MODULE *PortAllocator*

The port allocator updates an endpoint's ports field from its spec. For host-mode ports, it just copies them over. For *ingress*-mode ports, it checks that the requested port number is free (if given), or allocates an unused one from the dynamic range (if not).

Places where I deviated from the Go code to make the checks pass are marked with *XXX*. In summary:

# Reusing a previous assignment when we need it elsewhere

Example:

1. [*name* = "foo", dynamic] (initial configuration)
2. [*name* = "foo", dynamic], [*name* = "bar", published = 30000] (updated)

Here we will reject the update because we try to reuse port 30000 for "foo", even though we now need it for "bar".

# Duplicate similar ports

Example:

1. [*name* = "foo", dynamic]
2. [*name* = "foo", dynamic], [*name* = "foo", dynamic]

We will try to use port number 30000 for both ports and reject the allocation.

# Shadowing an existing host port

The allocator ignores host ports completely. It may allocate a dynamic *ingress* port using a network address that is already in use on some host. In this case, the original service becomes unreachable.

# Accepting a host port that is hidden

The allocator may accept a host-port allocation that it knows will be unreachable on any host because that address is already in use as an *ingress* port.

EXTENDS *Sequences*, *Integers*, *TLC*, *FiniteSets* Some libraries we use

$Range(S) \triangleq \{S[i] : i \in \text{DOMAIN } S\}$ Generic helper function

The set of protocols we support (*e.g.* {"*tcp*", "*udp*", "*sctp*"}). We assume that every protocol has the same set of static and dynamic ports.

CONSTANT *Protocol*

A host-mode port is available only via the host that ends up running the workload. Two different services can use the same port, as long as they run on different hosts.

$host \triangleq$ "host"

Ingress-mode ports are available via any node in the cluster. Connections will be forwarded to a host running the service.

$ingress \triangleq$ "ingress"

$Mode \triangleq \{ingress, host\}$

Port numbers that can be assigned by the allocator when the user requests a dynamic port number. The code currently uses 30000 . . 32767.

CONSTANT *DynamicPortNumber*

Non-dynamic port numbers.

CONSTANT *StaticPortNumber*

All port numbers. *SwarmKit* uses 1 .. 65535. We mainly define it this may to make it easy to make dynamic and static numbers symmetry sets in the model checker.

$PortNumber \triangleq DynamicPortNumber \cup StaticPortNumber$

A special value for the *published_port* field to indicate that the allocator should select the port. *SwarmKit* uses 0 for this.

$dynamic \triangleq \text{CHOOSE } x : x \notin PortNumber$

The type of endpoint *IDs* (*e.g.* STRING).

CONSTANT *EndpointID*

The type of port names (*e.g.* STRING).

CONSTANT *Name*

The maximum number of ports in a specification.

CONSTANT *MaxPorts*

The requirements for a port, provided by the user. The user can specify a port number directly (including any number in *DynamicPortNumber*), or can specify *dynamic* to have the system allocate it.

The real structure also includes *target_port* , which is the port inside the container. For the model, we can consider this as part of *name* (it just makes the port more unique).

$$
\begin{array}{l}
PortSpec \triangleq [\\
\quad \begin{array}{ll}
mode & : Mode, \\
name & : Name, \\
protocol & : Protocol, \\
published\_port & : StaticPortNumber \cup DynamicPortNumber \cup \{dynamic\}
\end{array} \\
]
\end{array}
$$

A configured port, after the allocator has done its job. Note: The *SwarmKit* code uses a single Go type for this and *PortSpec*.

$PortConfig \triangleq$

Either an *ingress* port:

$$
\begin{array}{l}
[\\
\quad \begin{array}{ll}
mode & : \{ingress\}, \\
name & : Name, \\
protocol & : Protocol, \\
published\_port & : PortNumber
\end{array}
\end{array}
$$

The port is now allocated

$$
]
$$

$\cup$ or a *host* port:

$$
\begin{array}{l}
[\\
\quad \begin{array}{ll}
mode & : \{host\}, \\
name & : Name,
\end{array}
\end{array}
$$

$\quad \mathit{protocol} \quad\quad\;\; : \mathit{Protocol},$

$\quad \mathit{published\_port} : \mathit{PortNumber} \cup \{\mathit{dynamic}\}$ Can still be unassigned

$\;]$

Two *PortConfig*/*PortSpec* values are "mostly equal" if they differ only in their *published_port*.

