower state, (3) no one wins election (election timeout elapses) - start new election.

Election Correctness. Safety - allow at most one winner per term. Each server gives out only one vote per term (persist on disk). Two different candidates can't accumulate majorities in the same term. Liveness - some candidates must eventually win. Choose election timeouts randomly in $[\Delta_{election}, 2 \cdot \Delta_{election}]$. One server usually times out and wins election before others wake up. Works well if $\Delta_{election} >>$ broadcast time.

Logs. Logs stored on stable storage and survives crashes. Entry committed if known to be stored on majority of servers. Durable and eventually executed by state machines. Normal Op - client sends command to leader. Leader appends command to log. Leader sends AppendEntries RPCs to followers. Once new entry committed - leader passes command to state machine, returns result to client. Leader notifies followers of committed entries in subsequent AppendEntries RPCs. Followers pass committed commands to state machines. Crashed/slow followers - leaders retry RPCs until succeed. Perf optimal in common case - one successful RPC to any majority of servers. Log Consistency - if log entries on different servers have same index and term, they store the same command. The logs are identical in all preceding entries. If a given entry is committed, all preceding entries are also committed.

Consistency Check. Each AppendEntries RPC contains index, term of entry preceding new ones. Follower must contain matching entry, otherwise it rejects request. Implements induction step to ensure coherency. At beginning of new leaders term, old leader may have left entries partially replicated. No special steps by new leader - just start normal op. Leaders log is truth. Will eventually make follower's log identical to leaders. Multiple crashes can leave many extraneous log entries.

Safety Requirement. Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry. RAFT safety property - if a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders. This guarantees the safety requirement. Leaders never overwrite their logs. Only entries in the leader's log can be committed. Entries must be committed before applying to state machine. Committed (restrictions on commitment) - present in future leader's logs (restrictions on leader election). Best leader - during elections choose candidate with log most likely to contain all committed entries. Candidates include index and term of last log entry in VoteReq. Voting server V denies vote if log is more complete ($lastLogTerm_c < lastLogTerm_v$) or ($lastLogTerm_c = lastLogTerm_v$ and $lastLogIndex_c < lastLogIndex_v$). Leader has most complete log among electing majority.

New Commitment Rule. For a leader to decide that an entry is committed: must be stored on a majority of servers. At least one new entry from leader's term must also be stored on majority of servers. Once entry 4 committed: s_5 cannot be elected leader for term T_5. Entries 3 and 4 both safe.

Repair Follower Log. New leader must make follower logs consistent with its own. Delete extraneous entries and fill in missing entries. Leader keeps nextIndex for each follower. Index of next log entry to send to that follower. Initialised to 1 + leader last index. When AppendEntries consistency check fails, decrement nextIndex and try again. When follower overwrites inconsistent entry, its deletes all subsequent entries.

Neutralise Old Leaders. Deposed leader not dead. Disconnected. Other servers elect new leader, Old leader reconnects and attempts to commit log entries. Use terms to detect stale leaders and candidates. Every RPC contains term of sender. If sender's term is older, RPCs is rejected, senders reverts to follower and updates its term. If receiver's term is older, it reverts to follower, updates its term and processes RPC normally. Election updates terms of majority of serves. Deposed server cannot commit new log entries.

Client Protocol. What if leader crashes after executing command but before responding - must not execute command twice. Client embeds unique ID in each command. Server includes ID and response in log entry. Before accepting command, leader checks log for entry with that ID. If ID found in log, ignore new command and return response from old command. Exactly-once semantics as long as client doesn't crash.

FLP Impossibility

Synchrony. Async - process exec steps and IPC can take place at arb time. No assumption that proc has phy clocks but sometimes useful to use logical clocks. Sync - upper bound on proc delays (time taken to exec step - receive msg do comp send msg). Upper bound on time for msg delivered (time from sending msg to it beign delivered to a destination proc). Perf may be constrained by bounds. Partially sync - real-world sys mostly sync with periods they aren't. Assume eventually sync. Comm assumptions - (1) async msg passing - proc sending msg continues after sending msg does not wait for msg to be delivered to dest proc. Build sync MP abstraction using async MP. (2) Reliable msg commms - assume msgs sent using reliable MP not dropped.

Result. Any algo for consensus in an async system has executions that do not terminate, even if only one process crashes. Result is true even if no process crashes.

Valent Configs. 0V - configs with 0 decision value. 1V - 1 decision val. Univalent (k valent) - configs with k decision value. k is 0 or 1. Config C decides value V if some process has an output bit V. Bivalent - indecisive.

