

An algorithm for replicating multi-threaded applications as done in replicated Popcorn. The application is made deterministic through the use of logical time. Any inter-thread synchronization operation must be protected by calls to *EnterSync* and *ExitSync*. Reads of the socket *API* are modeled by *EnterRead*. The scheduler processes (one per kernel) makes sure that the different copies of the application are consistent.

EXTENDS *Naturals, Sequences, Integers, Library*

CONSTANTS *Pid, MaxTime, Kernel, SchedulerPID, Requests*

ASSUME *SchedulerPID* \notin *Pid*

InitLogTime $\triangleq 1$

LogTime \triangleq *InitLogTime* .. *MaxTime* The set of logical time values

Processes are of the form $\langle k, pid \rangle$, where *k* is the kernel the process is running on.

P \triangleq *Kernel* \times *Pid*

Primary \triangleq CHOOSE *k* \in *Kernel* : TRUE

Ker(*p*) \triangleq *p*[1]

PID(*p*) \triangleq *p*[2]

OnKernel(*kernel*) \triangleq {*kernel*} \times *Pid*

Logical time comparison, using *PIDs* to break ties.

Less(*p*, *tp*, *q*, *tq*) \triangleq

$tp < tq \vee (tp = tq \wedge PID(p) \leq PID(q))$

The sequence of *TCP* packets that will be received. No duplicates allowed (therefore the set *TcpData* must be big enough) so that any misordering of the threads will lead to a different data read. For *TCPMultiStream*, each stream has different data.

StreamLength $\triangleq 3$

TcpData $\triangleq 1 \dots StreamLength * Requests$

TcpStream $\triangleq [i \in 1 \dots StreamLength \mapsto i]$

TcpMultiStream $\triangleq [r \in 1 \dots Requests \mapsto$
 $([i \in 1 \dots StreamLength \mapsto i + (StreamLength * (r - 1))])]$

ASSUME *NoDup*(*TcpStream*)

ASSUME *Len*(*TcpStream*) = *StreamLength*

Shifts a sequence by 1: *Shift*($\langle 1, 2, 3 \rangle$) = $\langle 2, 3 \rangle$ and *Shift*($\langle \rangle$) = $\langle \rangle$.

Shift(*s*) \triangleq

IF *Len*(*s*) > 1

THEN $[i \in 1 \dots (Len(s) - 1) \mapsto s[i + 1]]$

ELSE $\langle \rangle$

Shiftn(*s*) \triangleq

IF *Len*(*s*) > 1

```

THEN  $[i \in 1 \dots (Len(s) - 1) \mapsto s[i + 1]]$ 
ELSE  $\langle -1 \rangle$ 

```

The algorithm *ReadAppend* models a set of worker threads being scheduled by the deterministic scheduler and executing the following code.

```

Code of worker  $w$ : While(true){
   $x = read(socket)$ ;
  append( $\langle w, x \rangle$ , file);
}

```

Variables:

The variable *bumps* records all logical time bumps executed by the primary in order for the secondaries to do the same, *i.e.* the initial logical time, the new logical time, and the value read from the tcp buffer. $\langle t1, t2, d \rangle \in bumps[pid]$ means that the primary bumped process *pid* from logical time *t1* to *t2* and delivered the data *d*. Note that the scheduler set *bumps*[*pid*] to a value that depends on the logical time of the processes on all replicas, and this value is then immediately available to all replicas. A more detailed model would instead include a distributed implementation of the choice of the logical time to bump the process to.

reads[*p*] stores the last value read by *p* from the socket.

tcpBuff[*k*] represents the state of the tcp buffer on kernel *k*. Each time a process reads from the buffer, the buffer shrinks by 1.

Definitions:

Bumped(kernel) is the set of processes running on the kernel “kernel” which are waiting to execute a “bump” decided by the primary.

If $p \in Bumped(Ker(p))$ then *BumpedTo*(*p*) is the logical time to which *p* should be bumped to.

