

به نام خدا



درس: شبکه مخابرات داده‌ها

استاد: دکتر محمدرضا پاکروان

گزارش پروژه شماره ۴

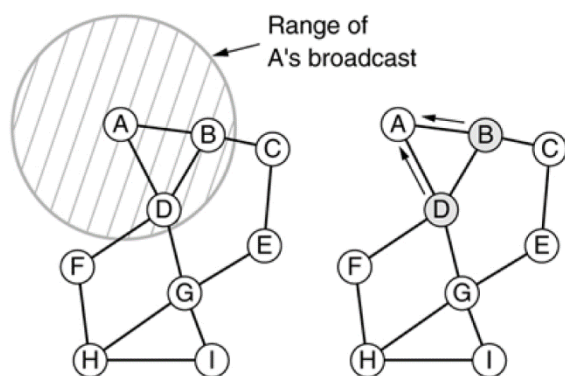
سید محمد امین منصوری طهرانی

۹۴۱۰۵۱۷۴

1. AODV Routing Protocol

۱. این روش routing مشابه روش DSR است و تفاوت در این است که به جای فرستاده شدن مسیر بهینه در هدر پکت‌های ارسالی از مبدأ که در شبکه‌های بزرگ که مسیر طولانی شود به صرفه نیست، اطلاعات مسیر در نودهای میانی ذخیره می‌شود. در ادامه به توضیح جزئیات روش با توجه به توضیحات مقاله و کلاس درس می‌پردازیم.

همان‌طور که گفته شد overhead پکت‌های ارسالی در شبکه‌های بزرگ قابل تحمل نیست. بنابراین به این سمت می‌رویم که جدول routing را در نودهای میانی تنظیم کنیم. ایده این است که تا زمانی که نیاز برقراری ارتباط به وجود نیامده است اطلاعاتی نگه داشته نشود. به محض این که برای مثال نود A به ارتباطی



با نود I نیاز داشت (تصویر زیر) با ارسال درخواست مسیر یا route request (RREQ) به همسایه‌های خود این نیاز را اعلام می‌کند. در بسته‌های ارسالی A یک destination sequence number نیز ارسال می‌شود تا مشخص شود چقدر این اطلاعات جدید و به روز است. هر نودی نیز node sequence number و یک

broadcast_id دارد. با هر بار درخواست مسیر جدید A این عدد (broadcast_id) یک واحد زیاد می‌شود. پکت‌های RREQ نیز حاوی متغیرهای زیر هستند:

Source_addr, source_seq_#, broadcast_id, dest_addr, dest_seq_#, hop_cnt

بسته‌های RREQ به وسیله دو متغیر broadcast_id و source_addr به طور یکتا مشخص می‌شوند و نودهای همسایه با دریافت آن‌ها اگر مسیری به مقصد مورد نظر مبدأ داشته باشند با ارسال بسته RREQ وجود این مسیر را با از طریق خودشان به مبدأ اعلام می‌کنند یا اگر مسیری به مقصد نداشته باشند و این بسته RREQ که گرفته‌اند تکراری نباشد (همان شماره مبدأ و همان شناسه broadcast) این بسته را به همسایه‌های خود forward می‌کنند و اطلاعات مسیر تا اینجا را نیز نگه می‌دارد (از چه کسی گرفته و به چه کسی داده). اگر این اتفاق بیفتد ضمن آن متغیر hop_cnt نیز یک واحد زیاد می‌شود.

Src_seq_# به روز بودن اطلاعات را راجع به مسیر بازگشت به مبدأ نشان می‌دهد و dest_seq_# مشخص می‌کند یک مسیر به مقصد باید چقدر جدید باشد تا به عنوان مسیر مناسب از طرف مبدأ پذیرفته شود. (مسیرهای قدیمی پذیرفته نشوند). در طی این گذر بسته RREQ هر نود اطلاعات همسایه‌ای که این بسته را از آن گرفته حفظ می‌کند و reverse path شکل می‌گیرد. مدت اعتبار این مسیرها حداقل به اندازه زمانی است که RREQ شبکه را طی کند و یک RREP تولید شود.

نهایتاً این بسته به یک نود می‌رسد که یا خود مقصد است یا نودی که به مقصد مسیری دارد. این نود میانی بررسی می‌کند `dest_seq_#` بسته از عدد مربوط به خودش بیشتر است یا نه. اگر عدد بسته کمتر بود همین مسیر را اعلام می‌کند چون جدیدتر است ولی اگر عدد بسته بیشتر بود این مسیر را به عنوان پاسخ به RREQ اعلام نمی‌کند. به جای آن بسته را به همسایه‌های خود broadcast می‌کند تا به نودی برسیم که بتواند مسیر را reply کند. این نود که بالاتر اشاره شد بسته RREP به صورت unicast در جهت برعکس مسیری که آمده می‌فرستد و broadcast نمی‌کند. بسته RREP نیز حاوی متغیرهای زیر است:

Src_addr, dest_addr, dest_seq_#, hop_cnt, lifetime

در مسیری که این بسته RREP باز می‌گردد نودهای میانی مسیر، نودی که این بسته را از آن گرفته‌اند ذخیره می‌کنند و اطلاعات timeout و اعتبار مسیر را نیز به روز رسانی می‌کنند. همچنین عدد `dest_seq_#` را ذخیره می‌کنند تا برای درخواست‌های بعدی مقایسه صحیح انجام شود. نودهایی هم که در مسیر RREP نیستند اطلاعات مسیر خود را پس از تمام شدن `active_route_timeout` دور می‌ریزند.

