Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling AODV

(Ad hoc On-Demand Distance Vector Routing Protocol)

Università di Pisa, Dipartimento di Informatica boerger@di.unipi.it
Universität Ulm, Abteilung Informatik alexander.raschke@uni-ulm.de

See Ch. 6.1 of ModelingCompanion

Why is a rigorous AODV model needed?

- C.E.Perkins/E.M.Belding-Royer/S.R.Das: *AODV* RFC 3561 de facto standard docu of 2003 is partly ambiguous, incomplete, contradictory
  - various implementations show different behavior on protocol relevant features (e.g. loops, route discovery, packet delivery, optimal routes)
    - a huge number of different interpretations of RFC 3561 are possible
- widely believed, performance relevant loop freedom claim (by Perkins and Royer in 1999) established only in 2013 for a 'correct' interpretation of AODV
  - in: A. Fehnker, R. van Glabbeek, P. Höfner, A. McIver, M. Portmann, W.L. Tan: A process algebra for wireless mesh networks used for *modelling, verifying and analysing AODV* (TR 5513, NICTA, 2013)
  - after some wrong/partial proofs in the literature
- TR 5513 identifies also some performance relevant shortcomings of AODV and five key implementations

Goal of this lecture

- explain the functional core behavior of AODV for users and programmers, 'from scratch' and reliably, by stepwise developing an Abstract State Machine (ASM) model
  - reflecting a correct and complete understanding of the (core) requirements in the de facto standard document RFC 3561
    - but informed by the professional analysis in the NICTA TR 5513

as a prerequisite for a rigorous high-level analysis, long before coding

- performed in the NICTA TR 5513, in process algebraic terms, for different interpretations and implementations of the RFC 3561 wrt
  - loop freedom
  - route discovery
  - packet delivery

  and related correctness and performance relevant issues

Mobile Ad hoc Network (MANET) routing protocols

In MANETs *every network agent*

- can move independently to change its position
- can (try to) send messages to every ‘directly connected’ network node it knows (’neighbor’)
- can (try to) broadcast messages to all its ‘neighbors’
- for (wireless) communication with any other network agent must ask a routing protocol to indicate a communication path to that destination

The *routing protocol*

- receives and elaborates *route request* messages, by forwarding them and generating reply messages once a route has been found
- receives and elaborates *route reply* messages by forwarding them back to the original requestor
- creates and propagates *route error* messages if some broken direct link is detected

Background and agent/protocol interaction structure

Background structure:

- network: graph $(Agent, Link)$ with dynamic sets of nodes and edges
- determines for each $a \in Agent$ a dynamic set $neighb(a)$

Agent/protocol interaction: when a *WantsToCommunicateWith* d

- in case a *KnowsActiveRouteTo*(d), it can right away $\textsc{StartCommunicationWith}(d)$, without entering *WaitingForRouteTo*(d) (which is initialized by $false$)
- otherwise it must $\textsc{GenerateRouteReq}(d)$ and becomes $WaitingForRouteTo(d)$ until via the protocol run it eventually $KnowsActiveRouteTo(d)$
- to $\textsc{GenerateRouteReq}(d)$ only once per required communication, it is called only when a is not already $WaitingForRouteTo(d)$
  - easily refinable to permit repeated route requests to a same d

Agent/protocol interaction component

$\textsc{PrepareComm} =$

if $\mathit{WantsToCommunicateWith}(\mathit{destination})$ **then**

if $\mathit{KnowsActiveRouteTo}(\mathit{destination})$

then

$\textsc{StartCommunicationWith}(\mathit{destination})$

$\mathit{WantsToCommunicateWith}(\mathit{destination}) := \mathit{false}$

$\mathit{WaitingForRouteTo}(\mathit{destination}) := \mathit{false}$

else

if not $\mathit{WaitingForRouteTo}(\mathit{destination})$ **then**

$\textsc{GenerateRouteReq}(\mathit{destination})$

$\mathit{WaitingForRouteTo}(\mathit{destination}) := \mathit{true}$

NB. $\mathit{WantsToCommunicateWith}$ is assumed to be set to true only by the application program and to false only by $\textsc{PrepareComm}$.