Proof. Execution in which algo A never reaches a decision - indecisive forever. First step - initial bivalent config. Second step - possible to keep doing steps without reaching univalent config. Forever bivalent. Lemma 1 - consider config C with schedules σ_1 and σ_2. Config $C1 = \sigma_1(C)$ and $C2 = \sigma_2(C)$. 2 sets of processes

taking steps in σ_1 and σ_2 are disjoint - then $\sigma_2(C1) + \sigma_1(C2) = C3$. Commute Property. Lemma 2 - Algo A has initial bivalent config. Assumption: no initial bivalent config (equivalently all initial configs are univalent). Must be adjacent 0V and 1V configs which differ in the state of one process p. Apply a sequence where process p does not take any steps (p fails immed). **Contradiction** - correct process decides same val for adjacent configs. Contradiction since one config is 1V and other is 0V - so one must be bivalent. Lemma 3 - C bivalent for Algo A. $E = (P, M)$ for event applicable to C. $\mathcal{C}$ is set of configs reachable from C without E. $\mathcal{D}$ is set of configs reachable from C by applying E = E(Cn) — Cn element of $\mathcal{C}$ and E is applicable to Cn . Then $\mathcal{D}$ contains a bivalent config. Informal - if we delay a message that is pending by some amount from one event to arb many, then there will be one config in which you receive the message and end up in an undecided state. Essentially - we end up going from one bivalent config to another by delaying a message - the algo remains indecisive forever. **Neighbouring Configs**. Assume D only contains univalent configs then derive a contradiction to prove that D contains a bivalent config. 2 configs C0 and C1 are neighbours if one results from other in a single step.

Theorem. There is an execution in which Algo A never terminates. Any deciding run allows the construction of an ∞ non-deciding run. Combine sections of runs to avoid making a decision (univalent). Proof - Lemma 2 states taht there exists an initial bivalent config. Apply Lemma 3 repeatedly - take steps to go from one bivalent config to another bivalent config repeatedly. Operationally - start from C0 (bivalent). Put processes in a queue and schedule them Round Robin (every message eventually received). Let E=(P,M) be the first event with message M and process P be the first process in the process queue. By Lemma 3 we can reach C1 (bivalent) where E is the last message received. Keep repeating and never make a decision.