If $p \in WaitingSync(Ker(p))$ then *IsNextProc*(*kernel*, *p*) is true iff *p* is the process to be scheduled next, that is: (1) *p* has the lowest *ltime* among running and waiting-sync processes and (2) if *q* is on the same kernel and *q* is waiting for a read and the primary has already decided to which logical time *tq* to bump *q*, then *ltime*[*p*] must be less than *tq*.

BumpTo is the logical time to which to bump a process that needs bumping. It is some logical time greater than all the logical times reached by any process on any kernel.

--algorithm *ReadAppend*{

variables

```

     $status = [p \in P \mapsto \text{“running”}]$ ,
     $ltime = [p \in P \mapsto InitLogTime]$ ,
     $file = [k \in Kernel \mapsto \langle \rangle]$ ,
     $bumps = [p \in Pid \mapsto \{\}]$ ,
     $reads = [p \in P \mapsto -1]$ ,
     $tcpBuff = [k \in Kernel \mapsto TcpMultiStream]$ ,
    Queue for accepted connections
     $socketQueue = [k \in Kernel \mapsto \langle \rangle]$ ,
    Queue for unhandled connections
     $requestQueue = [k \in Kernel \mapsto [r \in 1 \dots Requests \mapsto r]]$ ,
    The socket that is handled by a process
     $handledSocket = [p \in P \mapsto -1]$ ,

```

$connections = [k \in Kernel \mapsto \langle \rangle]$

```

define {
   $Run(p) \triangleq status[p] = \text{"running"}$ 
   $Running(kernel) \triangleq \{p \in OnKernel(kernel) : Run(p)\}$ 
   $WaitingSync(kernel) \triangleq$ 
     $\{p \in OnKernel(kernel) : status[p] = \text{"waiting sync"}\}$ 
   $WaitingRead(kernel) \triangleq$ 
     $\{p \in OnKernel(kernel) : status[p] = \text{"waiting read"}\}$ 
   $Bumped(kernel) \triangleq \{p \in OnKernel(kernel) :$ 
     $\wedge status[p] = \text{"waiting read"}$ 
     $\wedge \exists t \in LogTime : \exists d \in TcpData : \langle ltime[p], t, d \rangle \in bumps[PID(p)]\}$ 
   $BumpedTo(p) \triangleq$ 
     $CHOOSE t \in LogTime : \exists d \in TcpData : \langle ltime[p], t, d \rangle \in bumps[PID(p)]$ 
   $BumpData(p) \triangleq$ 
     $CHOOSE d \in TcpData : \exists t \in LogTime : \langle ltime[p], t, d \rangle \in bumps[PID(p)]$ 
   $IsNextProc(kernel, p) \triangleq$ 
     $\wedge \forall q \in Running(kernel) \cup WaitingSync(kernel) :$ 
     $q \neq p \Rightarrow Less(p, ltime[p], q, ltime[q])$ 
     $\wedge \forall q \in Bumped(kernel) : Less(p, ltime[p], q, BumpedTo(q))$ 
   $BumpTo \triangleq CHOOSE i \in LogTime : \forall p \in P : ltime[p] < i$ 
}

```

```

macro EnterRead(p){
   $status[p] := \text{"waiting read"} ;$ 
}
macro EnterSync(p){
   $status[p] := \text{"waiting sync"} ;$ 
}
macro ExitSync(p){
   $ltime[p] := ltime[p] + 1 ;$ 
}

```

Processes consume a connection

```

process ( $worker \in P$ ){
  ww1: while (TRUE){
    EnterSync(self) ;
  ww2: await Run(self) ;
  ww3: if ( $Len(requestQueue[Ker(self)]) > 0$ ){
     $handledSocket[self] := requestQueue[Ker(self)][1] ;$ 
     $requestQueue[Ker(self)] := Shift(requestQueue[Ker(self)]) ;$ 
  } ;
  ww4: ExitSync(self) ;
  ww5: if ( $handledSocket[self] \neq -1$ ){
    ww9: while ( $Len(tcpBuff[Ker(self)][handledSocket[self]]) > 0$ ){
      EnterRead(self) ;
    }
  }
}

```