Router components and AODV program structure

The main AODV program each node is equipped with:

AODVSPEC = **one of** -- main program

PREPARECOMM

ROUTER

The ROUTER consists of components to process request, reply and error msgs:

ROUTER = **one of**

PROCESSROUTEREQ

PROCESSROUTEREP

PROCESSROUTEERR

PROCESSROUTEERR = **one of**

GENERATEROUTEERR

PROPAGATEROUTEERR

AODV data structure: route table $RT(a)$ information

Each route table $entry$ keeps agent a's knowledge that

- the destination $dest(entry)$ of the $entry$ may be reachable
- in the direction as indicated by a neighbor node $nextHop(entry)$
- on a path of length $hopCount(entry)$ (distance to the destination)
- which can be considered as $Active$ (without $LinkBreak$)

To avoid loops in communication paths, each agent keeps a local request/-level counter:

- $localReqCount(a)$ and $curSeqNum(a)$, both initialized by 0
- $curSeqNum(a)$ possibly incremented when a receives a new request to reach a, after a $LinkBreak$ which made an $Active$ path to a $InActive$

Each route table $entry$ for $d = dest(entry)$ also records

- as $destSeqNum(entry)$ the last known value of $curSeqNum(d)$
- by $known(entry) \in \{true, false\}$ whether this number is valid

NB. For $precursor(entry)$ info for route error handling see below

AODV data structure: route table entry attributes

$$entryFor(d, RT) = \begin{cases} entry & \textbf{if forsome } entry \in RT \; dest(entry) = d \\ \textbf{undef} & \textbf{else} \end{cases}$$

$KnowsActiveRouteTo(destination)$ **iff**
$Active(entryFor(destination, RT))$

$$lastKnownDestSeqNum(d, RT) = \begin{cases} destSeqNum(entry) & \textbf{if forsome } entry \in RT \; dest(entry) = d \\ unknown & \textbf{else} \end{cases}$$

$ValidDestSeqNum(entry)$ **iff** $known(entry) = true$

AODV **data structure:** *RouteRequest*

Each route request msg $rreq \in RouteRequest$, when generated, records information about:

- the request destination $dest(rreq) \in Agent$
- the last known value $destSeqNum(rreq) \in NAT \cup \{unknown\}$ the request originator knows about the $curSeqNum(dest(rreq))$
- $known(rreq)$ indicating whether $destSeqNum(rreq)$ is reliable
- the request originator $origin(rreq) \in Agent$
- the originator's $originSeqNum(rreq) \in NAT$
- the length $hopCount(rreq) \in NAT$ of the path the $rreq$uest traveled from its $origin(rreq)$ to its current $rreq$-sender
- the value $localId(rreq) \in NAT$ of $localReqCount + 1$ at the $origin(rreq)$ when the $rreq$ is generated

NB. Global identification of $rreq$ via the following equation:

$$globalId(rreq) = (localId(rreq), origin(rreq))$$

AODV data structure: $RouteReply$

Each route reply msg $rrep \in RouteReply$, when generated, records information about:

- the destination $d = dest(rrep) \in Agent$ of the detected route
- the value $destSeqNum(rrep) \in NAT$ of $curSeqNum(d)$
  - as known at an intermediate node (not d), if $rrep$ is generated there
  - or as updated when the $dest$ination node d generates $rrep$
- the request originator $origin(rrep) \in Agent$ to whom the reply is addressed
- the length $hopCount(rrep) \in NAT$ of the current route from the $rrep$-sender to $dest(rrep)$

NB. We abstract from concerns about $rrep$ $lifetime$, network traffic and performance properties.

AODV **data structure:** $RouteError$

Each route error msg $rerr \in RouteError$, sent by an agent a,

- indicates a set of destinations, together with their increased $destSeqNum$ value, which became unreachable via a, i. e. cannot be reached at present using a as $nextHop$ of a route entry
- inactivates every route table $entry$ which uses the $rerr$ sender a as $nextHop$ to any relevant unreachable destination communicated by $rerr$
- is forwarded along the *precursor* chain

GenerateRouteReq(*destination*)

let $r = \mathbf{new}\ (RouteRequest)$ **in**

$dest(r) := destination$

$destSeqNum(r) :=$

$lastKnownDestSeqNum(destination, RT)$

if $entryFor(destination, RT) \neq \mathbf{undef}$

then $known(r) := known(entryFor(destination, RT))$

else $known(r) := false$

$origin(r) :=$ **self** $\quad originSeqNum(r) := curSeqNum + 1$

$hopCount(r) := 0 \quad localId(r) := localReqCount + 1$

$\textsc{Broadcast}(r)$ -- i.e. **forall** $n \in neighb$ **do** $\textsc{Send}(r,$ **to** $n)$

