ANÁLISIS COMPARATIVO DEL DESEMPEÑO DE ALGORITMOS RSA Y RWA SOBRE UNA RED ÓPTICA BASADA EN LA TOPOLOGÍA NSFNET

Andrés Felipe Sevilla Majin
Edward Camilo Zúñiga Quisoboní

Universidad del Cauca

Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Telecomunicaciones
Grupo I+D Nuevas Tecnologías en Telecomunicaciones - GNTT
Popayán, Abril de 2017

ANÁLISIS COMPARATIVO DEL DESEMPEÑO DE ALGORITMOS RSA Y RWA SOBRE UNA RED ÓPTICA BASADA EN LA TOPOLOGÍA NSFNET

ANEXOS A, B, C

Andrés Felipe Sevilla Majin
Edward Camilo Zúñiga Quisoboní

Director: Ph.D. Ing. José Giovanny López Perafán

Universidad del Cauca

Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Telecomunicaciones
Grupo I+D Nuevas Tecnologías en Telecomunicaciones - GNTT
Popayán, Abril de 2017

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

ANEXO A. TEORÍA DE GRAFOS

Un grafo es una estructura matemática que consta de nodos y conexiones llamadas aristas estas se encuentran conectadas en sus dos extremos a nodos o posiblemente al mismo nodo en los dos extremos [2].

A.1 Definiciones básicas

A.1.1 Grafo dirigido

Un grafo dirigido en su definición es un par $G = (V, A)$, donde V es un conjunto finito de nodos y A es un conjunto de pares ordenados de nodos llamados aristas como se muestra en la Figura A.1.

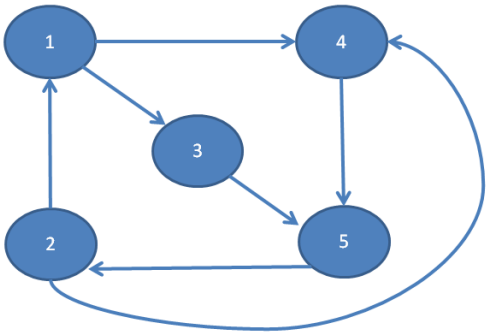


Figura 0.1 Grafo dirigido.

A.1.2 Grafo no dirigido

Es un par $G = (V, A)$, donde V es un conjunto finito de nodos y A es un conjunto de pares no ordenados de nodos, como se observa en la Figura A.2.

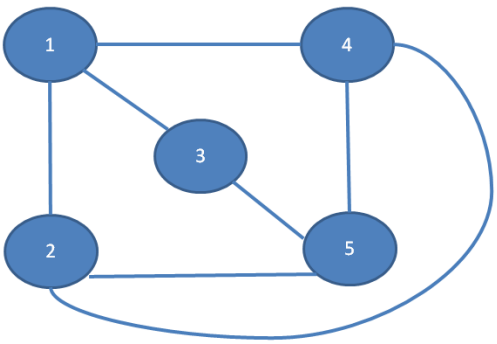


Figura 0.2 Grafo no dirigido

Si {*u,v*} es una arista de G, se dice que el nodo v es adyacente a u. Esta relación es simétrica.

A.1.3 Grafo simple y conexo

En un grafo simple no hay dos aristas que unan el mismo para de nodos. Si un grafo no es simple se le conoce como multigrafo.

Dos nodos *u,v* de un grafo son conexos, o *u* es conexo con *v*, si existe un camino que empieza en u y termina en v.se dice que un grafo es conexo si cualquier par de nodos es conexo [3].

A.1.4 Grado

Para todo nodo v, el grado de entrada es el número de aristas que inciden en v, grado de salida es el número de aristas que parten de v, por ende el grado es la suma de los grados anteriores [3].

A.1.5 Camino simple y ciclo

En un camino simple todos sus nodos son diferentes, con excepción quizás del primero y último.

Por otro lado, un ciclo es un camino simple $(V_0, V_1, \dots, V_k)$ donde los nodos inicial y final coinciden y contienen al menos una arista [3].

A.1.6 Longitud del camino

Es el número de aristas que forman un ciclo.