$\mathit{PortsMostlyEqual}(a,\, b) \;\triangleq$

$\quad \text{LET } \mathit{ignorePP}(x) \;\triangleq\; [f \in \text{DOMAIN } x \setminus \{\text{"published\_port"}\} \mapsto x[f]]$

$\quad \text{IN} \quad \mathit{ignorePP}(a) = \mathit{ignorePP}(b)$

A network address is a protocol and port-number pair.

$\mathit{Address} \;\triangleq\; \mathit{Protocol} \times \mathit{PortNumber}$

The network address of a *SwarmKit* port.

$\mathit{Addr}(\mathit{port}) \;\triangleq$

$\quad \langle \mathit{port.protocol},\, \mathit{port.published\_port} \rangle$

A finite sequence of maximum length *max* . Useful for model checking.

$\mathit{FiniteSeq}(S,\, \mathit{max}) \;\triangleq$

$\quad \text{UNION } \{[1 \mathrel{..} n \to S] : n \in 0 \mathrel{..} \mathit{max}\}$

An endpoint specification is just a list of port specifications.

$\mathit{EndpointSpec} \;\triangleq\; \mathit{FiniteSeq}(\mathit{PortSpec},\, \mathit{MaxPorts})$

An endpoint object records the currently allocated ports and the original specification that led to this assignment.

$\mathit{Endpoint} \;\triangleq\; [$

$\quad \mathit{spec} \;\;: \mathit{EndpointSpec},$

$\quad \mathit{ports} : \mathit{Seq}(\mathit{PortConfig})$

$]$

nullEndpoint is used to represent an endpoint that does not yet exist.

$\mathit{nullEndpoint} \;\triangleq\; [$

$\quad \mathit{spec} \;\;\mapsto \langle\rangle,$

$\quad \mathit{ports} \mapsto \langle\rangle$

$]$

The allocator returns a proposal for updating the state, rather than doing it immediately. Note that the port allocator's proposal is only valid until the next time an allocation is requested.

The *allocate* field is not really needed here, as we can generate it easily (it's just the set of *ingress* addresses in *ports*). *deallocate* wouldn't be needed if the commit operation took the old configuration as an argument, as it's just the *ingress* addresses in that. However, the Go code includes these fields, so we do too.

$\mathit{Proposal} \;\triangleq\; [$

$\quad \mathit{deallocate} : \text{SUBSET } \mathit{Address},$ Addresses to remove from *allocated*

$\quad \mathit{allocate} \quad : \text{SUBSET } \mathit{Address},$ Addresses to add to *allocated* (after deallocation)

$\quad \mathit{ports} \qquad : \mathit{Seq}(\mathit{PortConfig})$ The new value for *endpoint.ports*

$]$

```
 The allocator

 The set of allocated ingress addresses.
VARIABLE allocated

 The smallest item in a non-empty set.
MinElement(S) ≜ CHOOSE x ∈ S : ∀ y ∈ S : x ≤ y

 Return an updated version of spec in which dynamic ingress ports have been updated to copy
 the existing configution, where possible.
RecoverExistingPorts(endpoint, spec) ≜
  LET The current configuration
      oldPorts ≜ endpoint.ports
       Indexes in the list of ports for which we need to choose a port number:
      dynamics ≜ {i ∈ DOMAIN spec :
                        ∧ spec[i].mode = ingress
                        ∧ spec[i].published_port = dynamic}
       Ingress ports for which the user specified the port:
      forcedPorts ≜ {p ∈ Range(spec) :
                        ∧ p.mode = ingress
                        ∧ p.published_port ≠ dynamic}
       The (ingress) addresses the user specified manually:
      forcedAddrs ≜ {Addr(p) : p ∈ forcedPorts}
       The recovered port number for port i :
      Recover(i) ≜
        LET The port specification from the user:
            s ≜ spec[i]
             The currently configured port(s) that are like s :
            olds ≜ {j ∈ DOMAIN oldPorts :
                      ∧ ∃ k ∈ DOMAIN endpoint.spec :
                           ∧ endpoint.spec[k] = s        Spec hasn't changed
                      ∧ PortsMostlyEqual(oldPorts[j], s)
                       We're not forced to use this for something else:
                       XXX: does SwarmKit do this?
                       ∧ Addr(oldPorts[j]) ∉ forcedAddrs
                    }
             Whether s is similar to a previous dynamic port in the spec list:

             XXX: Looks like the SwarmKit code doesn't do this check.
            duplicate ≜ ∃ j ∈ 1 . . (i − 1) :   An earlier similar port in the list
                          ∧ j ∈ dynamics        needed a dynamic assignment too
                          ∧ PortsMostlyEqual(spec[i], spec[j])
      IN
      IF  ∨ olds = {}    If we haven't already got anything like s
          ∨ duplicate    Or we already used it
          THEN $dynamic$   Then don't update $s$ − it's still $dynamic$
          ELSE
             Use the first of the candidates for $published\_port$ :
            $oldPorts[MinElement(olds)].published\_port$
        The updates to apply:
      $recovered \triangleq [i \in dynamics \mapsto [spec[i]$ EXCEPT $!.published\_port = Recover(i)]]$
  IN
  $recovered$ @@ $spec$   Combine updates with other entries
```

