Formal Analysis of Network Configuration Protocols: An Overview

Alwyn E. Goodloe
NASA Langley Research Center

George Hagen
NASA Langley Research Center

Brendan Luksik
NASA Johnson Space Center

Motivation

- As space missions become more complex and longer duration, the avionics are becoming complex distributed systems
  - Expected to operate without significant downtime or human management
  - Very long durations
- Adapting Ethernet variants for networking
  - Time-Triggered Ethernet (TTE)
  - Avionics Full-Duplex Switched (AFDX) Ethernet
- Design tradeoffs favor determinism
  - Static network configuration enables predictable behavior
- Fixed number of network elements
- Each node maintains one or more configuration files
- Changes to the network configuration requires files be updated
- Given the long duration of space missions, how best to do the update?

Ways to Update Config Files

- Preplan and store all conceivable configurations
  - Consumes many resources
  - Assumes it is possible to plan for any scenario
- Gound based controllers could manually upload each file
  - Known to be error prone
  - Depends on reliable communication
- Have astronauts manually update files
  - Not practical and very error prone
- Develop a protocol that is robust to faults and failures to reconfigure the network
  - NASA researchers have developed such a protocol
- Such a protocol will need to have undergone extensive analysis
  - Failure can endanger the spacecraft and/or humans onboard!!

Network Communication Structure

Roles of Nodes in System

- Ground System (GS)(Root of Trust) – Starting point of all commands
  - Symmetrically omissive/Fail Silent
- Contact Coordinator (CC) – Direct connection to ground. Orchestrates distribution of messages
  - Byzantine/Fail Arbitrary
- Module Coordinator (MC) – “Regional” coordinator orchestrating distribution of messages to a particular fault containment region (FCR)
  - Byzantine/Fail Arbitrary
- Switch (SW) – Nodes that route messages, but can execute protocol commands
  - Asymmetric Omissive/Fail Arbitrary
- End System Participant (ES) - The end system nodes only accept and respond to the protocol
  - Byzantine/Fail Arbitrary

Protocol Structure

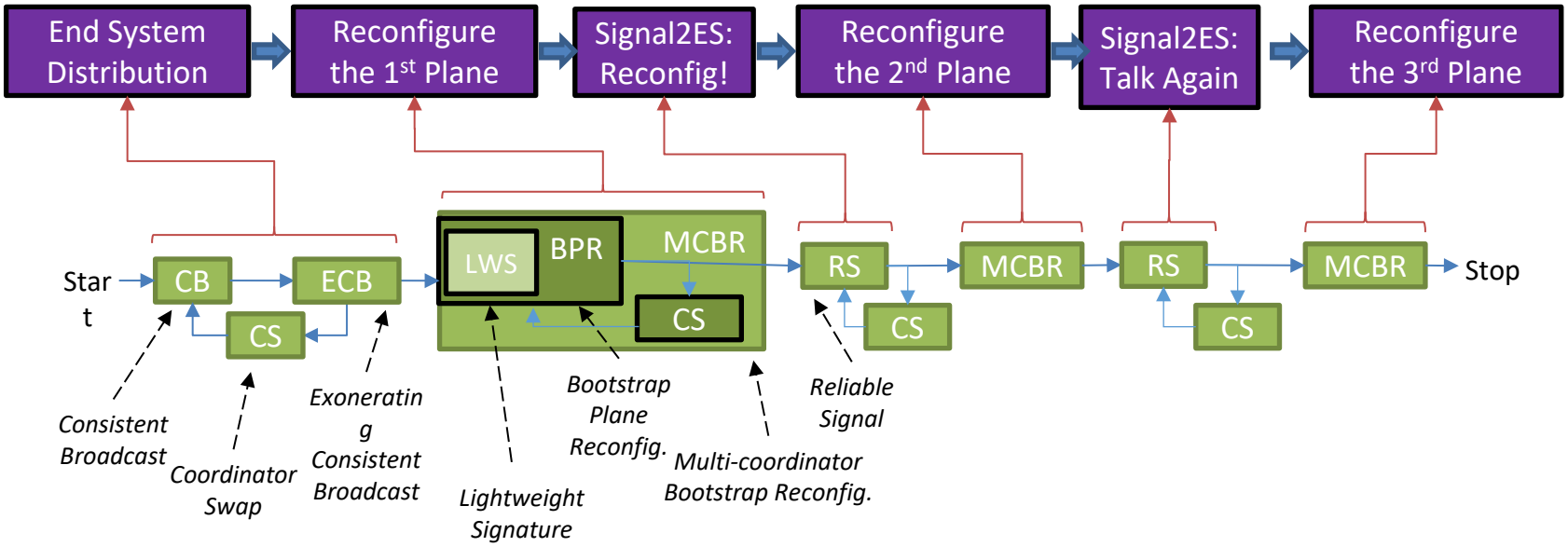
- The protocol is composed from a set of primitives
- Primitives are either cryptographic operations or small protocols
- Primitive protocols follow the same pattern:
  - Ground system sends command to coordinator
  - Coordinator running the protocol will broadcast commands to receiving nodes
  - Receiving nodes receive command and perform an operation and send a reply to coordinator
  - Ground system requests an acknowledgement
  - Coordinator gathers acks from participants and sends reply to ground system
  - Protocol on ground evaluates and acts on the information in the ack it receives

