Flooding Network Proofs

by Casper Rysgaard

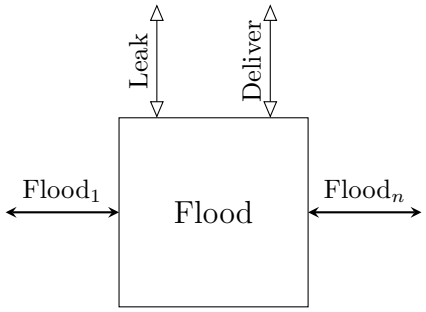
The proofs in the following report have been conducted in Coq, and can be found on
`https://github.com/Crowton/FloodingNetworkSafeAndLive`.

1 Introduction

In the following, we will consider a simple flooding network, with a fixed number of ports n. The purpose of the flooding network is to allow the ports to communicate, by sending messages on the network, which eventually arrives at the other ports. This thus represents the most basic of network, on which other protocols can be build to reorder the messages to arrive in a certain order.

In order to reason about the network, a model is needed. This model is the flooding agent, which will act as the flooding network. From the outside, the ports are connected. These ports can then send messages into the agent box, or receive messages from the agent box. Any other outside behavior to the agent is then modeled as special ports.

The model is described as the following interactive agent box:



Ports The channel connects n parties named $P_1, \ldots, P_n$. For each party P_i it has a port called Flood$_i$. It has special ports Leak and Deliver used to model that the network does not hide what is sent and that the adversary (or whomever controls Deliver) can determine when messages are delivered.

Init For each P_i it keeps a queue InTransit$_i$ which is initially empty.

Send On input (P_i, m) on Flood$_i$, it outputs (P_i, m) on Leak, and adds (P_i, m) to InTransit$_j$ for $j = 1, \ldots, n$. Note that it also adds to InTransit$_i$.

Deliver On input (P_i, P_j, m) on Deliver where $(P_i, m) \in$ InTransit$_j$, it removes (P_i, m) from InTransit$_j$ and outputs (P_i, m) on Flood$_j$.

From the above description, it can be observed, that the interactive agent can react to two different types of inputs, from here forth denoted as an `InputAction`, which is defined as

$$\begin{aligned}\texttt{InputAction} ::= \texttt{Send}\ &(\textit{port}, \textit{message}) \\ \mid \texttt{Deliver}\ &(\textit{port}, \textit{port}, \textit{message})\end{aligned}$$

where *port* is the type specifying how to describe a Port, and similarly *message* describe the type of the messages send on the network. Note that the first *port* of `Deliver` is the port the message comes from, and the second *port* is the port the message should be delivered at.

From the description, it can also be seen that for some port P_i, then the type of $\texttt{InTransit}_i$ is a set of *port*, *message* pairs. The collected `InTransit` buffer is then a map from a *port* P_i to the set $\texttt{InTransit}_i$.

The following properties of the system is then considered.

User Contract We require from the process P_i that it does not send the same m twice.

Safety If a correct P_j outputs (P_i, m), then earlier P_i sent (P_i, m).

Liveness If a correct P_i sends (P_i, m), then eventually all correct P_j deliver (P_i, m).

The main proof of this paper, is stating that the flooding agent comply with the Safety and Liveness property, under the User Contract assumption. A further assumption is added in order to prove Liveness, which states that the inputs are given to the flooding agent have Livenes, which will be defined later.

2 Functional formulations

2.1 Formulation of Flooding Agent

The flooding agent takes an `InputAction` and depending on the internal state of an `InTransit` buffer, potentially updates this buffer and/or potentially produces an output on some port. It can thus be viewed as a transition function, taking as input an `InputAction` and an `InTransit` buffer, and produces as output the updated `InTransit` buffer, and a potential output on some port. The last potential part of the output can be modeled as an optional pair of the port P_j, which the flooding agent outputs on, and the pair (P_i, m), which should be outputted. The function thus have the following signature.

$$Flood : (\texttt{InputAction}, \texttt{InTransit}) \rightarrow (\texttt{InTransit}, option\,(port, port, message))$$

For the formulation, *port* is modeled by a natural number, and *message* is modeled as a pair consisting of a string and a natural number. This is due to the later FiFo agent requiring sending information of this type on the flooding agent. It could be modeled as a single string, but cumbersome marshall and demarshall steps would need to be included, which is then skipped.