Temporal Logic of Actions

TLA+ ASCII

```
1.  \* Based on https://abt.informal.systems/docs/tla_basics_tutorials/tla+cheatsheet.html ----------------------

2.  \* Logic ------------------------------------------------------------------------------------------------
3.     BOOLEAN                  \* the set of all booleans - same as {TRUE, FALSE}
4.     TRUE                     \* Boolean true
5.     FALSE                    \* Boolean false
6.     ~e                       \* negation
7.     e1 /\ e2                 \* conjunction: e1 and e2;    /\ can also be placed at line start in multi-line conjunctions
8.     e1 \/ e2                 \* disjunction: e1 or  e2;    \/ can also be placed at line start in multi-line disjunctions
9.     e1 = e2, e1 # e2         \* equality, inequality
10.    e1 => e2                 \* implication: e2 is true whenever e1 is true     i.e. (~e1 \/ e2)  or (IF e1 THEN e2 ELSE TRUE)
11.    e1 <=> e2                \* equivalence: e1 is true iff e2 is true

1.  \* Quantifiers ------------------------------------------------------------------------------------------
2.     \A var \in S: e          \* for all elements var in set S, it holds that expression e is true.     Always TRUE  if S={}
3.     \E var \in S: e          \* there exists an element var in set S, such that expression e is true.  Always FALSE if S={}
4.                              \* for \A and \E, TLC requires S to be finite
5.     CHOOSE var \in S : e     \* magically always return the same element from the set {var \in S: e}, undefined if set is empty

6.  \* Integers ---------------- import integer operators with EXTEND Integers ------------------------------------
7.     Int, Nat                 \* the set of all integers (an infinite set), and all natural numbers (includes 0)
8.     1, -2, 1234567890        \* integer; integers are unbounded
9.     e1..e2                   \* integer range; the set of all integers from e1 to e2 inclusive, {} if e1 > e2
10.    e1 + e2, e1 - e2, e1 * e2  \* integer addition, subtraction, multiplication, also \div, % (mod), ^ (power)
11.    e1 < e2, e1 <= e2        \* less than, less than or equal
12.    e1 > e2, e1 >= e2        \* greater than, greater than or equal

13. \* Strings ----------------------------------------------------------------------------------------------
14.    STRING                   \* the set of all finite strings (an infinite set)
15.    "", "a", "hello, world"  \* string; strings can be compared for equality

BLUE sets, RED booleans, YELLOW integers, GREEN functions/records, PURPLE sequences/tuples, BLACK any/other

1.    \* Finite sets ---------- import set operators with EXTEND FiniteSets ------------------------------------
2.       {a, b, c}             \* set constructor: the set containing a, b, c.  TLC requires elements to be of the same type
3.       Cardinality(S)        \* the number of elements in finite set S
4.       e \in S, e \notin S   \* e is an element of set S, e is not an element of S
5.       S1 \subseteq S2       \* set S1 is a subset of set S2
6.       S \union S2           \* union of sets S1 and S2: all elements in S1 or in S2          \cup is shorter alternative
7.       S \intersect S2       \* intersection of sets S1 and S2: all elements in S1 and in S2  \cap is shorter alternative
8.       SUBSET S              \* powerset: set of all subsets of S
9.       UNION S               \* flatten:  set of all elements of elements of S where S is a set of sets
10.      S1 \ S2               \* difference: all elements in S1 but not in S2
11.      {var \in S: e}        \* subset/filter: subset of elements var in S, such that predicate e is true, e may include var
12.      {e: var \in S}        \* constructor: set of expressions e for all var is in S,  e may include var

13.   \* Functions/Maps ------------ maps keys to values ------------------------------------------------------
14.      [k \in KeyS |-> e]    \* function constructor: maps all keys k in KeyS to expression e (e may include k)
15.      fn[a]                 \* function application: the value of function fn at key k (an expression)
16.      DOMAIN fn             \* function domain: the set of keys of function fn
17.      [fn EXCEPT ![k] = e]  \* function fn with key k remapped to expression e (may reference @, the original fn[k])
18.      [fn EXCEPT ![k1] = e1,    \* function fn with multiple keys remapped:
                    ![k2] = e2, ...]  \*     k1 to e1 (@ in e1 will be equal to fn[k1]), k2 to e2 (@ in e2 will be equal to fn[k2])
19.      [KeyS -> ValueS]      \* function set constructor: set of all functions with keys from KeyS and values from ValueS

20.   \* Records ----------------- records are functions/maps internally (field names string values are the keys --
21.      [f1 |-> e1, f2 |-> e2, ...]  \* record constructor:  a record were fieldname f1 equals e1, fieldname f2 equals e2, ...
22.      rec.f                 \* record field access: the value of field f of record rec
23.      [rec EXCEPT !.f = e]  \* record rec with fieldname f remapped to expression e (may reference @, the original rec.f)
24.      [rec EXCEPT !.f1 = e1,    \* record rec with multiple fields remapped
                     !.f2 = e2, ...]  \*     f1 to e1 (@ in e1 is equal to rec.f1), f2 to e2 (@ in e2 is equal to rec.f2)
25.      [f1: S1, f2: S2, ...]  \* record set constructor: set of all records with fieldname f1 in set S1, fieldname f2 in set S2, ..

BLUE sets, RED booleans, YELLOW integers, GREEN functions/records, PURPLE sequences/tuples, BLACK any/other

1.    \* Sequences ---------- import sequence operators with EXTEND Sequences. Sequences are functions indexed from 1 --
2.       <<e1, e2, e3>>        \* sequence constructor: sequence of elements e1, e2, e3    Sequence elements should have the same type
3.       seq[i]                \* the ith element of the sequence seq (indexed from 1)
4.       seq1 \o seq2          \* the sequence seq1 concatenated with sequence seq2
5.       Len(seq)              \* the length of sequence seq
6.       Append(seq, e)        \* the sequence seq with e added to the end. Also SubSeq() and SelectSeq()
7.       Head(seq)             \* the first element of sequence seq
8.       Seq(S)                \* set of all finite sequences over set S

9.    \* Tuples ---------- tuples are functions/maps indexed from 1 ------------------------------------------
10.      <<e1, e2, e3>>        \* tuple constructor: tuple of elements e1,e2,e3                Tuple elements may have different types
11.      tup[i]                \* the ith element of the tuple tup (indexed from 1)
12.      S1 \X S2 \X S3        \* cartesian product: set of all tuples <<m1, m2, m3>>, where m1 is in S1, m2 is in S2, m3 is in S3

13.   \* Miscellaneous ---------------------------------------------------------------------------------------
14.      LET var == e1 \in e2  \* local definition: every occurrence of var in e2 is replaced with e1
15.      IF e THEN e1 ELSE e2  \* e1 if e is true, otherwise e2 if e is false

16.   \* Actions ---------- ------------------------------------------------------------------------------
17.      var'                  \* primed variable: denotes value of non-primed variable var in the next state
18.      UNCHANGED <<v1,v2,v3>>  \* variables v1, v2, v3 are unchanged in the next state; same as v1'=v1 /\ v2'=v2 /\ v3'=v3
19.      [A]_u  [A]_<<v1,v2,v3>>  \* stuttering action A, or all variables v1, v2, v3 unchanged in successor state
20.      <<A>>_u  <<A>>_<<v1,v2,v3>>  \* non-stuttering A and some variables v1, v2, v3 changed in successor state
21.      ENABLED A             \* true if action A is enabled e.g. pre-conditions are true, there exists is true, if condition true

22.   \* Temporal Logic ---------- ------------------------------------------------------------------------
23.      []F                   \* F is always true
24.      <>F                   \* F is eventually true
25.      F1 ~> F2              \* F1 leads to F2
26.      WF_v(A), SF_v(A)      \* weak/strong fairness for action A

BLUE sets, RED booleans, YELLOW integers, GREEN functions/records, PURPLE sequences/tuples, BLACK any/other
```

```
1.  \* Module structure ----------------------------------------------------------------------
2.     ---- MODULE module ----     \* starts TLA+ module (should be in file module.tla).  Min 4 '-' at start of and end of of line
3.     EXTENDS mod1, mod2, ...     \* Imports TLA+ modules mod1, mod2, ...   Can also use singular EXTEND, CONSTANT, VARIABLE
4.     CONSTANTS c1, c2, ...       \* declares constants c1, c2, ...         Defined in configuration file when doing model checking
5.     ASSUME e                    \* asserts expression e as an assumption
6.     VARIABLES var1, var2, ...   \* declares variables var1, var2, ...
7.     OpName == e                 \* defines operator OpName as expression e without parameters.   Operators are applied likes macros
8.     OpName(v1, v2, ...) == e    \* defines operator OpName as expression e with parameters v1, v2, ...,
9.     THEOREM e                   \* asserts expression e can be proved from the definitions and assumptions and rules of TLA+
10.    ====                        \* ends TLA+ module (everything after that is ignored).    Min 4 '=' at start of line

11.  \* Comment                    \* to end of line
12.  (* Comments *)               \* multi-line comment or boxed comment
13.  ------------------------      \* horizontal separator line

BLUE sets, RED booleans, YELLOW integers, GREEN functions/records, PURPLE sequences/tuples, BLACK any/other
```