بنابراین به طور خلاصه با به وجود آمدن نیاز ارتباط درخواست از نودها می‌گذرد تا به مبدأ یا نودی که مسیر مناسب و جدید تا مقصد دارد برسد و از این‌جا بسته جواب به سمت مبدأ باز می‌گردد و نودهای میانی مسیر مناسب برای انتقال بسته از A به مقصد مورد نظرش را ذخیره می‌کنند.

یک نکته هم در مورد path maintenance این روش وجود دارد و آن این است که نودها مرتباً با ارسال active, hello messages بودن نودهای مجاور خود را بررسی می‌کنند و اگر یک نود در مسیر فعال مورد نظر حذف شده باشد این اطلاع به مبدأ می‌رسد تا اگر نیاز دارد دوباره RREQ تولید کند. همچنین مسیرهایی که پس از مدتی توسط مبدأ استفاده نشوند expire می‌شوند.

۲. تفاوت اساسی بین این روش و روش AODV را می‌توان در دو اصطلاح reactive بودن و proactive بودن خلاصه کرد. AODV اصطلاحاً reactive است و همان‌طور که در سوال قبل توضیح داده شد تا نیاز برقراری ارتباط اعلام نشود مسیری اختصاص داده نمی‌شود و اطلاعات زیادی از شبکه ذخیره نمی‌شود. اما OLSR روش بهینه شده‌ی Link State Routing است و می‌دانیم که این روش به صورت on-demand مسیریابی نمی‌کند. در این روش دائماً توپولوژی شبکه با پکت‌هایی در تمام نودها به روز رسانی شده و نودها با دانستن توپولوژی کل شبکه عملیات Routing را با داشتن sink tree ها انجام می‌دهند. پس latency اختصاص مسیر در این روش و همچنین بهینه شده آن که OLSR باشد بسیار کمتر است و این در ازای مصرف پهنای باند برای ارسال پیغام‌های کنترلی بدست می‌آید. نکته OLSR این است که این overhead را به طرز بهینه‌ای اضافه می‌کند که مجموعاً استفاده از این بسته‌های کنترلی و حاوی توپولوژی شبکه به صرفه باشد. در ادامه به توضیح OLSR می‌پردازیم.

در LSR بسته‌های حاوی همسایه‌ها در شبکه broadcast می‌شود. در این روش فقط زیر مجموعه‌ای از لینک‌ها به بقیه شبکه معرفی می‌شوند. به این‌ها multipoint relay selectors می‌گوییم. هر نود نیز flooding را با انتشار اطلاعات فقط از طریق تعدادی نود که آن‌ها را انتخاب می‌کند و اصطلاحاً multipoint relay (MPR) خوانده می‌شوند حداقل می‌کند. در واقع فقط این نودها اطلاعاتی که نود مبدأ فرستاده را دوباره broadcast می‌کند و همسایه‌هایی از نود مبدأ که multipoint relay نیستند با گرفتن اطلاعات آن‌ها را بازارسال نمی‌کنند. اطلاعاتی که هر نود می‌فرستد به بقیه اطلاع می‌دهد که کدام نودهای همسایه او، این نود را به عنوان multipoint relay انتخاب کرده‌اند. در ضمن تمامی پیام‌های کنترلی یک seq_num دارند که مشخص شود اطلاعات این بسته چقدر به روز است.

این MPR ها به نحوی انتخاب می‌شوند که تمام مقصدهای با فاصله دو hop از مبدأ توسط آن‌ها پوشش داده شود و هر چه تعداد این‌ها کمتر باشد به بهینه بودن نزدیک‌تر می‌شویم. (البته باید توسط لینک‌های bi-directional متصل باشیم.) مسیریابی از طریق این MPR ها انجام می‌شود و با دریافت پکت‌های جدید MPR selector که از یک مبدأ می‌آید مسیرها مجدداً محاسبه و به روز رسانی می‌شود.

عملیات شناخت همسایه‌ها به وسیله hello messages صورت می‌گیرد. در این پیام‌ها اطلاعات همسایه‌های نود و لینک‌های او قرار دارد. بنابراین با ارسال شدن این پیام‌ها و جابجایی آن‌ها نودها همسایه‌های خود تا دو مرحله را خواهند شناخت و سپس قادر خواهند بود با الگوریتمی که می‌خواهند MPR ها را انتخاب کنند. سپس باز با جابجایی hello message ها نودها می‌توانند بفهمند توسط چه نودهای دیگری انتخاب شده‌اند و جدول‌های MPR selectors خود را تشکیل دهند. باز هم برای این جدول‌ها seq_num وجود دارد که به روز بودن اطلاعات جدول را مشخص کند.

برای شناخت توپولوژی توسط نودها نیز، هر نود جدول MPR selector خود را به صورت broadcast در کل شبکه می‌فرستد و این پیام‌ها topology control (TC) messages نام دارند. به وسیله این پیام‌ها توپولوژی شبکه توسط همه نودها شناخته شده و از طریق MPR های متوالی routing انجام می‌شود. این TC message ها مرتباً در شبکه فرستاده می‌شوند و جدول توپولوژی نودها را آپدیت می‌کنند.

۳.

بخش اول: مشخص است که این قسمت کتابخانه‌های لازم برای اجرای کد را include می‌کند. ماژول‌های اضافه شده شامل ماژول اینترنت، شبکه، موبیلیتی، wifi، Aodv helper و ماژول netanim و تعدادی ماژول دیگر است.

بخش دوم: using namespace ns3 برای راحت‌تر نوشتن ادامه کد به کار می‌رود تا در ابتدای هر کلاس به نوشتن دوباره ns3:: لازم نباشد. خط بعد از آن نیز مشخص می‌کند در خروجی و قسمت log گزارش چه چیزهایی نوشته شود که در این کد گزارش routing مربوط به Ad-hoc routing خواسته شده‌است.