The individual $\texttt{InTransit}_i$ sets are modeled as a simple list, as this model is easier to reason about. The user contract then takes care of no duplicate messages being send, and thus the lists must be sets.

The map from a *port* P_i to an individual $\texttt{InTransit}_i$ set, is modeled as a list, as the *ports* are represented as natural numbers, the lookup is then finding the value at a specific index in the list.

From the description, it can be observed, that all $\texttt{InTransit}_i$ buffers initially is empty. In order to then model this initial master buffer `InTransit` buffer, the following recursive function is used, which takes a parameter n, which is the size of the buffer it should generate.

```
InitFloodBuffer (n: nat)=
    match n with
        case 0 ⇒ []
        case n' + 1 ⇒
            [] :: (InitFloodBuffer n')
```

To model the flooding agents behavior, some help functions are first established. First is a function, which adds a *port*, *message* pair to all $\texttt{InTransit}_i$ buffers in the master buffer. This looks as the following recursive function.

```
AddMessageToTransit (inTransit: InTransit) (i: port) (m: message) =
    match inTransit with
        case [] ⇒ []
        case head :: tail ⇒
            ((i, m) :: head) :: (AddMessageToTransit tail i m)
```

Secondly is two functions, which checks if a given *port*, *message* pair is included in the $\texttt{InTransit}_j$'th buffer. These are both simple recursive functions. The first is responsible for finding the j'th index in the master `InTransit` buffer, which correspond to $\texttt{InTransit}_j$. The second function is then responsible for checking is the *port*, *message* pair is included in the given $\texttt{InTransit}_j$ buffer, which is done recursively. Both functions return two values; the possibly updated buffer it works on and a *boolean* value corresponding if the element searched for was found. The convention is that if this *boolean* is *true*, then the searched element was removed from the buffer, and otherwise the buffer is unchanged. This convention is also proved later. The functions looks as follows.

```
ExtractMessage (inTransit: InTransit_j) (i: port) (m: message) =
    match inTransit with
        case [] ⇒ ([], false)
        case (i', m') :: tail ⇒
            if i = i' and m = m'
                then (tail, true)
                else Let (tail', success) = ExtractMessage tail i m
                     ((i', m') :: tail', success)

ExtractPort (inTransit: InTransit) (i j: port) (m: message) =
    match (inTransit, j) with
        case ([], _) ⇒ ([], false)
        case (head :: tail, 0) ⇒
            Let (head', success) = ExtractMessage head i m
            (head' :: tail, success)
        case (head :: tail, j' + 1) ⇒
            Let (tail', success) = ExtractPort tail i j' m
            (head :: tail', success)
```

The flooding agent function can now use the above functions, to model the behavior wanted. If the `InputAction` is a `Send`, then the given message should be added to all $\texttt{InTransit}_i$ buffers, which the function AddMessageToTransit achieves. If the `InputAction` is instead a deliver, the wanted element should be checked for incluetion, and the buffer potentially updated to remove this, which the function ExtractPort does. The agent then have to specify from the given information, if the optional output should be *None* or contain *Some* outputted message. This looks as the following simple function.

```
FloodAgent (inp: InputAction) (inTransit: InTransit) =
    match inp with
        case Send (i, m) ⇒
            (AddMessageToTransit inTransit i m, None)
        case Deliver (i, j, m) ⇒
            Let (inTransit', success) = ExtractPort inTransit i, j, m
            if success
                then (inTransit', Some (j, i, m))
                else (inTransit', None)
```

As the proof is based on not a single `InputAction` given to the agent, but a sequence of these, an help execution function is modeled, to take a list of `InputAction`'s and an initial buffer. The actions are then feed through the flooding agent, using the output buffer from one action, as input to the next action.