Structure of our TLA+ specifications

```
---- MODULE name ----          \* .tla file
EXTENDS   m1, ..., mN          \* can also use plural keywords
CONSTANTS c1, ..., cN          \* defined in .cfg file
VARIABLES v1, ..., vN
Vars == << v1, ..., vN >>
Type == v1_formula /\ ... /\ vN_formula
-----------------------------  \* Spec. for state machine
Init == formula                \* Initial state
Def1 == formula                \* Definitions
Def2 == formula
Action1 == action_formula      \* Subactions of Next
Action2 == action_formula
Next == Action1 \/ ... \/ ActionN   \* Next state
-----------------------------
Fair == fairness_formula /\ ... /\ fairness_formula
Spec == Init /\ [][Next]_Vars /\ Fair
-----------------------------
NotDeadlock == []<>(ENABLED Next)   \* Properties
Typed == []Type
====
```

```
Configuration File   \* .cfg file

   SPECIFICATION Spec
   PROPERTY      NotDeadlock  Typed

TLC checks the absence of deadlock by default so we
will omit the NotDeadlock property from our .tla and
.cfg files.

If we have no Fairness definition then we can use the
following configuration statements instead:

   SPECIFICATION Spec
   INVARIANT     Type   \* mapped to PROPERTY []Type

or:
   INVARIANT Type     \* mapped to PROPERTY []Type
   INIT Init          \* assumes specification is
   NEXT Next          \* Init /\ [][Next]_declaredvars

Note: None of the names Vars, Type, Init, Next,
Deadlock, Typed, Fair, Spec are special. They're just
what we've decided to use in our examples.
```