بخش سوم: خط اول transmission rate را تنظیم می‌کند و با توجه به تکنولوژی‌های مختلف wifi باید string مورد نظر مقداردهی شود. (std در واقع namespace استاندارد است که کلاس‌های زیادی را در بر دارد و این‌جا برای مقداردهی دو متغیر string که از آن‌ها استفاده خواهیم کرد استفاده شده‌است). متغیر رشته‌ای DataRate نیز مقداردهی می‌شود. سپس متغیر سایز بسته‌ها تعریف شده و مقدار آن ۵۱۲ بایت قرار داده شده‌است. همچنین متغیرهای عددی تعداد نودها و پورت UDP و متغیرهای Boolean تعریف و مقداردهی شده‌اند. (tracing و verbose)

در دو خط زیر نیز برای on-off application که در تمرین قبل با آن آشنا شدیم مقادیر default ای تعیین می‌کنیم.

```
Config::SetDefault ("ns3::OnOffApplication::PacketSize", UIntegerValue (PacketSize));  
Config::SetDefault ("ns3::OnOffApplication::DataRate", StringValue (DataRate));
```

کدهای مربوط به شیء از جنس commandline نیز برای تغییر مقادیر default توسط کاربر در حین اجرای کد گذاشته شده‌اند.

```
ns3::PacketMetadata::Enable();
```

کد بالا نیز packet metadata را فعال می‌کند. متادیتا سرویسی است که برای هر بسته ارائه می‌شود و اجازه می‌دهد به اطلاعات متنوعی از آن دسترسی داشته باشیم. (hostname, instance ID, ssh keys, assigned IPs)

در انتها نیز نامی که فایل xml برای netanim قرار است ذخیره شود تعیین می‌شود.

بخش چهارم: از کلاس wifiHelper برای ساختن مجموعه‌های بزرگ از device های شبکه wifi استفاده می‌شود. در خط بعدی نیز در صورت صحیح بودن متغیر verbose که دیفالت آن غلط تعریف شد، تمامی اجزای گزارش‌ها با این دستور فعال می‌شوند.

بخش پنجم:

```
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();  
wifiPhy.Set ("RxGain", DoubleValue (-12) );  
// ns-3 supports RadioTap and Prism tracing extensions for 802.11b  
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
```

خط اول برای ساخت و مدیریت اشیا از جنس PHY (لایه فیزیکی) برای مدل yans است. (که در حالت default قرار داده شده‌است).

سپس بهره گیرنده در لایه فیزیکی مقداردهی شده است و واحد آن dB است.

در خط آخر نیز در صورتی که pcap tracing مورد نیاز باشد با اجرای این خط کد می‌تواند آن را با یکی از پروتکل‌هایی که support می‌شود (در این جا IEEE802.11) فعال کند. همان packet capture است و با این کار گزارش بسته‌ها را در دست خواهیم داشت).

بخش ششم:

```
YansWifiChannelHelper wifiChannel;  
wifiChannel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");  
wifiChannel.AddPropagationLoss("ns3::FriisPropagationLossModel");  
wifiPhy.SetChannel (wifiChannel.Create ());
```

مشخص است که یک کانال بی‌سیم از کلاس کانال‌های بی‌سیم Yans تعریف شده و تاخیر انتشار و افت کانال آن بر مبنای مدل Friis برای آن تعریف شده است. در خط آخر نیز کانال بی‌سیم تعریف شده به لایه فیزیکی تعریف شده در قسمت قبل مربوط می‌شود.

بخش هفتم:

```
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();  
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);  
wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager", "DataMode",StringValue(phyMode),  
"ControlMode",StringValue (phyMode));
```

در خط اول لایه MAC ای از کلاس Nqos برای شبکه‌های بی‌سیم تعریف می‌شود. (Quality of Service) برای این نوع لایه‌های MAC مهم در نظر گرفته نمی‌شود و متناسب با نیاز ما است. استاندارد آن 802.11b تنظیم می‌شود.

خط آخر نیز Remote Station Manager که لیستی حالت‌های per-remote-station را نگه می‌دارد را تنظیم می‌کند. رشته اول نوع remote station manager را تعیین می‌کند. در ادامه نیز مشخصات لایه فیزیکی به آن اضافه می‌شوند.

بخش هشتم:

```
wifiMac.SetType ("ns3::AdhocWifiMac");  
NetDeviceContainer Devices;  
Devices = wifi.Install (wifiPhy, wifiMac, Nodes);
```

نوع لایه MAC که در قسمت قبل تعریف شد این جا تعیین می‌شود. (Ad-hoc) سپس device های شبکه تعریف می‌شوند و در شبکه wifi ای که لایه فیزیکی و نودهای آن و لایه MAC برای آن تعریف شده نصب می‌شوند.

بخش نهم:

```
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::GridPositionAllocator", "MinX", DoubleValue (0.0),
"MinY", DoubleValue (0.0), "DeltaX", DoubleValue (5.0),
"DeltaY", DoubleValue (10.0), "GridWidth", UIntegerValue (3),
"LayoutType", StringValue ("RowFirst"));
Ptr<ListPositionAllocator> positionAlloc = CreateObject <ListPositionAllocator>();
positionAlloc ->Add(Vector(0, 0, 0));
positionAlloc ->Add(Vector(0, 500, 0));
mobility.SetPositionAllocator(positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (Nodes);
```

کلاس mobilityHelper برای تعیین موقعیت نودها و مدل های حرکت کردن آن ها استفاده می شود. سپس مشخصات دستگاه مختصات داده می شود. کمینه مولفه اول و دوم در صفحه دو بعدی و جهش های آن و سپس جای نودها با بردارهایی تعیین می شود و مدل موبیلیتی به این نودها اضافه می شود و این مدل نقطه ها به همراه موبیلیتی آن ها بر روی نودها فعال می شود.