In order to reason about what happened during this execution, a trace of all events is produced, which is then finally output at the end, along with the final `InTransit` buffer, to help reason about how the different `InputAction`'s affected the buffer.

The possible events in the trace are as follows.

```
FloodEvent ::= Input (port, message)
             | Leak (port, message)
             | Deliver (port, port, message)
             | Output (port, port, message)
```

The help execution then looks as follows.

```
ExecuteFloodAgent (inps: list InputAction) (inTransit: InTransit) =
    match inps with
      case [] ⇒ ([], inTransit)
      case Send (i, m) :: inps' ⇒
          Let (inTransit', _) = FloodAgent (Send (i, m)) inTransit
          Let (trace', inTransit'') = ExecuteFloodAgent inps' inTransit'
          (Input (i, m) :: Leak (i, m) :: trace', inTransit'')
      case Deliver (i, j, m) :: inps' ⇒
```

```
match FloodAgent (Deliver (i, j, m))
  case (inTransit', None) ⇒
       Let (trace', inTransit'') = ExecuteFloodAgent inps' inTransit'
       (Deliver (i, j, m) :: trace', inTransit'')
  case (inTransit', Some (j, i, m)) ⇒
       Let (trace', inTransit'') = ExecuteFloodAgent inps' inTransit'
       (Deliver (i, j, m) :: Output (i, j, m) :: trace', inTransit'')
```

2.1.1 Formulation of Properties

For the following, the syntax $list[index] = element$ will be used to describe a lookup on the item *list* at position *index*. The equal to some *element* then state that the lookup is not out of bounds, and that the item at position *index* in *list* is *element*.

The user contract on a list of `InputActions` can be formally defined as follows.

Definition 1 (User Contract on `InputAction` list). Given a list of `InputActions` *inp*, such that $inp[t_1] =$ `Send` (p, m_1) and $inp[t_2] =$ `Send` (p, m_2), where $t_1 \neq t_2$, then *inp* is said to satisfy the user contract, when $m_1 \neq m_2$.

From the definition of a `FloodEvent` trace, it is possible to formally state the definition of safety and liveness as follows.

Definition 2 (Safety of `FloodEvent` trace). Given a list of `FloodEvents` *trace*, such that $trace[t_2] =$ `Output` (i, j, m), then *trace* is said to satisfy safety, when there $\exists\ t_1$ such that $t_1 < t_2$ and $trace[t_1] =$ `Input` (i, m).

Definition 3 (Liveness of `FloodEvent` trace). Given a list of `FloodEvents` *trace* and the number of ports in the network n, such that $trace[t_1] =$ `Input` (i, m), then *trace* is said to satisfy liveness, when $\forall\ j < n$ there $\exists\ t_2$ such that $t_1 < t_2$ and $trace[t_2] =$ `Output` (i, j, m).

As liveness requires that all ports eventually deliver a message send by some port, it cannot be proved for any list of `InputActions`. The eventuality of delivers can however be stated as a property on the `InputAction` list, as each message send is then later tried to be delivered. The proof is then, that these tries must be successful, and that liveness is then satisfy. This property looks as follows.

Definition 4 (Eventual deliver guarantee of `InputAction` list). Given a list of `InputActions` *inp* and the number of ports in the network n, such that $inp[t_1] =$ `Input` (i, m), then *inp* is said to satisfy liveness, when $\forall\ j < n$ there $\exists\ t_2$ such that $t_1 < t_2$ and $inp[t_2] =$ `Output` (i, j, m).

For liveness it becomes important to reason about which `Input` event corresponds to which `Send` action, to take the properties of definition 4 and transfer them to definition 3. Following definition 1, there cannot be duplicate `Send` actions, and thus this follows trivially from this. Liveness can however be achieved without this restriction, in which case there can be multiple `Input` and `Send` pairs. In order to then pair these up, it becomes sufficient to pair all `Input` events up with the last corresponding `Send` action. This property is defined as follows.

