CSCI 234

Design of Internet Protocols: Formal Validation

George Blankenship

Outline

- Construction of a Formal Validation
- File Transfer Protocol
- Assumptions
- Transfer phases
- Model assumptions
- Modeling protocol layers

Construction of a Formal Validation

- *Build a high level prototype and verify that the design criteria are met*
- Protocol design is an iterative process. Not likely to be correct first time around
- Each time a design phase is completed, be convinced it is error-free

File Transfer Protocol

- A *point-to-point* protocol
- One sender and one receiver
- Provides *end-to-end* service between two users on two different machines

4 Elements of the Protocol

1. Service Specification
   a. connection establishment
   b. termination
   c. recovery from transmission errors
   d. flow control strategy
2. Transfer ASCII text files
3. Low undetected bit error probability
4. User able to abort a file transfer in progress and protocol able to recover from message loss

Channel Assumptions

- Full-duplex transfer
  – voice grade telephone lines
- Ignore networking issues
  – i.e. routing
- Minimal time for message to travel is approx 0.15 seconds

Protocol Vocabulary

- Initiate a file transfer
  `transfer(file_descriptor)`
- Interrupt a transfer in progress
  `Abort`
- Message to the source file server to verify the file and size
  `open(file_descriptor)`
- Connection is made with the remote file server
  `connect(size)`
- Message to remote file server to create a new file of the given size
  `create(size)`
- Message from remote file server to indicate whether an open or a create request is accepted or rejected
  `accept(size)`
  `reject(size)`

Formal Validation George Blankenship 7

Transfer Process Phases

1. **Establishment of a connection** with the local file server
2. **Establishment of a connection** with the remote file server
3. **Transfer** of data
4. **Orderly termination** of the connection

Formal Validation George Blankenship 8

Connection Between File Servers

- Two steps
  1. Initialization of the flow control protocol
  2. A handshake with the remote system using the **`connect`** and **`accept`** or **`reject`** message
- The synchronization of local and remote flow control protocols is necessary to guarantee that they agree on the initial sequence numbers to be used
  – A handshake using the message **`Sync`** and its acknowledgment **`sync_ack`**

Formal Validation George Blankenship 9

Data Transfer

- We need messages for retrieving the data from the file server and transmitting them to the remote system

  `data(cnt, ptr)`
- To signify the end of the transmission

  `eof`
- Completion of a file transfer

  `close`
- A simple flow control discipline that acknowledges correctly received data

  `ack`

Message Format

- Messages minimally require:
  - a type field
  - optional data field
- Messages carry:
  - sequence numbers
  - checksum

```
struct {
unsigned type : 4;
unsigned seqno : 2;
unsigned char data[376];
unsigned char checksum[2];
} message;
```


Procedure Rules

- Consider only the semantics of the protocol ignoring the syntax
- Layers
- Design divided into several layers:
  - Presentation layer: the user interacts with this layer
  - Session layer: controls the transfer itself
  - Data link layer: assumed that it can lose messages but not distort them

Protocol Environment

- User process
- File server
- Data link
- Can be two user processes, **one on each end of the data link**
  - Users can **submit a transfer** request at any time
  - After transfer request, originating user may also decide to **abort a transfer**
  - User waits for a response from the lower protocol layers, signaling success or failure of a completed transfer

User Layer

```
P proctype user_process(bit n){
   user_to_pres[n]!transfer;
   if
         :: pres_to_use[n]?accept -> goto Done
         :: pres_to_use[n]?reject -> goto Done
         :: use_to_pres[n]!abort->goto Aborted
   fi;
   Aborted:
   if
         :: pres_to_use[n]?accept -> goto Done
         :: pres_to_use[n]?reject -> goto Done
   fi;
   Done: skip
   }
```

`n` - identifies the user and the channels that it accesses

`transfer` - a message ordinarily carry a parameter that points to the file transferred

```
#define QSZ  N /* queue size */
chan use_to_pres[2] = [QSZ] of { byte };
chan pres_to_use[2] = [QSZ] of { byte };
chan pres_to_ses[2] = [QSZ] of { byte };
chan ses_to_pres[2] = [QSZ] of {byte,byte};
chan ses_to_flow[2] = [QSZ] of {byte,byte};
chan flow_to_ses[2] = [QSZ] of {byte,byte};
chan dll_to_flow[2] = [QSZ] of {byte,byte};
chan flow_to_dll[2] = [QSZ] of {byte,byte};
```


Channels

- Channels for the synchronous communication between the session layer and the file server

```
chan ses_to_fsrv[2] = [0] of { byte };
chan fsrv_to_ses[2] = [0] of { byte };
```

- Ten different types of message channels
- One copy being instantiated for each side of the connection