بخش دهم:

```
OlsrHelper olsr;
Ipv4StaticRoutingHelper staticRouting;
Ipv4ListRoutingHelper list;
list.Add (staticRouting, 0);
list.Add (olsr, 10);
InternetStackHelper internet;
internet.SetRoutingHelper (list); // has effect on the next Install ()
internet.Install (Nodes);
```

ابتدا از کلاس OLSR یک routing تعریف می کنیم که به نودهایمان اضافه کنیم. سپس اشیا static routing و list routing با IPv4 اضافه می شوند. به متغیر لیست باید یک routing از شی staticRoutingHelper اضافه شود و عدد دوم اولویت آن را مشخص می کند. سپس olsr اضافه می شود. کلاس بعدی که به نام internet تعریف عملکرد IP/TCP/UDP را به نودهای موجود اضافه می کند. سپس روش routing به اینترنت اضافه می شود و بر روی نودها نصب می شود.

بخش یازدهم:

```
Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("192.168.1.0", "255.255.255.0");
Ipv4InterfaceContainer i;
i = ipv4.Assign (Devices);
```

در این قسمت به نودها آدرس IP اضافه می کنیم.(به کمک شی ipv4)

بخش دوازدهم:

```
PacketSinkHelper UDPSink ("ns3::UdpSocketFactory", InetSocketAddress(Ipv4Address::GetAny (),
UDPport));
ApplicationContainer App;
NodeContainer SourceNode = NodeContainer (Nodes.Get (0));
NodeContainer SinkNode = NodeContainer (Nodes.Get (5));
//To Create a UDP packet sink
App = UDPSink.Install(SinkNode);
App.Start (Seconds (30.0));
App.Stop (Seconds (60.0));
//To Create a UDP packet source
OnOffHelper UDPsource ("ns3::UdpSocketFactory", InetSocketAddress(i.GetAddress(5,0), UDPport));
UDPsource.SetAttribute ("OnTime", RandomVariableValue (ConstantVariable (1)));
UDPsource.SetAttribute ("OffTime", RandomVariableValue (ConstantVariable (0)));
App = UDPsource.Install(SourceNode);
App.Start (Seconds (30.0));
App.Stop (Seconds (60.0));
```

خط اول برای instantiate کردن راحت تر اپلیکیشن ها بر روی نودها نوشته شده است. سپس شیء اپلیکیشن ساخته می شود. دو node container هم تعریف می شود. سپس برنامه منبع و مقصد بر روی آن ها نصب می شود و مدت زمان اجرا هم تعیین می شود. (برنامه منبع همان on-off application بود).

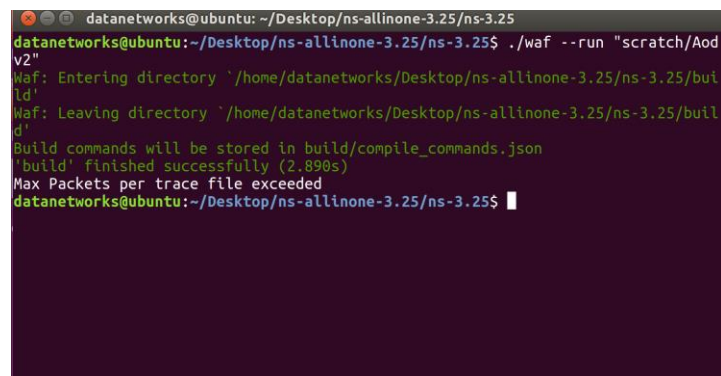
بخش سیزدهم:

```
AnimationInterface anim (animFile);
Ptr<OutputStreamWrapper> routingStream = Create<OutputStreamWrapper> ("moh-aodv-routes",
std::ios::out);
aodv.PrintRoutingTableAllEvery (Seconds (1), routingStream);
```

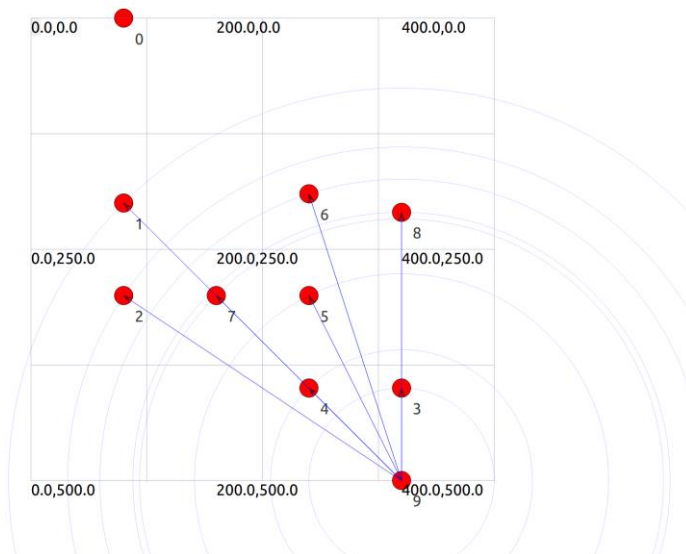
فایل انیمیشن ساخته می شود. (برای استفاده در netanim) و در خط آخر تعیین می شود جدول routing همه نودها در هر ثانیه چاپ شود.