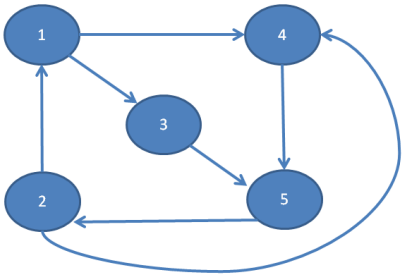
A.2 Representación de un grafo

Existen tres formas de representar un grafo: mediante la matriz de adyacencias, matriz de incidencias, lista de adyacencias y lista de incidencias. La estructura a elegir dependerá de las características del grafo y el algoritmo a utilizar es su manipulación. En este trabajo de grado se utilizó la matriz de adyacencias, la cual se explica a continuación.

A.2.1 Matriz de adyacencias

Sea un grafo G con nodos $n\,\{V_i\}_{i=1}^{n}$. Es la matriz de orden $n * n,\ A = \lfloor a_{ij} \rfloor$ tal que a_{ij} es igual al número de aristas del vértice v_i al v_j [4]. Esta representación es de rápido acceso pero su desventaja radica en el consumo elevado de memoria.

La representación del grafo se muestra en la Figura A.3, esta es una matriz cuadrada de tamaño V^2, donde *V* es el número de nodos.



	1	2	3	4	5
1	0	0	1	1	0
2	1	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	1
5	0	1	0	0	0

Figura 0.3 Matriz de adyacencias

A.3. Grafos ponderados

Un grafo $G = (V, A)$ se dice que es ponderaod si tiene asociado una fiuncion $W: A \rightarrow R$, conocida como función de ponderación. El valor de cada arista es determinado por lo nodos $v_i \; y \; v_j$ se conoce como el peso de la arista y se denota por w_{ij}. También se le llama peso del camino, a la suma de los pesos de las aristas que lo integran. Si el peso es máximo entre dos nodos se le conoce como camino crítico.

A.3.1 Algoritmo de Dijkstra

Este algoritmo básicamente permite determinar la ruta más corta que une los nodos de origen y destino de un grafo. El algoritmo de Dijkstra utiliza un tipo de estructuras de colas llamado cola de prioridad.
El funcionamiento del algoritmo de Dijkstra comienza marcando todos los nodos como no utilizados, asignándolos con distancias de valor infinito relativo. Se parte de un nodo inicial conocido, a partir de este nodo se visitan sus nodos adyacentes, como Dijkstra es una técnica voraz la cual utiliza el principio de que para que un camino sea optimo, todos los caminos contenidos sean óptimos luego entre todos los vértices adyacentes se busca el que se encuentre más cerca a nuestro nodo origen, el cual es tomado como punto intermedio y se evalúa si se puede llegar más rápido desde este nodo a los demás. Después se escoge el siguiente más cercano (con distancias actualizadas) y se repite el proceso.

Este algoritmo es utilizado en redes conmutación óptica de ráfagas (*OBS, Optical Burst Switching*) como la estudiada en este trabajo de grado, se le asigna un costo (distancia) y de esta manera algunos protocolos lo utilizan para encontrar la ruta más corta.

ANEXO B. ALGORITMOS

B.1 Algoritmo para la generación y envío de paquetes.

Se generan paquetes por intervalos de tiempo, a los cuales se les asigna atributos como su dirección origen, dirección de destino y longitud. La dirección de destino se obtiene aleatoriamente mediante una función uniforme, encargada de escoger un número entre cero y el número de nodos de la red. Además, la prioridad (o tipo de servicio) se obtiene aleatoriamente de igual manera con tres valores posibles (0 ,1 y 2).

```
if (msg == generatePacket)
   {
     // Manejo y envio del paquete
      int destAddress = destAddresses[intuniform(0, destAddresses.size()-1)];
      //int destAddress = 1;

      char pkname[40];
      sprintf(pkname,"to-%d-#%ld", destAddress, pkCounter++);
      EV << "generating packet " << pkname << endl;

      Packet *pkt = new Packet(pkname);
      pkt->setByteLength(packetLengthBytes->longValue());
      pkt->setSrcAddr(myAddress);
      pkt->setDestAddr(destAddress);

      // Se establece la prioridad
      int type = intuniform(0,2);
      pkt->setPriority(type);

      send(pkt,"out");
      scheduleAt(simTime() + sendIATime->doubleValue(), generatePacket);
      if (ev.isGUI()) bubble("Generating packet...");
   }
```

B.2 Algoritmo para la obtención de reglas por dirección de destino y prioridad de los paquetes.