Definition 5 (No later `Send` actions from point). Given a list of `InputActions` *inp*, a point in the list t, and a *port* and *message* pair (p, m), then $\forall\ t' > t$, such that $inp[t'] =$ `Send` (p', m'), then *inp* is said to have no later `Send` of a specific type (p, m) if it holds that $(p, m) \neq (p', m')$.

3 Proofs of Safety and Liveness

3.1 Theorem statement

Theorem 1 (Safety). For any list of `InputActions` *inp* and a number of ports n, such that definition 1 hold on *inp*, and ExecuteFloodAgent *inp* (InitFloodBuffer n) = $(trace, finalBuffer)$, then the flooding agent is said to hold on safety if definition 2 holds on *trace*.

Theorem 2 (Liveness). For any list of `InputActions` *inp* and a number of ports n, such that definition 1 hold on *inp*, definition 4 hold on *inp* and n, and ExecuteFloodAgent *inp* (InitFloodBuffer n) = (*trace*, $finalBuffer$), then the flooding agent is said to hold on liveness if definition 3 holds on *trace*.

In order to prove the above theorems, a number of lemmas must be proven first.

3.2 User Contract Properties

Lemma 1 (User Contract on Sublist). For all two lists of `InputActions` *inp* and *inp*′, such that definition 1 holds on the concatenation of the lists *inp* `++` *inp*′, then definition 1 also holds for just *inp*.

Proof. By unfolding of definition 1. □

3.3 No Later `Send` Actions Properties

Lemma 2 (No Later `Send` Actions on Sublist). For all two lists of `InputActions` *inp* and *inp*′, such that definition 5 holds on the concatenation of the lists *inp* `++` *inp*′ along with a point in the list t, and a *port* and *message* pair (p, m), then definition 5 also holds on *inp* with t and (p, m).

Proof. By unfolding of definition 5 and destructing of the placement of t. □

Lemma 3 (No Later `Send` Action or No `Send` actions). For all lists of `InputActions` *inp* and *port* and *message* pair (p, m), it holds that there either $\exists\, t$ such that $inp[t] =$ `Send` (p, m) and definition 5 holds on *inp*, t and (p, m), or it holds that $\forall\, t$ then $inp[t] \neq$ `Send` (p, m).

Proof. By induction on *inp*. In the inductive case, destruct on the induction hypothesis and observe if the newest action is `Send` (p, m). □

3.4 Add Message to Transit Properties

Lemma 4 (Add Message to Transit Reveals Older Message). For all `InTransit` buffers buf and buf' and *port* and *message* pair (p, m), such that $buf' =$ AddMessageToTransit buf p m and for some t, where $buf'[t] = portBuf'$, then there must $\exists\, portBuf$, such that $portBuf' = (p, m)$ `::` $portBuf$.

Proof. By induction on buf. In the induction case destruct on t. □

Lemma 5 (Add Message to Transit for Old Lookups). For all `InTransit` buffers buf and buf' and *port* and *message* pair (p, m), such that $buf' =$ AddMessageToTransit buf p m and for some t, where $buf'[t] = (p, m)$ `::` $portBuf'$, then it holds that $buf[t] = portBuf'$

Proof. The proof has the same structure as the proof for lemma 4, using induction on buf and the definition of AddMessageToTransit. □

3.5 Extract Message and Port Properties

Lemma 6 (Extract Port not Success Returns Same Buffer). For all `InTransit` buffers buf, *port* j, and *port* and *message* pair (p, m), such that ExtractPort $portBuf$ p j m = (buf', $false$), then $buf = buf'$.

Proof. By induction on buf. In the inductive case destruct on j. For $j = 0$ construct similar proof on `ExtractMessage`, with induction on the $portBuf$. For $j > 0$ use the induction hypothesis. □

Lemma 7 (Extract Port Success Reveal the Message). For all `InTransit` buffers buf, some *port* j, and *port* and *message* pair (p, m), such that ExtractPort buf p j m = (buf', $true$), then there must $\exists\, k$, h such that $buf[k] = portBuf$ and $portBuf[h] = (p, m)$.