۴. با توجه به error هایی که مشاهده شد ۴ کتابخانه include شدند. Main به برنامه اضافه شد. هم چنین بقیه نودها در قسمت ۹ اضافه شدند. در آخر کد هم چند خط که از تمرین قبل یاد گرفته بودیم اضافه شد. به علاوه قطع شدن شبیه سازی در قسمت ۱۲.

نتیجه در تصویر زیر مشاهده می شود:

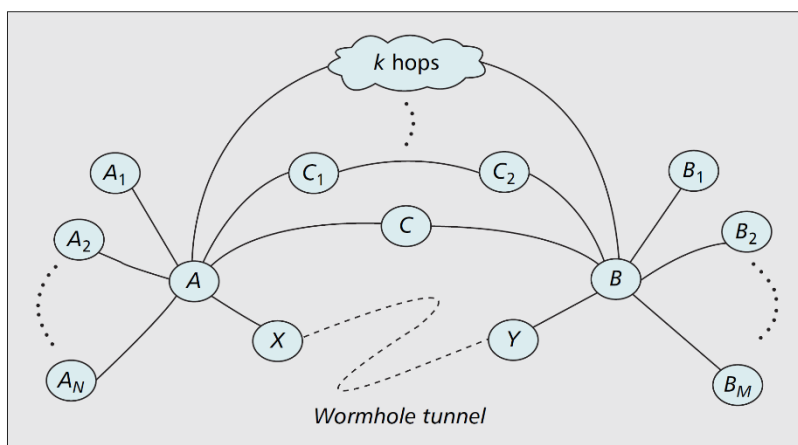


```
datanetworks@ubuntu: ~/Desktop/ns-allinone-3.25/ns-3.25
datanetworks@ubuntu:~/Desktop/ns-allinone-3.25/ns-3.25$ ./waf --run "scratch/Aodv2"
Waf: Entering directory `/home/datanetworks/Desktop/ns-allinone-3.25/ns-3.25/build'
Waf: Leaving directory `/home/datanetworks/Desktop/ns-allinone-3.25/ns-3.25/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (2.890s)
Max Packets per trace file exceeded
datanetworks@ubuntu:~/Desktop/ns-allinone-3.25/ns-3.25$
```

2. Wormhole Attack in AODV Routing Protocol

۵. هدف این حمله مختل کردن عملیات routing توسط نودهای شبکه ad-hoc است. این حمله به این صورت است که attacker در دو نقطه از شبکه نودهایی را قرار می‌دهد و بسته‌هایی که از یک نود خاص می‌رسد را از نود اول خود گرفته و از طریق یک لینک با تاخیر بسیار کم به نود دوم خود ارسال می‌کند و به مقصد می‌رساند. (در حالی که در حالت عادی مسیر طولانی‌تری برای رسیدن پیام به مقصد باید طی می‌شد.) به این ترتیب نود مبدأ فکر می‌کند که واقعاً به نود مقصد نزدیک است و از این به بعد بسته‌های نود مبدأ توسط تونل کرم‌چاله یا wormhole link فرستاده می‌شوند و طبیعتاً attacker قادر خواهد بود بسته‌هایی را به طور دلخواه انتخاب کند و دور بریزد و ارتباط دو نود مبدأ و مقصد را مختل کند. (شکل زیر)



در این شکل نودهای X و Y نودهایی هستند که توسط attacker قرار داده شده‌اند و مسیر خط چین همان wormhole tunnel است که تاخیر بسیار کمی دارد و الگوریتم‌های routing را فریب می‌دهد.

برای مثال اثر آن را بر روی پروتکل AODV و OLSR به طور خلاصه بررسی می‌کنیم. اگر بسته‌های RREQ را در الگوریتم AODV مورد هدف قرار دهد نودهای نزدیک نود حمله‌کننده هیچ مقصدی با فاصله بیش از ۲ hop پیدا نمی‌کنند. به این ترتیب ارتباط نودها را با دور ریختن بسته‌های آن‌ها مختل می‌کند یا اطلاعات ارسالی را می‌دزدد. اگر hello message ها را در الگوریتم OLSR مورد هدف قرار دهد نودها همسایه‌های

خود را اشتباه تشخیص می‌دهند و به تبع آن MPR های اشتباهی نیز انتخاب خواهند شد و از آن جایی که این اطلاعات و جدول MPR selectors به کل شبکه broadcast می‌شود، تمام شبکه اشتباه شناخته خواهد شد و کل ارتباط از بین می‌رود.

۶. روش‌های متعددی برای تشخیص و جلوگیری از wormhole attack ها وجود دارد. یک دسته روش‌ها از packet leashes استفاده می‌کنند.

یک حالت آن Geographical leashes است که از ملزومات آن سنکرون بودن نودهای شبکه تا حد نسبتاً خوب و قابلیت مکان‌یابی نودها می‌باشد که ما را به محدودیت‌های GPS محدود می‌کند. روش این است که زمان و مکان مبدأ ارسال بسته در آن نوشته شود و گیرنده مدت زمان طی شدن مسیر بسته را اندازه بگیرد و با فاصله‌ای که می‌داند تشخیص دهد بسته مدت زمان معقولی در راه بوده یا خیر و حمله را تشخیص دهد. یک حالت دیگر temporal leashes است که در آن فقط زمان ارسال و یک زمان expire که اگر بسته بعد از رسید نباید پذیرفته شود را در بسته گذاشته می‌گذارد. گیرنده فرض می‌کند بسته با سرعت منتقل می‌شود و با جمع کردن زمان ارسال و خطای سنکرون بودن clk خود می‌بیند که زمان expire گذشته یا نه و در مورد وجود یا عدم وجود حمله تصمیم می‌گیرد. (رابطه زیر)

$$t_e = t_s + \frac{L}{c} - \Delta = t_r - \Delta$$

در این روش سنکرون بودن بسیار زیادی لازم است (از مرتبه نانو ثانیه) و بنابراین تقریباً غیر قابل اجرا است. یک روش دیگر استفاده از آنتن‌های جهتی است که بین هر دو نود کلید خاصی برای رمزنگاری وجود دارد و یافتن همسایه‌ها با فرستادن hello messages توسط آنتن‌ها در جهت‌های مختلف انجام می‌شود.