Este algoritmo se encarga de obtener el archivo correspondiente a las reglas y comprara cada línea usando la clase "**Classifier_Rules**", la cual verifica si la dirección y prioridad del paquete son compatibles con las reglas consignadas en el archivo.

```
 if (numOuts != 0){

      //Almacena la información proveniente de Clasifier_Rules
      rules = (Classifier_Rules*)calloc(numOuts,sizeof(Classifier_Rules)); //se usa calloc
para reservar memoria, de forma dinámica
      //lee una línea cada vez y crea la regla asociada para cada cola.
      char *line = (char*)calloc(1500,sizeof(char)); //Debido a esto, el máximo número
de caracteres por línea es 1500.
      int i=0;

      //abre el archivo Rules.dat y lo almacena en rulesFile
      const char *rulesFile = par("rules");

      //Si rules está vacio, se muestra un mensaje de error y para la simulación.
      if(strlen(rulesFile) == 0){
         opp_error("El archivo de reglas no se encuentra definido");
      }
      // abre el archivo Rules.Dat en *ruleFile
      FILE *ruleFile = fopen(rulesFile,"r"); // r = leer el archivo... w=escribir...d=Borrar

      if(ruleFile != NULL){
         //fgets lee hasta que se encuentra un carácter \n
        while(fgets(line,1500,ruleFile) != NULL){
          if(strcmp(line,"\n") != 0 && line[0] != '#'){ //Ignora los comentarios (líneas que
empiezan con #)
             //toma "line" como parámetro de entrada
             //para realizar la comparación de los parámetros
             rules[i] = Classifier_Rules(line);
            i++;
          }
        }
      }
      else{
         opp_error("No es posible abrir el archivo de reglas");
      }

      fclose(ruleFile);
      // i debe ser igual a numQueues. Si no lo es, seguramente hay un error.
      if(!(i == numOuts)){
         printf("(Classifier_Rules) Aviso: El despachador de reglas no coincide con las
colas de los modulos.\n");
      }
```

```
      free(line);
   }
}
```

B.3 Algoritmo para la clasificación de paquetes según las reglas.

Este algoritmo se encarga de verificar que las reglas y características del paquete coincidan, si es así se retorne un valor de tipo booleano con verdadero para que el paquete sea enviado a la primera compuerta donde coincide la ráfaga.

```
while(token != NULL){
       if(strcmp(token,"destAddr") == 0){ //Si la regla tiene la dirección de destino?

        //El siguiente token es una dirección de destino, si no lo es, genera un error.
        token = strtok(NULL," \n");
        if(token != NULL){
           destAddr = atoi(token);
           isSet[0] = true;
        }
        else{
         throw runtime_error("No se puede parsear el valor de la direccion de destino en
la regla");
        }
      }
     else if(strcmp(token,"priority") == 0){ //La regla tiene un tipo de servicio?
        token = strtok(NULL," \n");
        if(token != NULL){
            priority = atoi(token);
              isSet[1] = true;
        }
        else{
           throw runtime_error("No es posible parsear el valor de la prioridad en la
regla");
        }
      }
     token = strtok(NULL," ");
   }
}
```