$\textsc{Increment}(curSeqNum) \quad \textsc{Increment}(localReqCount)$

$\textsc{Buffer}(r)$ -- i.e. $\textsc{Insert}(globalId(r), ReceivedReq)$

NB. $\textsc{Buffer}(r)$ helps to recognize whether r has been 'seen' already

$\text{ProcessRouteReq}(rreq)$

if $Received(rreq)$ **and** $rreq \in RouteRequest$ **then**

  if not $AlreadyReceivedBefore(rreq)$ **then** -- $rreq$ processed once

    $\text{Buffer}(rreq)$

    if $HasNewReverseRouteInfo(rreq)$ **then**

      $\text{BuildReverseRoute}(rreq)$

   seq

    if $FoundValidPathFor(rreq)$

      then $\text{GenerateRouteReply}(rreq)$

      else $\text{ForwardRefreshedReq}(rreq)$

  $\text{Consume}(rreq)$

NB. $\text{GenerateRouteReply}$ sends $rrep$ly to $nextHop$ in—possibly by BuildReverseRoute updated—$entryFor(origin(rreq), RT)$, which could be different from $sender(rreq)$.

A req where $HasNewReverseRouteInfo(req)$ is false

$AlreadyReceivedBefore(req)$ **iff** $globalId(req) \in ReceivedReq$

// i.e. req has been BUFFERed when received for the first time

Such route req msgs are simply discarded. Nothing else happens.

Otherwise, if the req brings no new reverse route information to the RT of the receiving agent, the existing reverse route entry is kept unchanged and the protocol proceeds to either GENERATEROUTEREPLY($rreq$) or FORWARDREFRESHEDREQ($rreq$).

$HasNewReverseRouteInfo(req)$ **iff** $req \in RouteRequest$ **and**

$ThereIsNoRouteInfoFor(origin(req), RT)$ **or**

$(ThereIsRouteInfoFor(origin(req), RT)$ **and**

$HasNewOriginInfo(req, RT))$

When $HasNewOriginInfo(req)$ to update the reverse route

$HasNewOriginInfo(req, RT)$ **iff**

let $entry = entryFor(origin(req), RT)$

$originSeqNum(req) > destSeqNum(entry)$

or $originSeqNum(req) = destSeqNum(entry)$ **and**

$(hopCount(req) < hopCount(entry)$ **or**

not $Active(entry))$

NB. For comparison with $destSeqNum$bers, every natural number n is stipulated to be better than $unknown$, formally $unknown < n$.

$\textsc{BuildReverseRoute}(rreq)$ component

if $ThereIsRouteInfoFor(origin(rreq), RT)$ **then**

$\quad \textsc{UpdateReverseRoute}(entryFor(origin(rreq), RT), rreq)$

else $\textsc{ExtendReverseRoute}(RT, rreq)$

where

$\quad \textsc{UpdateReverseRoute}(e, req) =$

$\qquad destSeqNum(e) := originSeqNum(req)$ -- freshest $destSeqNum$

$\qquad known(e) := true \qquad Active(e) := true$

$\qquad nextHop(e) := sender(req)$

$\qquad hopCount(e) := hopCount(req) + 1$ -- maybe shorter path

$\quad \textsc{ExtendReverseRoute}(RT, req) =$

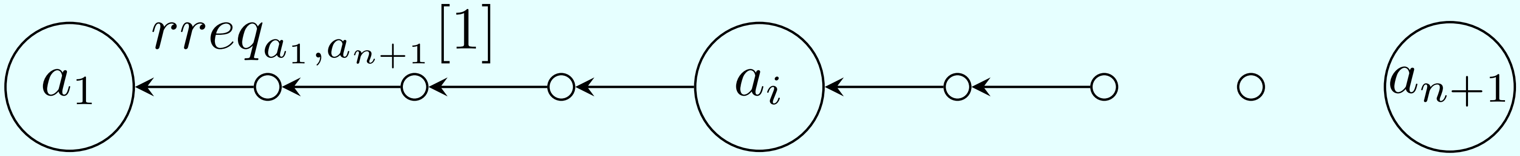
$\qquad$ **let** $e = \mathbf{new}\ (RT)$