$Allocate(endpoint, spec)$ returns a set of possible proposals to update $endpoint$ to $spec$.

The real system will only return a single proposal. For cases where this function returns {} a real implementation must reject the request. For cases where it returns a non-empty set, a real implementation must return one of the elements as its proposal.

```
$Allocate(endpoint, specFromUser) \triangleq$
  LET   Step 1 : $Recover$ dynamic ports from old configuration
      $spec \triangleq RecoverExistingPorts(endpoint, specFromUser)$
       Step 2 : Reject bad user requests due to static assignments
       All the $ingress$ ports in the existing configuration:
      $oldIngressPorts \triangleq \{p \in Range(endpoint.ports) : p.mode = ingress\}$
       Addresses currently in use by $endpoint$ :
      $deallocate \triangleq \{Addr(p) : p \in oldIngressPorts\}$
       Addresses used by other $endpoints$:
      $addrsForOthers \triangleq allocated \setminus deallocate$
       Did the user request a port that another endpoint is using?
      $alreadyInUse \triangleq \exists p \in Range(spec) :$
                            $\land p.mode = ingress$
                            $\land p.published\_port \neq dynamic$
                            $\land Addr(p) \in addrsForOthers$
       Did the user specify the same static ($ingress$) address twice?
      $haveForcedDuplicates \triangleq$
        $\exists i, j \in$ DOMAIN $spec$ :
          $\land i \neq j$
          $\land$ LET $si \triangleq spec[i]$
                  $sj \triangleq spec[j]$
             IN
             $\land si.mode = ingress \land si.published\_port \neq dynamic$
             $\land sj.mode = ingress \land sj.published\_port \neq dynamic$
             $\land Addr(si) = Addr(sj)$
   IN
   IF $alreadyInUse \lor haveForcedDuplicates$ THEN $\{\}$  Reject
    ELSE
     Step 3 : Assign dynamic ports
```

There are various ways of assigning the ports. *e.g.* picking the lowest free port, starting the search from the last allocated number, checking already-free ports first and then using ports from *deallocate* only as a last resort. We'll avoid over-specifying by allowing any behaviour here.

```
LET
    Ingress ports that still need to be assigned:
   portsNeeded ≜ {i ∈ DOMAIN spec :
                        ∧ spec[i].mode = ingress
                        ∧ spec[i].published_port = dynamic}
  Possible ways of allocating them. Each element of this set is a mapping from a port index
  to a port number in the dynamic range.
   allocs ≜
     {alloc ∈ [portsNeeded → DynamicPortNumber] :
        Check that alloc is reasonable:
       LET NA(i) ≜  The proposed network address of i
         IF i ∈ DOMAIN alloc THEN ⟨spec[i].protocol, alloc[i]⟩
          ELSE Addr(spec[i])
       IN
       ∀ i ∈ DOMAIN alloc :  For each dynamic ingress port:
           No other endpoint is using this address:
          ∧ NA(i) ∉ addrsForOthers
           We're not already trying to allocate this address:
          ∧ ∀ j ∈ DOMAIN spec \ {i} :
                ∨ spec[j].mode = host
                ∨ NA(i) ≠ NA(j)
     }
    Create a proposal object from an allocation mapping:
   Result(alloc) ≜
    LET
      ports ≜ [i ∈ DOMAIN alloc ↦
                 [spec[i] EXCEPT !.published_port = alloc[i]]
               ] @@ spec
      ingressPorts ≜ {p ∈ Range(ports) : p.mode = ingress}
    IN
    [
      deallocate ↦ deallocate,
      allocate ↦ {Addr(p) : p ∈ ingressPorts},
      ports ↦ ports
    ]
IN
{Result(x) : x ∈ allocs}

The result of applying prop to the current allocations.
Apply(prop) ≜
  (allocated \ prop.deallocate) ∪ prop.allocate
```