File Server/Presentation

- An incoming file transfer begins with a **`create`** message.
- The file server responds with an **`accept`** or a **`reject`** message.
- If the request is accepted zero or more data messages follow.
- The file server falls back into its initial state upon reception of the final **`eof`** or **`close`** (on abort).
- Important at this level is *when* data can be passed, not *which* data will be passed

```
proctype fserver(bit n) {
      int fd, size, ptr, cnt;
      do
      :: ses_to_fsrv[n]?create(size)->
               if /*nondeterministic choice*/
               :: fsrv_to_ses[n]!reject
               :: fsrv_to_ses[n]!accept ->
                  do
                  :: ses_to_fsrv[n]?data(cnt,ptr)
                  :: ses_to_fsrv[n]?eof->break /*abort*/
                  :: ses_to_fsrv[n]?close -> break
                  od
               fi
      :: ses_to_fsrv[n]?open(fd)-> /* outgoing */
      ...
od
}
```


Presentation Layer

- Reasons for transfer to fail

1. Local system busy serving an incoming file transfer
2. Local server rejects the request, e.g., file does not exist
3. Remote server rejects the request, e.g., no space
4. Collision between incoming and outgoing transfer requests
5. Transfer aborted by the user

- Reasons 1 and 4 are transient, may disappear if request is repeated.

```
#define FATAL         1   /* failure type */
#define NON_FATAL     2   /* repeatable   */
#define COMPLETE      3   /* success      */
proctype present(bit n){
     byte status, uabort;
IDLE:
     do
     :: use_to_pres[n]?transfer ->uabort = 0;goto TRANSFER
     :: use_to_pres[n]?abort -> skip  /* ignore */
     od;
TRANSFER:
     pres_to_ses[n]!transfer;
     do
     :: use_to_pres[n]?abort ->
               if
               :: (!uabort) ->uabort = 1;use_to_pres[n]!abort
               :: (uabort) -> skip
               fi
     :: ses_to_pres[n]?accept -> goto DONE;
     :: ses_to_pres[n]?reject(status) ->
               if
               :: (status == FATAL || uabort) ->goto FAIL
               :: (status == NON_FATAL &&!uabort) ->goto
     TRANSFER
               fi
     od;
DONE:
     pres_to_use[n]!accept;
     goto IDLE;
FAIL:
     pres_to_use[n]!reject;
     goto IDLE;
}
```


Presentation Layer Assumption

- The main assumption the presentation layer makes about the session layer is that it will eventually respond to a transfer request with either an `accept` or a `reject` message.

Session Layer

- Mediator between presentation and link layer
- Message arriving at data link is queued to session then to presentation
- Message created by presentation is queued to session and then queued to link

```
proctype session(bit n) {
        bit toggle;
        byte type, status;
IDLE:
        do
        :: pres_to_ses[n]?type ->
                        if
                        :: (type == transfer) -> goto DATA_OUT
                        :: (type != transfer) -> /* ignore */
                        fi
        :: flow_to_ses[n]?type ->
                        if
                        :: (type == connect) -> goto DATA_IN
                        :: (type != connect) ->/* ignore */
                        fi
        od;
          DATA_OUT:
                ...
          DATA_IN:
                ...
          }

DATA_IN:
        ses_to_fsrv[n]!create;
        do
        :: fsrv_to_ses[n]?reject -> ses_to_flow[n]!reject;goto IDLE
        :: fsrv_to_ses[n]?accept -> ses_to_flow[n]!accept;break
        od;
        ... incoming data transfer ...
        ... close connection etc.  ...
```


Data Link Layer

- *Data link* is protected with an error detection protocol. Can arbitrarily omit messages from the sequence that is passed, using some hidden oracle to decide the fate of each message
- This can be modeled as a nondeterministic choice

```
proctype data_link(){
   byte type, seq;
   do
   :: flow_to_dll[0]?type,seq ->
       if
       :: dll_to_flow[1]!type,seq
       :: skip  /* lose */
       fi
   :: flow_to_dll[1]?type,seq ->
       if
       :: dll_to_flow[0]!type,seq
       :: skip  /* lose */
       fi
   Od
   }
```


Correctness Requirements

- As a correctness requirement for the presentation layer we will identify its valid end-states.
- There is only one valid end-state, the `IDLE` state.
- We replace `IDLE` with `endIDLE`.
- This is the state the presentation layer has to be when a transfer terminates.
- Provided that the assumptions about the user layer and session layer are true, it is possible to show that this requirement is satisfied.
- We look at the possibilities:
- The presentation layer may block in several statements, however, according to the assumptions about the user and session layers none of those statements can remain nonexecutable forever