$\qquad\quad dest(e) := origin(req) \qquad precursor(e) := \emptyset$

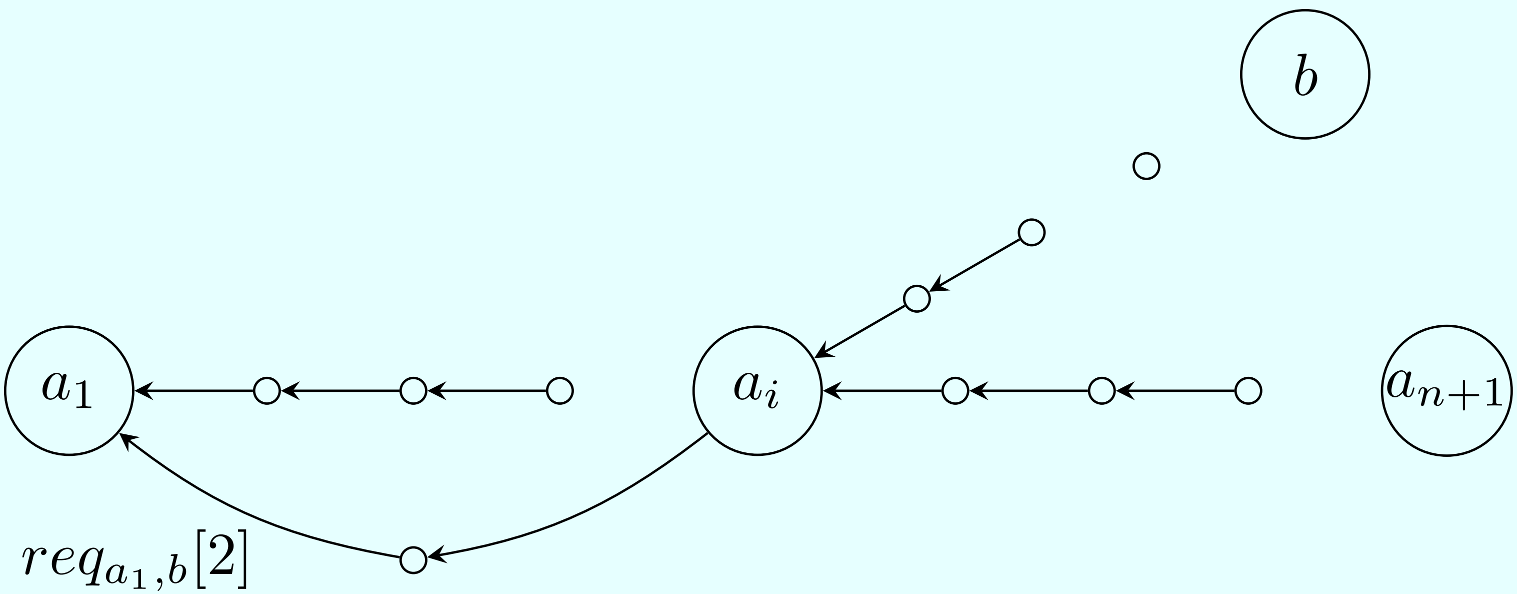
$\qquad\quad \textsc{UpdateReverseRoute}(e, req)$

Redirecting Reverse Route example

- Reverse routes for segments of $rreq$-path from a_1 to a_{n+1} are created.



- a_i receives a new req from a_1 to another destination and redirects the $rreq$ reverse route at a_i.[1]



[1] Figures © 2018 Springer-Verlag Berlin Heidelberg, reprinted with permission

The meaning of $FoundValidPathFor(req)$

- either req is received by the $dest$ination node
- or an 'intermediate' node which $KnowsFreshEnoughRouteFor$ the $dest(req)$ receives the req:

$FoundValidPathFor(req)$ **iff**

$dest(req) =$ **self** **or** $KnowsFreshEnoughRouteFor(req, RT)$

A 'fresh enough' route entry besides being $Active$ must have a $ValidDestSeqNum$ber that is not smaller than the $destSeqNum$ber of the received route request :