Proof. By induction on buf. In the inductive case destruct on j. For $j = 0$ construct similar proof on `ExtractMessage`, with induction on the $portBuf$. For $j > 0$ use the induction hypothesis. □

Lemma 8 (Extract Port Success Preserves other Messages). For all `InTransit` buffers buf, some $port$ j, and $port$ and $message$ pairs (p, m), such that ExtractPort buf p j $m = (buf', true)$, and then for all indexes k and h where $buf'[k] = portBuf'$ and $portBuf'[h] = (p', m')$, then there must $\exists\, k'$, h' such that $buf[k'] = portBuf$ and $portBuf[h'] = (p', m')$.

Proof. By induction on buf. In the inductive case destruct on j. For $j = 0$ construct similar proof on `ExtractMessage`, with induction on the $portBuf$. For $j > 0$ use the induction hypothesis. □

Lemma 9 (Extract from Add Message Result cannot Fail). For all `InTransit` buffers buf and buf' and $port$ and $message$ pair (p, m), such that $buf' =$ AddMessageToTransit buf p m, then $\forall\, j < |buf|$, there must $\exists\, buf''$, such that ExtractPort buf' p j $m = (buf'', true)$.

Proof. By induction on buf. In the inductive case destruct on j. For $j = 0$ destruct over the if condition of the `AddMessageToTransit`. For $j > 0$ use the induction hypothesis. □

3.6 Execution Properties

Lemma 10 (Concatenation of Input Lists results in concatenation of Traces). For all lists of `InputActions` inp and inp', `InTransit` buffers buf and buf'' and lists of `FloodEvents` $trace$, such that

$$\text{ExecuteFloodAgent } (inp \texttt{ ++ } inp')\ buf = (trace,\ buf'')$$

then there must $\exists\, buf'$, $traceHead$, $traceTail$, such that

$$trace = traceHead \texttt{ ++ } traceTail$$

$$\text{ExecuteFloodAgent } inp\ buf = (traceHead,\ buf')$$

$$\text{ExecuteFloodAgent } inp'\ buf' = (traceTail,\ buf'')\ .$$

Proof. By induction on inp. In the induction case destruct on the first `InputAction` in the list. □

Lemma 11 (Input in Trace reveals Send in Input). For all lists of `InputActions` inp, lists of `FloodEvents` $trace$ and `InTransit` buffers buf and buf', such that ExecuteFloodAgent inp $buf = (trace, buf)$, where for some t and $port$ and $message$ pair (p, m) it holds that $trace[t] =$ `Input` (p, m), then there must $\exists\, t'$ such that $inp[t'] =$ `Send` (p, m).

Proof. By induction on inp. In the induction case destruct on the first `InputAction` in the list. □

Lemma 12 (Input in Trace reveals last Send in Input). For all lists of `InputActions` inp, lists of `FloodEvents` $trace$ and `InTransit` buffers buf and buf', such that ExecuteFloodAgent inp $buf =$ $(trace, buf')$, where for some t and $port$ and $message$ pair (p, m) it holds that $trace[t] =$ `Input` (p, m), then there must $\exists\, t'$ such that $inp[t'] =$ `Send` (p, m) and definition 5 holds for inp, t and (p, m).

Proof. Use lemma 11 to gain a `Send` in inp. Destruct over lemma 3, arriving at a contradiction in the second case. □

Lemma 13 (`Send` in Input start implies `Input` in Trace start). For all lists of `InputActions` inp, points t and $port$ and $message$ pair (p, m), such that $inp[t] = (p, m)$, and all `InputActions` $action$, such that definition 5 holds for (inp `++` $action$), t and (p, m), and all `InTransit` buffers buf and buf' and lists of `FloodEvents` $traceHead$ and $traceTail$, such that ExecuteFloodAgent $action$ $buf =$ $(traceTail, buf')$, and all t', such that $(traceHead$ `++` $traceTail)[t'] =$ `Input` (p, m), then $traceHead[t'] =$ `Input` (p, m).