Some LTL Equivalences

- Distribution:
  - $\Box(p \wedge q) \equiv \Box p \wedge \Box q$ $\quad$ $\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$ $\quad$ $p\ \mathcal{U}\ (q \vee r) \equiv (p\mathcal{U}\ q) \vee (p\mathcal{U}\ r)$
  - $\Diamond(p \vee q) \equiv \Diamond p \vee \Diamond q$ $\quad$ $\bigcirc(p \wedge q) \equiv \bigcirc p \wedge \bigcirc q$ $\quad$ $(p \wedge q)\ \mathcal{U}\ r \equiv (p\mathcal{U}\ r) \wedge (q\mathcal{U}\ r)$
- Duals:
  - $\neg\Diamond p \equiv \Box\neg p$ $\quad$ $\neg\bigcirc p \equiv \bigcirc\neg p$
  - $\neg\Box p \equiv \Diamond\neg p$
- Idempotency:
  - $\Box p \equiv \Box\Box p$ $\quad$ $p\ \mathcal{U}\ (q\ \mathcal{U}\ r) \equiv (p\ \mathcal{U}\ q)\ \mathcal{U}\ r) \equiv p\ \mathcal{U}\ r$
  - $\Diamond p \equiv \Diamond\Diamond p$
- Absorption:
  - $\Box\Diamond p \equiv \Diamond\Box\Diamond p$
  - $\Diamond\Box p \equiv \Box\Diamond\Box p$
- Other: $\quad$ $\texttt{TRUE}\ \mathcal{U}\ p \equiv \Diamond p$

Example – Abstract Alternating Bit Protocol p229

MODULE *AbstractAltBit*

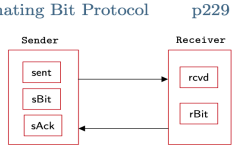
EXTENDS *Naturals*
CONSTANTS *Data*
VARIABLES *sBit, sAck, rBit, sent, rcvd*
A_Vars $\triangleq$ *(sBit, sAck, rBit, sent, rcvd)*

A_Init $\triangleq$ $\wedge sBit \in \{0,1\}$ $\wedge sent \in Data$
$\wedge sAck = sBit$ $\wedge rcvd \in Data$
$\wedge rBit = sBit$

A_SendMsg(d) $\triangleq$ $\wedge sAck = sBit$
$\wedge sent' = d$
$\wedge sBit' = 1 - sBit$
$\wedge$ UNCHANGED $\langle sAck, rBit, rcvd\rangle$

A_ReceiveMsg $\triangleq$ $\wedge rBit \neq sBit$
$\wedge rBit' = sBit$
$\wedge rcvd' = sent$
$\wedge$ UNCHANGED $\langle sBit, sAck, sent\rangle$

A_ReceiveAck $\triangleq$ $\wedge rBit \neq sAck$
$\wedge sAck' = rBit$
$\wedge$ UNCHANGED $\langle sBit, rBit, sent, rcvd\rangle$



A_Next $\triangleq$ $\vee \exists d \in Data : A\_SendMsg(d)$
$\vee A\_ReceiveMsg \vee A\_ReceiveAck$

A_Fair $\triangleq$ $\wedge$ WF$_{A_Vars}$*(A_ReceiveMsg)*
$\wedge$ WF$_{A_Vars}$*(A_ReceiveAck)*

A_Spec $\triangleq$ *A_Init* $\wedge \Box[A\_Next]_{A\_Vars} \wedge A\_Fair$

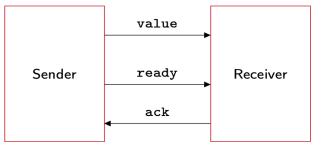
A_Typed $\triangleq$ $\Box$*Type*

Some LTL Patterns

Invariance	$\Box p$	always p
Guarantee	$\Diamond p$	eventually p
Response	$p \Rightarrow \Diamond q$	p implies eventually q
Progress	$\Box\Diamond p$	always eventually p (may need to wait)
Stability	$\Diamond\Box p$	eventually always p
Correlation	$\Diamond p \Rightarrow \Diamond q$	eventually p implies eventually q

```
MODULE Async
EXTENDS   Naturals
CONSTANT  Data
VARIABLES value, ready, ack
Vars ≜ ⟨value, ready, ack⟩
Type ≜ ∧ value ∈ Data
       ∧ ready ∈ {0, 1}
       ∧ ack   ∈ {0, 1}
-----------------------------------------
Init ≜ ∧ value ∈ Data
       ∧ ready ∈ {0, 1}
       ∧ ack = ready
Send ≜ ∧ ready = ack
       ∧ value' ∈ Data
       ∧ ready' = 1 − ready
       ∧ UNCHANGED ack
Receive ≜ ∧ ready ≠ ack
          ∧ ack' = 1 − ack
          ∧ UNCHANGED ⟨value, ready⟩
Next ≜ Send ∨ Receive
Spec ≜ Init ∧ □[Next]_Vars
-----------------------------------------
Typed ≜ □Type
```