$KnowsFreshEnoughRouteFor(req, RT)$ **iff forsome** $entry \in RT$

$dest(entry) = dest(req)$ **and** $ValidDestSeqNum(entry)$

and $destSeqNum(entry) \geq destSeqNum(req)$

and $Active(entry)$

req **forwarding increases** $hopCount$ **and possibly** $destSeqNum$

$\textsc{ForwardRefreshedReq}(r) =$

let $r' =$ **new** $(RouteRequest)$

$\textsc{Copy}(dest, origin, originSeqNum, localId, known,$

from r **to** $r')$

$hopCount(r') := hopCount(r) + 1$

$destSeqNum(r') := max\{destSeqNum(r),$

$lastKnownDestSeqNum(dest(r), RT)\}$

$\textsc{Broadcast}(r')$

where

$\textsc{Copy}(f_1, \ldots, f_n,$ **from** arg **to** $arg') =$

forall $1 \leq i \leq n$ **do** $f_i(arg') := f_i(arg)$

$\textsc{GenerateRouteReply}(rreq)$ **at** $dest$ **or intermediate node**

let r = **new** $(RouteReply)$

let $revEntry = entryFor(origin(rreq), RT)$

$dest(r) := dest(rreq)$

$origin(r) := origin(rreq)$

if $dest(rreq) =$ **self then** -- reply at destination node

$hopCount(r) := 0$

$destSeqNum(r) := max\{curSeqNum, destSeqNum(rreq)\}$

$curSeqNum := max\{curSeqNum, destSeqNum(rreq)\}$

else let $fwdEntry = entryFor(dest(rreq), RT)$ -- at intermediate

$hopCount(r) := hopCount(fwdEntry)$ -- node

$destSeqNum(r) := destSeqNum(fwdEntry)$

$\textsc{PrecursorInsertion}(nextHop(revEntry), fwdEntry)$

$\textsc{Send}(r,$ **to** $nextHop(revEntry))$ -- maybe to $sender(rreq)$

The role of precursor nodes

Consider the case that a node a has an $entryFor(d, RT)$ and by a $rerr$or msg, received by a, node d is reported as unreachable.

Then each $neighbor(a)$ which has a route entry to d that uses a as $nextHop$ is called a $precursor$ of a.

The $precursor$ set is recorded in $entryFor(d, RT)$ so that such a $rerr$or msg can be propagated to its elements.

$$\textsc{PrecursorInsertion}(node, entry) =$$
$$\quad \textsc{Insert}(node, precursor(entry))$$

NB. We leave 'gratuitous' replies as an exercise. Through gratuitous replies, a destination node obtains a reverse route to the request originator without having requested a route, namely in case an intermediate node answered the request. For this case one needs also a $\textsc{PrecursorInsertion}$ of $nextHop(fwdEntry)$ into $precursor(revEntry)$.

ProcessRouteRep($rrep$)

if $Received(rrep)$ **and** $rrep \in RouteReply$ **then**

  if $HasNewForwardRouteInfo(rrep)$ **then** -- else just discard $rrep$

    BuildForwardRoute($rrep$)

    if $MustForward(rrep)$ **then** ForwardRefreshedRep($rrep$)

  Consume($rrep$)

where -- note symmetry to ProcessRouteReq

  $HasNewForwardRouteInfo(rep)$ **iff** $rep \in RouteReply$ **and**

    $ThereIsNoRouteInfoFor(dest(rep), RT)$ **or**

      $(ThereIsRouteInfoFor(dest(rep), RT)$ **and**

        $HasNewDestInfo(rep, RT))$

  $MustForward(rep)$ **iff**

    $origin(rep) \neq$ **self** **and** $Active(entryFor(origin(rep), RT))$

Meaning of $HasNewDestInfo$

$HasNewDestInfo(rep, RT)$ **iff** -- rep has either

let $entry = entryFor(dest(rrep), RT)$

$destSeqNum(rep) > destSeqNum(entry)$ -- better $destSeqNum$

or $(destSeqNum(rep) = destSeqNum(entry)$

and $hopCount(rep) + 1 < hopCount(entry))$

-- or shorter path

or $(destSeqNum(rep) = destSeqNum(entry)$

and $Active(entry) = false)$ -- or **not** $Active(entry)$

NB. Remember $unknown < n$ for each $n = 0, 1, \ldots$

$\textsc{BuildForwardRoute}(rrep)$ with fresh info

if $ThereIsRouteInfoFor(dest(rrep), RT)$

then $\textsc{UpdateForwardRoute}(entryFor(dest(rrep), RT), rrep)$

else $\textsc{ExtendForwardRoute}(RT, rrep)$ -- create new entry

where

$\textsc{UpdateForwardRoute}(e, rep) =$ -- copying fresh info

$destSeqNum(e) := destSeqNum(rep)$ $known(e) := true$

$nextHop(e) := sender(rep)$ $hopCount(e) := hopCount(rep) + 1$

$Active(e) := true$ $\textsc{SetPrecursor}(rep, e)$

$\textsc{ExtendForwardRoute}(RT, rep) =$

let $e =$ **new** (RT)