Proof. By showing that the `Input` (p, m) cannot be in $traceTail$, it is implied that it must be in $traceHead$. □

Lemma 14 (Executing `InputAction` list preserves buffer length). For all lists of `InputActions` inp and `InTransit` buffers buf and buf', such that ExecuteFloodAgent $inp\ buf = (trace, buf')$ for some list of `FloodEvents` $trace$, then $|buf| = |buf'|$.

Proof. By induction on inp. In the inductive case use sub-lemmas for preservation of buffer length for AddMessageToTransit and ExtractPort. □

3.7 Safety

Lemma 15 (Trace Safety append non-`Output`). For all lists of `FloodEvents` $trace$, such that definition 2 holds on $trace$, and all `FloodEvents` t, such that $t \neq$ `Output` $(i,\ j,\ m)$, then definition 2 holds on $trace$ `++` $[t]$.

Proof. By unfolding 2 it is observed that for all elements in $trace$ `++` $[t]$ that is an `Output FloodEvent`, there must be some earlier `Input FloodEvent`. As t cannot be a `Output FloodEvent`, then the proof follows from the hypothesis of definition 2 holding on $trace$. □

Lemma 16 (Trace Safety append `Output`). For all lists of `FloodEvents` $trace$, such that definition 2 holds on $trace$, and all $port$ and $message$ pairs (i, m), such that $trace[t] =$ `Input` (i, m) for some t, then for all $port$s j, definition 2 holds for $trace$ `++` [`Output` (i, j, m)].

Proof. By unfolding 2, it is observed that an earlier `Input FloodEvent` must be found for each `Output FloodEvent`. Destructing over the placement of the `Output` in $trace$ `++` [`Output` (i, j, m)], the `Output` in $trace$ can be proved using the hypothesis, that definition 2 holds on $trace$. The second case can then be proved using the `Input` placement hypothesis. □

Lemma 17 (Buffer messages reveals `Input` in the trace). For all lists of `InputActions` inp, number of ports n, lists of `FloodEvents` $trace$ and `InTransit` buffers buf, such that

$$\text{ExecuteFloodAgent } inp\ (\text{InitFloodBuffer } n) = (trace, buf)$$

and all indexes i and j and all $port$ and $message$ pairs (p, m), such that $buf[i][j] = (p, m)$, then there must $\exists t$, $trace[t] =$ `Input` (p, m).

Proof. By reverse induction on inp. In the inductive case use lemma 10 to use the induction hypothesis, gaining that for the resulting buffer after the first part of execution, the property holds. Then destruct on the latest `InputAction` in inp.

For a `Send InputAction`, use that for the first message in each $\texttt{InTransit}_i$ buffer in buf, the newest `Send` is the desired `Send`, and for later messages, then lemma 4 and 5 allows for the induction hypothesis to be applied.

For a `Deliver InputAction`, use lemma 8 for successful delivers and lemma 6 for unsuccessful delivers, to apply the induction hypothesis. □

Restatement of Theorem 1 (Safety). For any list of `InputActions` inp and a number of ports n, such that definition 1 hold on inp, and

$$\text{ExecuteFloodAgent } inp\ (\text{InitFloodBuffer } n) = (trace,\ finalBuffer)$$

then the flooding agent is said to hold on safety if definition 2 holds on $trace$.

Proof. By reverse induction on inp. In the inductive case use lemma 10 to use the induction hypothesis, gaining that definition 2 holds for the first part of the list. Then destruct on the latest `InputAction` in inp. For a `Send InputAction` using lemma 15 allows for the induction hypothesis to be used.