یک روش بسیار مناسب که در مقاله پیوست شده در فایل تمرین بحث شده برای تشخیص حمله در پروتکل OLSR است. مبنای آن تشخیص لینک‌های مشکوک در مرحله اول و سپس بررسی wormhole tunnel بودن آن لینک‌ها است. یک ویژگی این نوع حمله این است که latency بسته‌هایی که از این مسیر می‌روند نسبت به مسیرهای معمولی است. علت این است که تعداد زیادی از بسته‌هایی که باید از مسیرهای multi hop عبور می‌کرده‌اند، به وسیله این تونل هدایت می‌شوند. بنابراین بار این مسیر زیاد شده و تاخیر صف‌های آن را افزایش می‌دهد. اما باید توجه کرد این شرط کافی برای wormhole tunnel بودن یک لینک نیست زیرا شرایط ازدحام و پردازش نودها نیز ممکن است همین نتایج را بدهد. در این روش لینک‌هایی که این تاخیر بیشتر را دارند اصطلاحاً به تونل بودن برای حمله «مظنون» می‌شوند.

روش تشخیص: پس از هر N ارسال hello های معمولی یک نوع hello جدید ارسال می‌شود که پس از رسیدن به مقصد نیاز به reply دارد. وقتی این بسته فرستاده شد یک expiration time نیز در نظر

گرفته می‌شود. وقتی یک گیرنده این بسته را بگیرد برای شلوغ نشدن شبکه آدرس فرستنده آن را به همراه مدت زمانی که برای جواب آن بسته خاص صبر می‌کند تا جواب همه `hello_req` های رسیده را به صورت `piggybacked` ارسال کند ذخیره می‌کند. هر نودی اگر `hello_rep` مربوط به خودش را بگیرد و مدت زمان صبر کردن آن را با زمان رسیدن با هم در نظر بگیرد و با زمان `timeout` خود مقایسه کند می‌تواند بفهمد که لینکی که از آن `hello_rep` آمده مظنون است یا خیر. اگر مظنون نبود `proven` و اگر بود `suspicious` تلقی می‌شود. حال برای بررسی لینک‌های مظنون، از بسته‌های جدید `probing` و `ack` آن‌ها استفاده می‌کنیم. (در این مرحله به خاطر اهمیت آن‌ها، آن‌ها را رمز شده نیز می‌فرستیم). هدف این است که ببینیم نود آن سر لینک مظنون چه احساسی نسبت به این نود مبدأ دارد و اگر هر دو لینک را مظنون ببینند نتیجه می‌شود لینک یک تونل `wormhole` است.

پس از دریافت `probing packet` اگر از اعتبار نود مبدأ اطلاعی داشت که در بسته `ack` آن را می‌فرستد. در غیر این صورت یک `hello_req` می‌فرستد و مبدأ فوراً در جواب آن یک `hello_rep` می‌فرستد و نود مقصد در مورد اعتبار نود مبدأ تصمیم می‌گیرد و در `ack` نتیجه را اعلام می‌کند. اگر هر دو نظرشان `proven` باشد لینک معتبر شناخته می‌شود. اگر هر دو `suspicious` تشخیص دهند لینک بسته می‌شود و از آن ارسال نمی‌شود. اگر یکی `proven` و یکی مظنون باشد عملیات تایید تونل کرمچاله دوباره تکرار شده و اگر بار دیگر حتی یکی از دو طرف مظنون تشخیص بدهد لینک بسته شده و به عنوان حمله تشخیص داده می‌شود.

3. Blackhole Attack in AODV Routing Protocol

۷. در این حمله، یک نود که اصطلاحاً آن را malicious می‌گوییم خودش را برای نود هدف به عنوان نودی معرفی می‌کند که کمترین فاصله را تا مقصد آن نود دارد. (بدون این که جدول routing اش را نگاه کند.) در پروتکل‌هایی که بر flooding استوارند پاسخ این نود زودتر از نودی که واقعاً کمترین مسیر را تا مقصد دارد به مبدأ می‌رسد و نود مبدأ RREP های دیگر را در نظر نمی‌گیرد چون این مسیر کمترین فاصله را دارد و مسیری جعلی ساخته می‌شود که حمله‌کننده می‌تواند بسته‌ها را دور بریزد یا به مقصدی که مد نظر خودش است بفرستد.

این نوع حمله در AODV می‌تواند به دو صورت داخلی یا خارجی باشد. در نوع داخلی به محض این که شناس وارد شدن به شبکه پیدا شود از طریق روش بالا عضوی از یک active route می‌شود و می‌تواند با ارسال داده حمله را آغاز کند. داخلی بودن به این دلیل است که جزوی از نودهای فعال داخلی شده است. تشخیص آن هم دشوار است. نوع خارجی همان است که در پاراگراف بالایی به آن اشاره شد.