$dest(e) := dest(rep)$ $\textsc{UpdateForwardRoute}(e, rep)$

$\textsc{SetPrecursor}(rep, e) =$ **if** $MustForward(rep)$ **then**

$\textsc{Insert}(nextHop(entryFor(origin(rep), RT)), precursor(e))$

$\textsc{ForwardRefreshedRep}(rep)$

Upon forwarding rep, only the $hopCount$ is updated:

$$\textsc{ForwardRefreshedRep}(rep) =$$

$$\quad \textbf{let } rep' = \mathbf{new}\ (RouteReply)$$

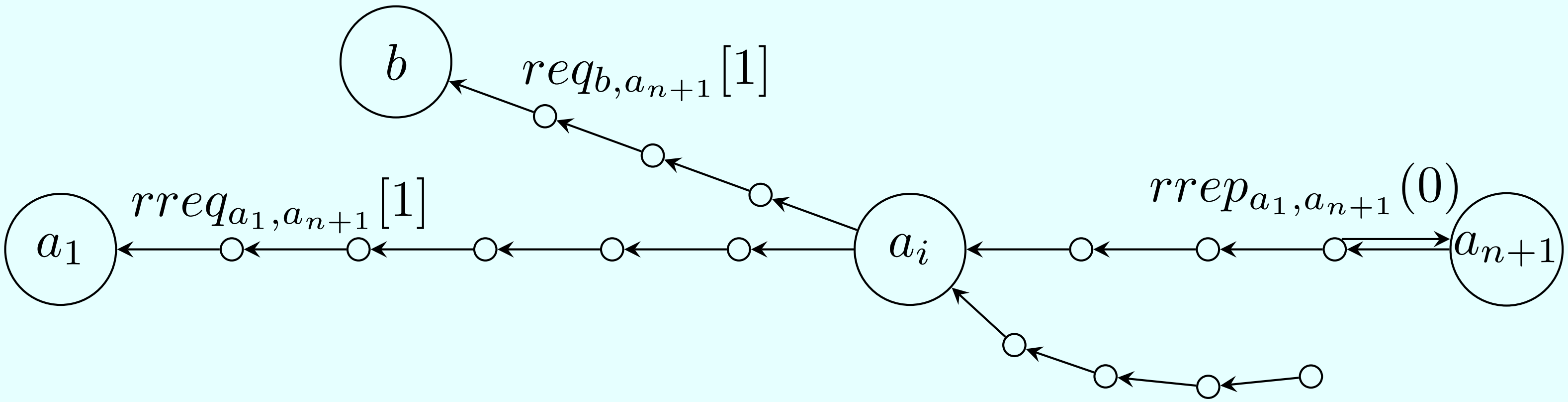
$$\quad\quad \textsc{Copy}(dest, destSeqNum, origin, \textbf{from } rep \textbf{ to } rep')$$

$$\quad\quad hopCount(rep') := hopCount(rep) + 1$$

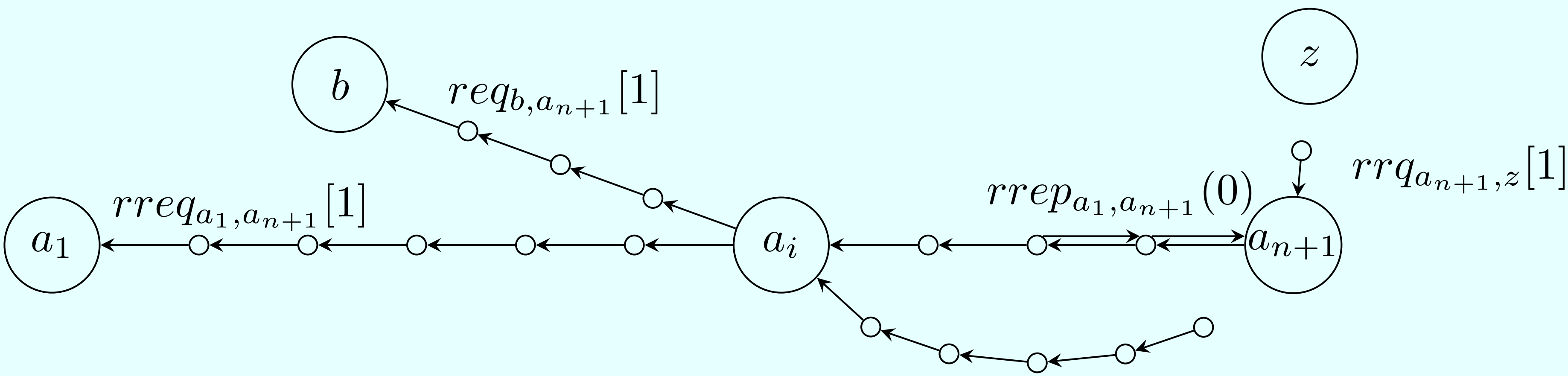
$$\quad\quad \textsc{Send}(rep', \textbf{to } nextHop(entryFor(origin(rep), RT)))$$