The test system (allocator + user)

The set of active *endpoints* (the allocator doesn't look at this)

```
VARIABLES endpoints

vars ≜ ⟨allocated, endpoints⟩

 The user creates a new endpoint
NewEndpoint ≜
  ∃ s ∈ EndpointSpec :                                 s is the new spec
  ∃ id ∈ EndpointID \ DOMAIN endpoints :               id is an unused endpoint ID
  ∃ prop ∈ Allocate(nullEndpoint, s) :                 prop is a proposal from the allocator
  LET e ≜ [spec ↦ s, ports ↦ prop.ports]               e is the new endpoint
  IN        Update the store:
   ∧ endpoints′ = id :> e @@ endpoints                 Add e to endpoints
   ∧ allocated′  = Apply(prop)                         Tell the allocator to commit

 The user updates an existing endpoint
UpdateEndpoint ≜
  ∃ s ∈ EndpointSpec :                                 s is the new spec
  ∃ id ∈ DOMAIN endpoints :                            id is an existing endpoint
  ∃ prop ∈ Allocate(endpoints[id], s) :                prop is a proposal from the allocator
  LET e ≜ [spec ↦ s, ports ↦ prop.ports]               e is the new endpoint
  IN
   ∧ endpoints′ = [endpoints EXCEPT ![id] = e]
   ∧ allocated′  = Apply(prop)

 Remove an existing endpoint
RemoveEndpoint ≜
  ∃ id ∈ DOMAIN endpoints :                             id is an existing endpoint
  LET props ≜ Allocate(endpoints[id], ⟨⟩)              Ask the allocator to remove all ports
  IN
   ∧ Assert(props ≠ {}, "Rejected remove operation!")
   ∧ ∃ prop ∈ props :
        Commit the removal proposal
       ∧ endpoints′ = [i ∈ DOMAIN endpoints \ {id} ↦ endpoints[i]]
       ∧ allocated′  = Apply(prop)

  The initial state of the system, with no endpoints or allocations. When restarting SwarmKit,
  saved endpoints can be loaded and allocated as if they were being added as new services using the
  Restore operation. Note: SwarmKit does not check whether the saved allocations are consistent
  at restore time.
Init ≜
   ∧ endpoints = ⟨⟩
   ∧ allocated  = {}

 The possible ways of using the allocator.
Next ≜
   ∨ NewEndpoint
   ∨ UpdateEndpoint
  $\lor RemoveEndpoint$

$Spec \triangleq$
  $Init \land \Box[Next]_{vars}$
```

```
 Properties to check

 Check that the variables have the expected types.
$TypeOK \triangleq$
  $\land allocated \subseteq Address$                                    A set of addresses
  $\land \text{DOMAIN } endpoints \subseteq EndpointID$                   A partial map from endpoint IDs
  $\land endpoints \in [\text{DOMAIN } endpoints \rightarrow Endpoint]$   to Endpoints.

  Check that the state of the system is consistent: all addresses marked as allocated are needed
  by some endpoint, all endpoints have a configuration that matches their requirements, and no
  two endpoints have been allocated the same address.
$AllocationsOK \triangleq$
   Every port the allocator thinks is allocated is actually used by some endpoint
  $\land \forall addr \in allocated :$
    $\exists e \in Range(endpoints) :$
    $\exists p \in Range(e.ports) :$
    $Addr(p) = addr$
   Every endpoint's configuration is correct
  $\land \forall eid \in \text{DOMAIN } endpoints :$
    LET $e \triangleq endpoints[eid]$
    IN
     $\land Len(e.spec) = Len(e.ports)$          We have the right number of ports configured
     $\land \forall i \in \text{DOMAIN } e.spec :$         For each port . . .
         LET $spec \triangleq e.spec[i]$
             $port \triangleq e.ports[i]$
         IN
           The actual port is the same as its specification, ignoring dynamic ingress port numbers.
          $\land$ IF $spec.mode = ingress \land spec.published\_port = dynamic$
              THEN $PortsMostlyEqual(spec, port)$
              ELSE $spec = port$
           The port's address is in the allocated set.
          $\land port.mode = ingress \Rightarrow Addr(port) \in allocated$
        There are no other users of this port. We only check spec.mode = ingress because we
        don't check collisions between host ports here and we'll find any host/ingress conflict
        anyway when we come to check the other port.

        XXX: host/host collisions need to be avoided by the scheduler, not the allocator. How-
        ever:

        "the scheduler is not involved in host mode ports. it was a very rushed feature, if i recall
        correctly, and it's sensitive to collisions."