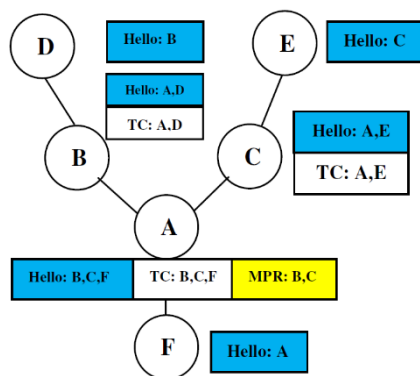


Figure 3: (a). OLSR without Blackhole

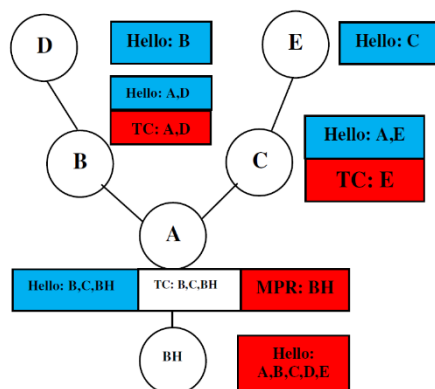


Figure 3: (b). OLSR with Blackhole

این حمله در OLSR به این صورت است که نود attacker تلاش می‌کند خودش به عنوان MPR انتخاب شود. به این صورت که در hello message ارسالی به نودی که نزدیک آن است، اعلام می‌کند به تمام همسایه‌ها و همسایه‌های آن همسایه‌ها دسترسی دارد و در نتیجه منطقی است که نود هدف آن را به عنوان تنها MPR انتخاب کند و از آن جا که MPR selector های نودهای همسایه را خراب کرده، TC message هایی که باید به صورت broadcast در شبکه پخش شوند نیز خراب می‌شوند و عملکرد شبکه مختل می‌شود. هم‌چنین با مکان ممتازی که به عنوان MPR پیدا می‌کند می‌تواند حمله DoS یا Denial of Service انجام دهد و بسته‌ها را از نود هدف انتقال ندهد.

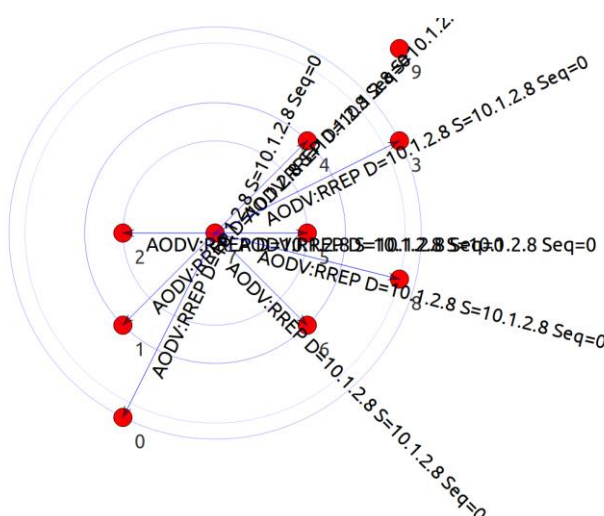
۸. از تفاوت‌های آن‌ها می‌توان به نحوه حمله اشاره کرد که در حمله کرم‌چاله دو نود حضور دارند و یک تونل با تاخیر بسیار کم ایجاد می‌کنند حال آن که در حمله سیاه‌چاله فقط یک نود حضور دارد و حمله را انجام می‌دهد. به نظر می‌رسد

در حمله کرم‌چاله حتماً جنس حمله internal است ولی در حمله سیاه‌چاله می‌توان external هم بود همان‌طور که توضیح داده شد. جنس حمله یکی DoS است (سیاه‌چاله) و جنس حمله کرم‌چاله مختل کردن ارتباط یا دزدیدن بسته‌ها است. یک تفاوت دیگر این است که در روش کرم‌چاله همسایه‌های اشتباهی به نودها معرفی می‌شوند و در روش سیاه‌چاله خود نود مهاجم خود را به عنوان همسایه جعل می‌کند.

۹. در حملات فعال هدف اختلال ایجاد کردن در عملکرد شبکه، دزدیدن اطلاعات مهم و یا نابود کردن اطلاعات در حال ارسال شدن است. این اهداف با در دست گرفتن یک نود malicious در داخل شبکه به سادگی قابل انجام است و می‌تواند با ارسال بسته‌های ساختگی عملکرد را دچار اختلال کند یا مثل حمله سیاه‌چاله DoS داشته باشیم. در حالت غیرفعال یا منفعل یا passive حمله هیچ‌یک از اهداف بالا را دنبال نمی‌کند و فقط به شبکه گوش می‌دهد تا از آن اطلاعات بدست آورد و بفهمد در آن چه خبر است، نحوه ارتباط نودها با یکدیگر، جای‌گیری آن‌ها و موقعیتشان را به دست آورد تا هنگام حمله اصلی اطلاعات کافی برای دزدیدن بسته‌ها یا انجام انواع حمله‌ها داشته باشد.

۱۰. نتیجه اجرای کد در تصویر زیر مشاهده می‌شود.

```
datanetworks@ubuntu: ~/Desktop/ns-allinone-3.25/ns-3.25
datanetworks@ubuntu:~/Desktop/ns-allinone-3.25/ns-3.25$ ./waf --run "scratch/blackhole"
Waf: Entering directory `/home/datanetworks/Desktop/ns-allinone-3.25/ns-3.25/build'
Waf: Leaving directory `/home/datanetworks/Desktop/ns-allinone-3.25/ns-3.25/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (4.026s)
Launching Blackhole Attack! Packet dropped . . .
Launching Blackhole Attack! Packet dropped . . .
datanetworks@ubuntu:~/Desktop/ns-allinone-3.25/ns-3.25$
```



۱۱. در صورتی که کتابخانه‌هایی که در ns3 از قبل موجود نبوده‌اند را نیاز داشته باشیم باید ابتدا آن‌ها را نصب کنیم. در کد blackhole نیز myapp.h استفاده شده و برای آن باید patch را نصب کنیم. در قسمت‌های قبل به patch نیاز نداشتیم چون همه کتابخانه‌های مورد نیازمان در ns3 موجود بودند.