Redirecting Forward Route example (1)

- Destination a_{n+1} answers $rreq$ before req by $rrep$ with $destSeqNum$ 0

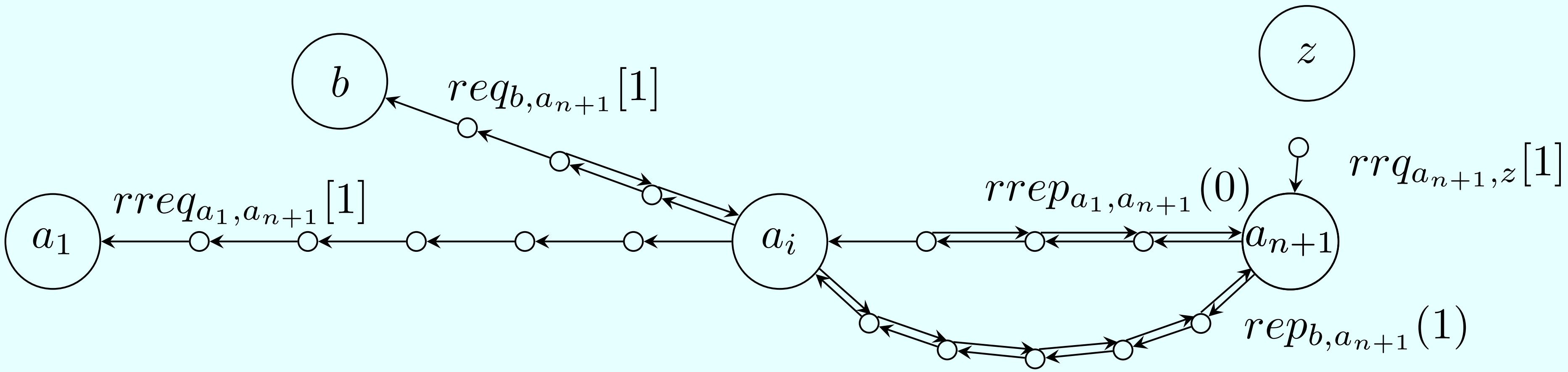


- a_{n+1} broadcasts a new rrq for destination z, $curSeqNum := 1$



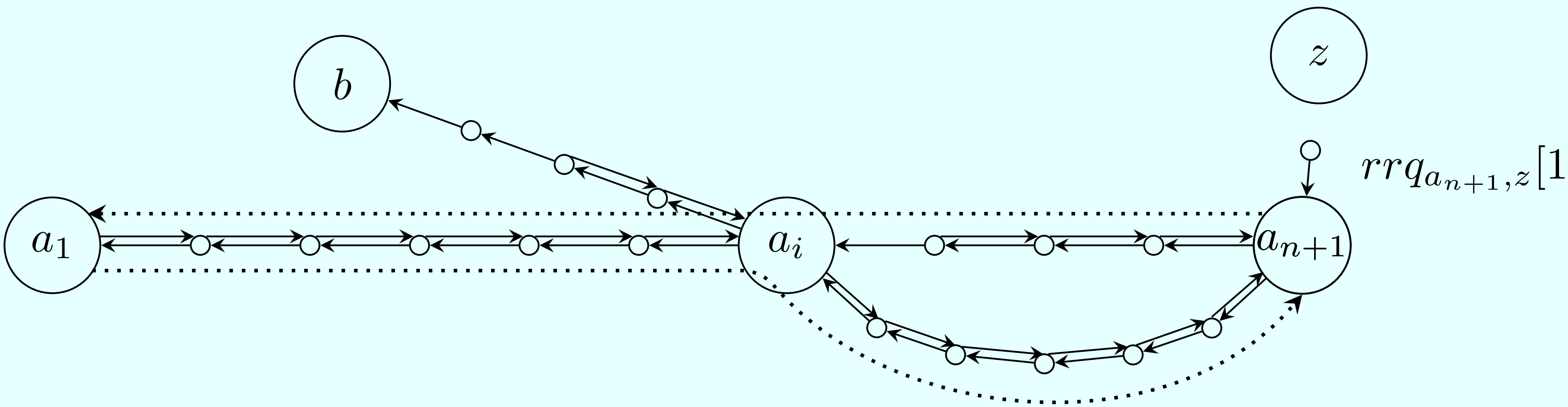
Redirecting Forward Route example (2)