Protocol Primitives

- Coordinator Swap (CS) – Swap out/in nodes designated as coordinators
- Bootstrap Plane Reconfiguration (BPR) – One coordinator directs the reconfiguration of switches in a plane
- Multi-Coordinator Bootstrap (MBPR) – Reconfigure switches using all available coordinators
- Reliable Bootstrap Reconfiguration (RBR) – Disable planes when MBR fails
- Consistent Broadcast (CB) – Configure end systems
- Exonerating Consistent Broadcast (ECB) – Identify faults when configuring end systems
- Reliable Signal (RS) – Send value to nodes and verify that coordinator sent values to each node

Protocol Steps

Flow of Top-Level Operations



Flow of Primitives that Comprise Top-Level Operations

Challenge

- For the primitives, developers identified several correctness properties:
  - Authenticity
  - Validity
  - Verifiability
- Informal proofs of correctness of individual primitives
- Developers wanted to increase confidence that there was no unintended harmful interactions among the primitives when composed into a larger protocol
- They asked us to help answer this question
- We are reporting on WORK IN PROGRESS

Modeling Process

- Construct an abstract model of the system
  - Model network elements (switches, connections)
  - Model protocol primitives and their composition
    - Abstraction requires tradeoffs
- We are building two models using different tools making different tradeoffs
  - One model naturally synchronous, network layout changed easily, and easily model arbitrary node failures
  - The other model has more fidelity in modeling network elements, but with fewer nodes and more difficult to change configuration
- Properties to be checked
  - Do primitives interfere with each other?
  - Failure modes
  - Currently in early stages

What We Model

- Network elements - Switches, coordinators, end systems, parallel network planes
- Coordinator Swap (CS) – Swap out/in nodes designated as coordinators
- Consistent Broadcast (CB) – Configure end systems
- Exonerating Consistent Broadcast (ECB) – Identify faults when configuring end systems
- Reliable Signal (RS) – Send value to nodes and verify that coordinator sent values to each node

Maude

- Maude is a high-level specification language
  - Developed at SRI and UIUC
- Maude is a typed language where the types are called *Sorts*
  - Object oriented
- Equations create equivalent classes and substitute one equal term with another
  - eq $t = t'$
- Rewriting rules transform terms in ways that do not necessarily substitute one term for another
  - Rewriting is a logic of concurrent change
  - rl $t_1\ t_2\ \ldots t_n\ \rightarrow t'_1\ t'_2\ \ldots t'_m$.
  - Crl $t_1\ t_2\ \ldots t_n\ \rightarrow t'_1\ t'_2\ \ldots t'_m$ if $e = e'$.

Protocol Models

- Each protocol is modeled as a state machine executing at a node
  - State machines defined for GS, CC, MC, SW, and ES
  - Each state is a rule in the model
- Model abstracts away implementation details
- Protocols are simplified to configure one node in system
- Must limit state explosion

Queue in Maude

```
fmod QUEUE {X :: TRIV} is

  sort NeQueue{X} Queue{X} .
  subsort NeQueue{X} < Queue{X} .
  op empty : -> Queue{X} [ctor] .
  op enqueue : Queue{X} X$Elt -> NeQueue{X} [ctor] .
  op dequeue : NeQueue{X} -> Queue{X} .
  op first : NeQueue{X} -> X$Elt .
  op isEmpty : Queue{X} -> Bool .

  eq dequeue(enqueue(empty,E)) = empty .
  ceq dequeue(enqueue(Q,E)) = enqueue(dequeue(Q),E) if Q =/= empty .
  eq first(enqueue(empty,E)) = E .
  ceq first(enqueue(Q,E)) = first(Q) if Q =/= empty .
  eq isEmpty(empty) = true .
  eq isEmpty(enqueue(Q,E)) = false .
  eq isEmpty(enqueue(empty,E)) = false .
```