۱۲. راه‌های مختلفی برای جلوگیری از این حمله وجود دارد که به برخی از آن‌ها اشاره می‌کنیم. (برخی از روش‌های اشاره شده در مقاله)

یکی از این روش‌ها DSR اصلاح شده است. در این روش علاوه بر کوتاه‌ترین مسیری که در ابتدا پیدا می‌شود مسیر کوتاهی که در اولویت دوم قرار دارد نیز در نظر گرفته می‌شود و پکت‌های ارسالی و دریافت شده هر دو اندازه‌گیری می‌شوند و اگر در مسیر اول تفاوت فاحشی وجود داشته باشد می‌توان نتیجه گرفت یک نود malicious در حال حمله سیاه‌چاله است. در نتیجه نودها دیگر برای آن نود چیزی نمی‌فرستند.

یک روش دیگر RSSRE است که در آن ارزیابی اعتبار بر اساس کوریلیشن بین نودهاست. مکانیزم آن ارتقای همکاری اعضای خوشه‌ها برای اجرای بهتر routing در حضور نودهای malicious است.

روشی که مقاله ارائه داده نیز برای پروتکل AODV می‌باشد و به این صورت است که نودها به خوشه‌هایی تقسیم می‌شوند که هر خوشه یک سر خوشه دارد و ارتباط بین خوشه‌ها از طریق این سر خوشه‌ها است. این سرخوشه‌ها نیز تصادفی انتخاب می‌شوند. تعدادی check-point نیز در شبکه کار گذاشته می‌شود تا نرخ پکت‌های عبوری را بررسی کند.

دو حالتی که مبدأ و مقصد در یک خوشه باشند و نباشند را جدا بررسی می‌کنیم. اگر باشند پس از RREQ یک RREP از مقصد برمی‌گردد و سایر نودها باید با سرخوشه یک مرتبه ping کنند و نودی که این کار را نکند مظنون شناخته می‌شود.

در حالتی که در دو خوشه جدا باشند، check-point ها نرخ عبور را کنترل می‌کنند و اگر ناهماهنگی مشاهده شود به جستجوی نود مظنون در این مسیر بین خوشه‌ای می‌پردازند.

4. Other Kinds of Attacks

Grey Hole Attacks

در این حمله مهاجم با موافقت برای forwarding بسته‌ها شبکه را به اشتباه می‌اندازد به این صورت که رفتار نرمال از خود بروز می‌دهد و در جواب RREQ یک RREP مناسب می‌فرستد ولی وقتی بسته‌ها به دستش رسید آن‌ها را دور می‌ریزد! در واقع با این کار سرویس را ارائه نمی‌دهد و حمله از نوع DoS می‌شود. در واقع در عین forward کردن بسته‌ها را دور می‌ریزد. به خاطر رفتار متناوبی آن تشخیص آن بسیار دشوار است. تفاوت آن با سیاه‌چاله در این است که پیغام جعلی نمی‌فرستد و فقط وظیفه‌اش را انجام نمی‌دهد و تفاوت آن با کرم‌چاله نیز این است که در این روش routing شبکه را به اشتباه نمی‌اندازد و فقط بسته‌ها را دور می‌ریزد و ضمناً حمله با یک نود تنها نیز انجام می‌شود.

Hello Flood Attacks

این حمله اجرای ساده ولی خسارت سنگین دارد. می‌تواند هم از RREQ و هم Flooding استفاده کند. در حالت اول بسته‌های RREQ را در شبکه flood می‌کند و resource زیادی از شبکه را با این بسته‌ها مصرف می‌کند. در واقع می‌تواند با انتخاب IP Address هایی که در شبکه وجود ندارند این کار را بکند و هیچ نودی نمی‌تواند به این درخواست پاسخ بدهد. در روش data flooding بین همه نودها مسیر ایجاد می‌کند و داده‌های نامربوط و بی‌استفاده در شبکه می‌ریزد و باعث ایجاد ازدحام می‌شود.

Selfish Node Attack

این وضعیت زمانی اتفاق می‌افتد که یک نود با سایر نودها همکاری نکند و هدف آن فقط حفظ resource های خودش باشد. (برای همین به آن نود خودخواه می‌گویند). این اتفاق باعث اختلال در ترافیک شبکه می‌شود. این رد کردن forwarding می‌تواند به دو صورت انجام شود. یکی اعلام مسیر دیگر برای بسته که واقعاً وجود ندارد یا کمتر بهینه است و یکی دور ریختن بسته‌ها. این نود هم‌چنین می‌تواند در مواقعی که نیاز دارد از شبکه استفاده کند و در مواقع دیگر «ساکت» باشد و به این ترتیب از دید شبکه پنهان می‌شود.

Jellyfish Attack

در این حمله مهاجم تاخیرهای ناخواسته را به شبکه تحمیل می‌کند. ابتدا دسترسی به شبکه پیدا می‌کند و وقتی عضوی از آن شد به بسته‌هایی که می‌گیرد عمداً تاخیر اضافه می‌کند و این تاخیرها در ادامه در کل شبکه منتشر می‌شوند. نتیجه آن هم زیاد شدن تاخیر end to end و هم‌چنین jitter خواهد بود که هر دو عملکرد شبکه را به شدت کاهش می‌دهند.