- a_{n+1} answers req by rep with $destSeqNum(rep) := 1$
- a_i receives rep before $rrep$ and establishes $entryFor(a_{n+1}, RT(a_i))$ with $destSeqNum$ 1
- $rrep_{a_1,a_{n+1}}(0)$ is discarded at a_i



Redirecting Forward Route example (3)

- A new route request may be answered by a_i as intermediate node establishing the dotted communication path $(\cdots)$ from a_1 to a_{n+1}
- The communication path from a_{n+1} to a_1 still goes along the reverse route established by $rreq_{a_1,a_{n+1}}[1]$[2]



[2] Figures (1)-(3) © 2018 Springer-Verlag Berlin Heidelberg, reprinted with permission

Generation of error messages

$\textsc{GenerateRouteErr} =$

let $BrokenEntry =$ -- compute $Active$ entries with broken link

$\{entry \in RT \mid LinkBreak(nextHop(entry))$ **and** $Active(entry)\}$

forall $entry \in BrokenEntry$ -- if there are any

$Active(entry) := false$ -- Invalidate $entry$

$\textsc{Increment}(destSeqNum(entry))$ -- and $destSeqNum$

let $rerr = \{(dest(e), destSeqNum(e) + 1) \mid$

$e \in BrokenEntry$ **and** $precursor(e) \neq \emptyset\}$

forall $a \in precursor(entry)$ $\textsc{Send}(rerr,$ **to** $a)$

NB. Stipulation $unknown + 1 = unknown$

$\textsc{PropagateRouteErr}(rerr)$

if $Received(rerr)$ **and** $rerr \in RouteError$ **then**

  let $UnreachDest = \{(d, s) \in rerr \mid$ **forsome** $entry \in RT$

    $d = dest(entry)$ **and** $nextHop(entry) = sender(rerr)$

    and $Active(entry)$ **and** $destSeqNum(entry) < s\}$

  forall $(d, s) \in UnreachDest$ **let** $entry = entryFor(d, RT)$

    $Active(entry) := false$

    $destSeqNum(entry) := s$

    forall $a \in precursor(entry)$ $\textsc{Send}(rerr',$ **to** $a)$

    if $WaitingForRouteTo(d)$ **then** $\textsc{ReGenerateRouteReq}(d)$

  $\textsc{Consume}(rerr)$

where $err' =$

  $\{(d', s') \in UnreachDest \mid precursor(entryFor(d', RT)) \neq \emptyset\}$

$\textsc{ReGenerateRouteReq}(d) = (WaitingForRouteTo(d) := false)$

Conclusion: What else to do with a rigorous model?

- system debugging by rigorous model analysis
  - (dis)prove system properties of interest (e.g. using PVS, KIV,...)
    - for AODV: loop freedom and correctness proved, route discovery and packet delivery disproved for process algebra model in TR 5513 (for loop freedom using Isabelle)
  - identify shortcomings (here e.g. non-optimal routes)
  - testing/modelchecking of executable model refinements (e.g. in CoreASM, Asmeta, ...)
- system evaluation by comparison of implementations with the model
  - in TR 5513 done for five key AODV implementations, three of them shown to possibly produce routing loops
- model reuse for experimenting with system extensions/variations, prior to coding
- documentation for maintenance needs

References

- C.E. Perkins, E.M. Belding-Royer, S. Das: *Ad hoc On-Demand Distance Vector (AODV) Routing*
  – RFC 3561, Networking Group. http:///www.ietf.org/rfc/rfc3561.txt
- A. Fehnker, R. van Glabbeek, P. Höfner, A. McIver, M. Portmann, W.L. Tan: *A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV*
  – TR 5513, NICTA, 2013. http://www.nicta.com.au/pub?id=5513
- E. Börger, A. Raschke: *Modeling Companion for Software Practitioners*
  – Springer 2018. (Ch.6.1) http://modelingbook.informatik.uni-ulm.de