For a `Deliver InputAction`, use lemma 15 for a unsuccessful deliver, and apply the induction hypothesis. For a successful deliver use lemma 7 to gain an index in buf with the message delivered, and use lemma 17 to gain that a `Send InputAction` must be in the first part of the inp. Then use lemma 16, to apply the induction hypothesis. □

3.8 Liveness

Lemma 18 (Unsuccessful deliver implied by unsuccessful deliver on different added message). For all `InTrasit` buffers buf and buf' and *port*s i, j and i' and *message*s m and m', such that $(i, m) \neq (i', m')$ and ExtractPort (AddMessageToTransit buf i' m') i j m = (buf', $false$), then there must $\exists\, buf''$ such that ExtractPort buf i j m = (buf'', $false$)

Proof. By induction on buf. In the inductive case destruct on j. For $j = 0$ the message is tried to be extracted from the first $\texttt{InTransit}_i$ buffer. By the hypothesis' then the message cannot be the first element. It cannot be in the second half either, so the same buffer is returned. For $j > 0$ the proof follows from the induction hypothesis. □

Lemma 19 (Fail on last `Deliver` with earlier `Send` implies successful `Deliver` in between). For all lists of `InputAction`s inp, points in the list t_1, *port*s i and j, *message* m, number of ports n, lists of `FloodEvent`s $trace$ and `InTransit` buffers buf, such that $inp[t_1]$ = `Send` (i, m),

$$\text{ExecuteFloodAgent } inp \text{ (InitFloodBuffer } n) = (trace, buf) \ ,$$

$j < n$ and ExtractPort buf i j m = (buf', $false$) for some buf', then there must $\exists t_2, t_1 < t_2$ and $inp[t_2]$ = `Deliver` (i, j, m).

Proof. By reverse induction on inp. In the inductive case destruct on the position of t_1. With t_1 in the first part of the list, destruct on the latest `InputAction` in inp. For a `Send InputAction` it must be a different send, and using lemma 18 allows for the induction hypothesis to be used. For a `Deliver InputAction` it is either the desired `Deliver` or the induction hypothesis can be applied.

For t_1 being the last index in the list, the `Deliver` cannot fail, thus arriving at a contradiction. □

Lemma 20 (`Send` followed by `Deliver` in Input implies `Input` and `Output` in Trace). For all lists of `InputAction`s inp, *port*s i and j, *message*s m and points t_1 and t_2, such that $inp[t_1]$ = `Send` (i, m), $trace[t_2]$ = `Deliver` (i, j, m), $t_1 < t_2$, definition 5 holds for inp with t_1 i and m, and for all number of ports n, lists of `FloodEvent`s $trace$ and `InTransit` buffers buf, such that

$$\text{ExecuteFloodAgent } inp \text{ (InitFloodBuffer } n) = (trace, buf) \ ,$$

and $trace[t_1']$ = `Input` (i, m) for some t_1', and $j < n$, then there must $\exists\, t_2'$, $t_1' < t_2'$ and $trace[t_2']$ = `Output` (i, j, m).

Proof. By reverse induction on inp. In the inductive case destruct over the position of t_1. When t_1 is in the first part of inp, further destruct over the position of t_2 in inp. When both are in the start of the list, use lemma 2 and 13 to apply the induction hypothesis.

When t_2 is the end of the list, the latest `InputAction` must be `Deliver` (i, j, m). This deliver is either successful or not. For a successful deliver, the `Input` must be in the first part of $trace$ and the `Output` in the last part of the list. For an unsuccessful deliver, using lemma 19 implies that an earlier `Deliver` must be in inp. Using lemma 2 then allows for the induction hypothesis to be applied.

When t_1 is the end of the list, then t_2 must be an element outside of the list, which gives a contradiction. □

Restatement of Theorem 2 (Liveness). For any list of `InputAction`s inp and a number of ports n, such that definition 1 hold on inp, definition 4 holds in the inp and n, and

$$\text{ExecuteFloodAgent } inp \text{ (InitFloodBuffer } n) = (trace, finalBuffer) \ ,$$

then the flooding agent is said to hold on liveness if definition 3 holds on $trace$.

Proof. Unfolding definition 3 and sing lemma 12 gives the position of the `Send InputAction` in *inp*. Unfolding definition 4 gives the position of the `Deliver InputAction` in *inp*. Using lemma 20 then gives the position of the `Output FloodEvent` in *trace*, thus proving the theorem. □

4 References

Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlando. *Distributed Systems and Security.* 2020. Chapter 4, Consistent Communication.