Channels

- Channel is an object comprised of a queue and identifiers for the end points

< A : Channel | queue : Q , in : B, out : C >

- All nodes in the network are connected by pairs of unidirectional channels
- Messages are sent and received by putting them into and removing the from channel queues

Packets and Switches

- Packets have source, destination, msg type, and payload
- op < _ _ _ | _ > : Address Address MsgType Payload -> Packet
- Switches move packets
- Keep more than one routing table at each switch
  - Config swap
- Each table maps addresses to channel IDs
- Switch is a map of maps :
- op sw-routingtable :_ : Map{Nat , Map{Address,Oid}} -> Attribute .
- When routing a packet, select the routing table and look at the destination of the packet in an inbound queue and lookup the channel to put that packet in
  - (RT [SelRT]) [pi-dst(first(QI))

Kind2

- Developed at University of Iowa
- SMT-based K-induction model checker
  - Model checking + Inductive proofs (up to a depth limit)
  - Supports FOL LTL properties
- Lustre input
  - Used in aviation industry, basis of Esterel development platform
  - Synchronous dataflow language
  - Data streams and functional modules (nodes)

Network layout example (lustre)

```
node toy_system (  ok : bool^numNodes;  startConfig:int; timeOut:int;  root_msgIn:msgType)  returns
(root_msgOut:msgType);
[...]
let
  root_msgOut = planes_A_msg_a;
  (planes_A_msg_a, planes_A_msg_b) = planes_1(ok, sw_pl_A_id_base, startconfig, mcordid, root_msgIn,
con_cord_msg_a);
  (con_cord_msg_a, con_cord_msg_b) =
            coordinator_1_1(ok, ccordid, startconfig, rootid, cChildren, timeOut, planes_A_msg_b,
planes_B_msg_a);
  (planes_B_msg_a, planes_B_msg_b) =
            planes_1(ok, sw_pl_B_id_base, startconfig, mcordid, con_cord_msg_b, mod_cord_msg_a);
  (mod_cord_msg_a, mod_cord_msg_b) =
            coordinator_1_1(ok, mcordid, startconfig, ccordid, mChildren, timeOut-1, planes_B_msg_b,
planes_C_msg_a);
  (planes_C_msg_a, planes_C_msg_b) =
            planes_3(ok, sw_pl_C_id_base, startconfig, mcordid, mod_cord_msg_b, end_pt_msg_a1,
end_pt_msg_a2);
  (end_pt_msg_a1, end_pt_msg_b1) = end_sys(ok, endid1, startconfig, ccordid, planes_C_msg_b, null_message);
  (end_pt_msg_a2, end_pt_msg_b2) = end_sys(ok, endid2, startconfig, ccordid, planes_C_msg_b, null_message);
tel
```

Network layout example (block diagram)

Lustre modeling

- Nodes represent physical systems (switches, coordinators, end systems)
- Streams represent network connections
- Message packets are data streams of records
- Node failures are based on a status stream, which can disable arbitrary nodes, e.g. no more than 1 simultaneous failures
- Switches are largely pass-through for message packets, excepting targeted configuration packets
- End systems store configurations and reply to targeted messages
- Most protocol logic is in coordinator nodes
- Network planes consist of parallel series of switches

Message Packets

- Stream of Packets:

```
type msgType = struct {
  mtype : mtypes; (* message type id *)
  sourceId : int; (* origin, used to check coordinator *)
  targetId : int; (* destination *)
  newValue : int; (* new value to store in log *)
  fields : int^numNodes; (* return log payload *)
};
```

Immediate Next Steps

- Verify that the composed system does not have any unintended inference among the components
  - ECB ; CS ; ECB ; RS; CS ; RS
- Inject faults into module coordinator, switches and end systems
- Explore different classes of faults
- Likely fault scenarios for MC and ES:
  - Fail Omissive – a device fails to send or receive an arbitrary number of packets
  - Fail-Inconsistent – One set of receivers gets correct messages and another gets detectably incorrect
  - Fail Arbitrary - device is free to generate arbitrary packets at arbitrary points in time. Device can fail inconsistently

Future Work

- Model plane reconfiguration protocols
- Explore how to modularize/automate aspects of the model building
- Explore how to better model synchronized protocols in Maude
- Probabilistic model checking
- Publish models via NASA software release process

Questions?

Contact Information:

Alwyn Goodloe.
a.goodloe@nasa.gov

George Hagen
george.hagen@nasa.gov

Brendan Luksik
brendan.k.luksik@nasa.gov