Example: Async p23



- Each server's *currentTerm* monotonically increases

$$\forall i \in Server : currentTerm[i] \leq currentTerm'[i]$$

- Election Safety Property. There is at most one leader per term. Note: *elections* is a "history variable" that keeps track of elections and is used in the TLAPS proof.

$$\forall e, f \in elections : e.eterm = f.eterm \Rightarrow e.eleader = f.eleader$$

- Leader Append-Only Property. Leaders logs grow monotonically during their term.

$$\forall e \in elections : currentTerm[e.leader] = e.term \Rightarrow \forall\ index \in 1..Len(log[e.leader]) : log'[e.leader][index] = log[e.leader][index]$$

- State Machine Safety Property. Servers only apply entries that are *committed* in their current term.

$$\forall i \in Server : \wedge commitIndex[i] \leq Len(log[i]) \wedge \forall \langle index, term\rangle \in log[i] : index \leq commitIndex[i] \Rightarrow \langle index, term\rangle \in committed(currentTerm[i])$$

- Committed. An entry <*index, term*> is committed at term *t* if it present in every leader's log following term *t*.

$$committed(t) \triangleq \{\langle index, term\rangle : \forall\ election \in elections : election.eterm > t \Rightarrow \langle index, term\rangle \in election.elog\}$$

Temporal Logic Operators. **Atom a** - a true in current moment/state. **Next** $\circ p$ - p is true in next moment/state. Next not available in TLA+. **Always** $\Box p$ - true now and in all future moments/states. **Eventually** $\Diamond p$ - p is true now or in some future moment/state. Always Eventually $\Box\Diamond p$ - p is true infinitely often (infinitely many times). p is repeatedly true - moments or states where p isn't true. Hungry, Redlight. **Eventually Always** $\Diamond\Box p$ - p will be true from some state. There will be a moment/state where p will be true and will remain true. p will become continuously true. Terminated, for example.
Fairness. Constraints we assume the scheduler of the system will enforce - to fairly select the next enabled action to execute. Without fairness constraints its possible that some liveness properties may never progress. E.g. if an enabled action is never executed, so are used to guarantee that parts of the system make progress. Expressed in temporal logic. Weak Fairness - $\Diamond\Box A => \Box\Diamond A$. If an action A is continuously (Eventually Alw) enabled then it will be executed infinitely often. Strong Fairness $\Box\Diamond A => \Box\Diamond A$. If action A is repeatedly (Alw Eventually) enabled then it will be executed infinitely often when in enabled state. Repeatedly - times when A not enabled. Absolute Fairness $\Box\Diamond A$ - A will be executed infinitely often whether enabled or not. Properties true under WF also true under SF and AF.
TLA+ Fairness. WF constraint $WF_v(A) \triangleq \Diamond\Box(ENABLED < A >_v) => \Box\Diamond < A >_v$ - if non-stuttering action A is continuously enabled then it will be executed repeatedly. A must eventually occur if A is continuously enabled. Useful for preventing unfairness due to worst-case interleaving/scheduling of actions. SF Constraint $SF_v(A) \triangleq \Box\Diamond(ENABLED < A >_v) => \Box\Diamond < A >_v$. If non-stuttering action A is repeatedly enabled then it will be executed repeatedly (inf often) when in enabled state. A must eventually occur if A is repeatedly enabled. Useful for handling contention. $Typed \triangleq \Box Type$ - not type.
Safety. Nothing bad ever happens. Assert what may happen. If a safety property is violated by a behaviour it will have been violated in a particular step from which we can produce a counterexample that shows an error trace to the violating step.
Liveness. Something good eventually happens. Liveness properties assert what must happen - cannot be violated by a particular step - ie rest of behaviour can always make the liveness property true. Check for inf behaviours - clock stopped clicking or message never delivered. Requries temporal logic but ad-hoc temporal formulas can be error-prone. Conjunction of $WF_v(A)$ and $SF_v(A)$ formulas. Next constraints what steps may occur while Fairness only defines what steps must eventually occur. Fairness should not constrain Init or Next (machine closed spec). Typically, actions used in Fairness formula are subactions of Next that define how behaviours keep executing Next actions when Next is enabled.