B
Remote Procedure Calls

This Appendix presents the operational semantics of the remote procedure call defined in Chapter 4.

B.1 The definition of `ShareRPC`

```
ShareRPC :: (RPCDef in out *(PState .l .p)) .l *(IOState .m .p)
         -> (RPCId  in out,*IOState .m .p)
ShareRPC {rpcFunction=f,rpcInitIO=initIO} l io=:{ioGroup=procs}
=   (rpcId, {io & ioGroup=[process:procs]})
where
    process = {ppLocal   = l,
               pInitIO   = [initRPC:initIO],
               pDevices  = []}
    (rpcId, initRPC) = openRPC f
```

B.2 The definition of `SendRPC`

```
SendRPC :: (RPCId in out) in *(PState .l .p) -> (SuccessOrFail out, *PState .l .p)
SendRPC rpcId in ps=:{procId=id}
|   not (isSuccessful oid) = (Fail,psUnblockThisProcess (contextSwitchIn (id,gs1)))
|   otherwise              = (out, psUnblockThisProcess (contextSwitchIn (id,gs3)))
where
    ps1      = psBlockThisProcess rpcId ps
    gs       = contextSwitchOut ps1
    (oid,gs1)= gsLocateRPC rpcId gs
    id1      = getSuccessValue oid
    gs2      = while (gsCond id) gsEval gs1
    (rs,gs3) = gsGetRuntimeState id gs2
    out      = rsGetGrant rs

    while :: (.x -> (Bool,.x)) (.x -> .x) .x -> .x
    while cond do x
    |     done      = x1
    |     otherwise = while cond do (do x1)
    where (done,x1) = cond x

gsCond :: ProcId GState -> (Bool,GState)
gsCond pId gs
|   not (rsIsGranted rs) = (True, gs1)
|   otherwise            = gsGetRandomBool gs1
where
    (rs,gs1) = gsGetRuntimeState pId gs

gsEval :: GState -> GState
gsEval gs
|   rsIsRunning rs                          = contextSwitchOut ps1`
|   rsIsGranted rs                          = contextSwitchOut ps1
|   not (isSuccessful oid)                  = contextSwitchOut ps2`
|   rsIsBlocked rs` && not (rsIsGranted rs`)  = contextSwitchOut ps3
|   otherwise                               = contextSwitchOut ps5
where
    (id,ps)  = switchToSomeProcess gs
    (rs,ps1) = psGetRuntimeState ps
    ps1`     = psEval ps1
    req      = rsGetRPCReq rs
    rpcId    = req.rpcRReceiver
    (oid,ps2)= psLocateRPC rpcId ps1
    id1      = getSuccessValue oid
    req1     = {req & rpcROut     = Fail,
                      rpcRGranted = True}
    ps2`     = psSetRuntimeState (Blocked req1) ps2
    (rs`,ps3)= psGetOtherRuntimeState id1 ps2
```

```
    (out,ps4)= psCommunicate req id id1 ps3
    req2     = {req & rpcROut     = Success out,
                      rpcRGranted = True}
    ps5      = psSetRuntimeState (Blocked req2) ps4

psCommunicate :: (RPCReq in out) ProcId ProcId (PState .l .p) -> (out, PState .l .p)
psCommunicate req psId pId ps
=   (out, ps4)
where
    gs       = contextSwitchOut ps
    ps1      = contextSwitchIn (pId,gs)
    (f,ps2)  = psGetRPC req.rpcRReceiver ps1
    (out,ps3)= f (req.rpcRIn,ps2)
    gs1      = contextSwitchOut ps3
    ps4      = contextSwitchIn psId gs1
```

B.3 Operations on `SuccessOrFail`

```
::  SuccessOrFail x = Success x | Fail

isSuccessful    :: (SuccessOrFail x) -> Bool
isSuccessful    (Success _) = True
isSuccessful    _           = False

getSuccessValue :: (SuccessOrFail x) -> x
getSuccessValue (Success x) = x
```

B.4 Operations on GState

```
gsHasProcess :: ProcId GState -> Bool
gsHasProcess pId gs = process pId is in gs

gsLocateRPC :: (RPCId in out) GState -> (SuccessOrFail ProcId, GState)
gsLocateRPC rpcId gs = the procid of the RPC process

gsGetRuntimeState :: ProcId GState -> (RuntimeState, GState)
gsGetRuntimeState pId gs
=   (runtime, gs1)
where
    ps            = contextSwitchIn (pId,gs)
    (runtime, ps1)= psGetRuntimeState ps
    gs1           = contextSwitchOut ps1

gsGetRandomBool :: GState -> (Bool, GState)
gsGetRandomBool gs = a random boolean value
```

B.5 Operations on `PState`

```
psBlockThisProcess :: (RPCId in out) in (PState .l .p) -> PState .l .p
psBlockThisProcess rpcId in ps = {ps & pRuntime=Blocked rpcReq}
where
    rpcReq = rpcrNewRPCReq rpcId in
```

```
psUnblockThisProcess :: (PState .l .p) -> PState .l .p
psUnblockThisProcess ps = {ps & pRuntime=Running}

psGetRuntimeState :: (PState .l .p) -> (RuntimeState, PState .l .p)
psGetRuntimeState ps = (ps.pRuntime, ps)

psSetRuntimeState :: RuntimeState (PState .l .p) -> PState .l .p
psSetRuntimeState rs ps = {ps & pRuntime=rs}

psGetRPC :: (RPCId in out) (PState .l .p)
         -> (RPCFunction in out (PState .l .p), PState .l .p)
psGetRPC rpcId ps = yield the RPC function of the RPC process
```

B.6 Operations on `RuntimeState`

```
::  RuntimeState      =   Blocked (RPCReq Void Void)
                      |   Running
::  RPCReq E.in E.out =   { rpcRReceiver:: RPCId in out,
                            rpcRGranted :: Bool,
                            rpcRIn      :: in,
                            rpcROut     :: SuccessOrFail out   }

rsIsBlocked :: RuntimeState -> Bool
rsIsBlocked (Blocked _) = True
rsIsBlocked _           = False

rsIsGranted :: RuntimeState -> Bool
rsIsGranted (Blocked rpcReq) = rpcReq.rpcRGranted
rsIsGranted _                = False

rsGetRPCReq :: RuntimeState -> RPCReq Void Void
rsGetRPCReq (Blocked rpcReq) = rpcReq

rsGetGrant  :: RuntimeState (RPCId in out) -> SuccessOrFail out
rsGetGrant  (Blocked rpcReq) rpcId
|   rpcId == rpcReq.rpcRReceiver = rpcReq.rpcROut
|   otherwise                    = Fail
rsGetGrant _ _                   = Fail

rsIsRunning :: RuntimeState -> Bool
rsIsRunning Running = True
rsIsRunning _       = False


rpcrNewRPCReq :: (RPCId in out) in -> RPCReq Void Void
rpcrNewRPCReq pId rpcId in
=   {rpcRReceiver= rpcId,
     rpcRGranted = False,
     rpcRIn      = in,
     rpcROut     = Fail    }
```