```

```
      $\land spec.mode = ingress \Rightarrow$
        $\forall eid2 \in \text{DOMAIN } endpoints :$
        $\forall i2 \in \text{DOMAIN } endpoints[eid2].ports :$
          $\langle eid, i \rangle \neq \langle eid2, i2 \rangle \Rightarrow$        Don't check a port against itself
          LET $p2 \triangleq endpoints[eid2].ports[i2]$
          IN
            The other port must have a different network address:
            $\lor Addr(port) \neq Addr(p2)$
           XXX: an exception to this rule for ingress/host conflicts: We can't use the same
           address for an ingress and a host port because an ingress port must be allocated on
           every node, and so would conflict with the host port. However, this is a known bug
           in SwarmKit. For now, ignore ingress/host conflicts:
              $\lor p2.mode = host$

Check that spec is OK in itself (ignoring any other endpoints).
$SpecOK(spec) \triangleq$
  $\forall i \in \text{DOMAIN } spec :$   For every pair of ports $\langle i, j \rangle$
  $\forall j \in 1 \,..\, (i - 1) :$
    $\lor spec[i].mode = host$   Don't care about host-mode ports
    $\lor spec[j].mode = host$
    $\lor Addr(spec[i]) \neq Addr(spec[j])$          The requested addresses are different
    $\lor spec[i].published\_port = dynamic$   or they are both dynamic.

Special value to indicate creation of a new endpoint.
$nullId \triangleq \text{CHOOSE } x : x \notin EndpointID$

Checks that the allocator rejects a request only if it should.
$RejectJustified \triangleq$
  $\forall s \in EndpointSpec :$                                        s is the new spec
    $\forall eid \in \text{DOMAIN } endpoints \cup \{nullId\} :$          eid is the endpoint to update, or nullId for creation
    LET $oldEndpoint \triangleq \text{IF } eid = nullId \text{ THEN } nullEndpoint \text{ ELSE } endpoints[eid]$
          The possible allocations, or {} if rejected:
          $props \triangleq Allocate(oldEndpoint, s)$
          Ports used in our old configuration. We can conflict with these:
          $dealloc \triangleq \{Addr(p) : p \in \{p \in Range(oldEndpoint.ports) : p.mode = ingress\}\}$
          Ports not used by us:
          $usedByOthers \triangleq allocated \setminus dealloc$
          Ingress ports where the user chose the port number:
          $staticPorts \triangleq \{p \in Range(s) : \land p.mode = ingress$
                                                  $\land p.published\_port \neq dynamic\}$
          All the ingress addresses chosen by the user:
          $staticAddr \triangleq \{Addr(p) : p \in staticPorts\}$
          We expect the allocation to be rejected:
          $rejectOK \triangleq$
              The specification is itself invalid:
              $\lor \neg SpecOK(s)$
        We asked for a port that is already in use:
        ∨ staticAddr ∩ usedByOthers ≠ {}
        There aren't enough free dynamic addresses for some protocol:
        ∨ ∃ proto ∈ Protocol :
            LET dynNeeded ≜ Cardinality({i ∈ DOMAIN s :
                                            ∧ s[i].protocol = proto
                                            ∧ s[i].mode = ingress
                                            ∧ s[i].published_port = dynamic})
                dynAvail  ≜ Cardinality({a ∈ Address \ (usedByOthers ∪ staticAddr) :
                                            ∧ a[1] = proto
                                            ∧ a[2] ∈ DynamicPortNumber})
          Note: dynNeeded is an over-estimate because we might be able to reuse an existing
          static address.
            IN    dynNeeded > dynAvail
    IN
     If the allocator rejected the new spec, we understand why:
     ∨ props = {} ⇒ rejectOK
     ∨ Print(⟨props, rejectOK, endpoints, eid, s⟩, FALSE)    Log the reason on error

If an endpoint's spec didn't change, then its allocation shouldn't change either. This tests that
Allocate is idempotent. XXX: This is not currently the case, because if we have two similar
specs then we only copy the existing allocation for the first one.
StepAllocateIdempotent ≜
  ∀ eid ∈ DOMAIN endpoints ∩ DOMAIN endpoints′ :
  LET ep ≜ endpoints[eid]
  IN
  ep.spec = (ep.spec)′
   ⇒
  ep.ports = (ep.ports)′

All steps are idempotent.
AllocateIdempotent ≜
  □[StepAllocateIdempotent]_vars
```