B.4 Algoritmo para establecer el criterio de ensamble. Este algoritmo recibe los paquetes y los almacena en una cola mediante tres criterios: número máximo de paquetes, el tamaño en Bytes y el tiempo de ensamblaje. Cuando alguno de los tres criterios se cumpla, se ensambla la ráfaga con el método "**assemblyBurst**".

```
// Comprobar si el busrtifier está vacio
    if(burst.empty()) // Si lo está, inicializar el tiempo máximo y la cola
    {
       scheduleAt(simTime() + maxTime, maxTime_msg ); //Establece el tiempo máximo
       //Registrar el momento en el que llega el primer paquete
         firstPacket_t = simTime();
    }

    else if(desborde && !addLastPacket)
    { // Entra si se necesita ensamblar la ráfaga antes de que el paquete sea insertado en la cola
       if(burst.empty())opp_error("No es posible ensamblar una ráfaga con una cola vacía");

       //Ensamblar la ráfaga e iniciar contadores
       assemblyBurst();

       if(maxTime_msg->isScheduled())cancelEvent(maxTime_msg); //Se cancela el tiempo maximo y se programa uno nuevo
       scheduleAt(simTime() + maxTime, maxTime_msg);
       firstPacket_t = simTime();

       //Calcular si la sobrecarga se da con el primer paquete
              desborde = false;

       if(((burstBits + pqt->getBitLength()+ tamHeaderPacketBits)> maxSizeBits)) desborde=true;
    }
    //Insertar el paquete actual en la cola
    timeAssembled.collect(simTime()-firstPacket_t);
    burst.insert(pqt);
    burstBits += pqt->getBitLength() + tamHeaderPacketBits;
    numPacketsInBurst++;
    //Si la sobrecarga no está habilitada, pero ocurre sobrecarga cuando se inserta el primer paquete, se genera un error.
```

```
    if(desborde && !addLastPacket)
       opp_error("La sobrecarga se generó insertando el primer mensaje y los requerimientos no permiten sobrecarga(overflowLastPacket es falso)");
    //Si ocurre la sobrecarga o se llena al máximo número de paquetes se debe ensamblar la ráfaga.
    if((desborde || numPacketsInBurst == numPackets) || burstBits ==maxSizeBits)
    {
       assemblyBurst();
       if(maxTime_msg->isScheduled()) cancelEvent(maxTime_msg);
    }
  }
```

B.5 Algoritmo para desencapsulamiento de la ráfaga.

Este algoritmo es el encargado de recibir el inicio de la ráfaga almacenándolo en una lista, una vez recibido el "**endBCP**" se busca en la lista el identificador que le corresponde, abstrayendo la rafaga de la lista y desencapsulándola.

```
while(!iter_list.end()){
    item = (ScheduleBurst*)iter_list++;
    if(item->getIdBurst() == bId){
       burst = check_and_cast <Burst *> (item->decapsulate());
       listSize -= burst->getBitLength();
       delete burstList.remove(item);
       delete burst;
       numElems--;
       numElemsVector.record(numElems);
       return;
    }
  }
```

B.6 Algoritmo para programar el envío del BCP.

Este algoritmo verifica que la ráfaga haya llegado desde el modulo "**Burstifier**", se encarga de encontrar el canal mediante el horizonte más cercano con la función "**findNearestHorizon**".

```
 if(!(msg->isSelfMessage())) // Recibe una ráfaga desde BurstAssembler (Burstifier)
  {
```

```
    //Emite el mensaje a ráfaga
    Burst *rfg = check_and_cast<Burst*>(msg);

    burstTam.record(rfg->getBitLength());
    burstRecv++;

    int wl = 0;
    int pos=0;

    wl=findNearestHorizon(); //retorna la posición del horizonte más apto para enviar la ráfaga actual

    char pkname[40];
    sprintf(pkname,"----Wavelengt--%d",wl);
    EV << "info" << pkname << endl;

    //Inserta el id de la ráfaga
    rfg->setSendId(getId()); //retorna el identificador del módulo

    BurstSenderInfo *myInfo = new BurstSenderInfo();

    myInfo->setIdBurstifier(rfg->getIdBurstifier()); //retorna el índice del vector correspondiente al módulo
    myInfo->setNumSeq(rfg->getSecNum()); //Contador de ráfagas...burstCounter este sec num sirve cuando los paquetes o ráfagas llegan desordenados, con este indicador, se puede ordenar cuando lleguen al receptor
    myInfo->setAssignedLambda(wl);

    BurstifierInfo *lInfo = (BurstifierInfo*) rfg->removeControlInfo();
    myInfo->setLabel(lInfo->getLabel()); //se define en el .ini una etiqueta de destino para cada burstifier
    delete lInfo;

    //Crea el automensaje "schedule the bcp_ini send"
    cMessage *ctrMsg= new cMessage("Sched");

    //Entra aquí si la ráfaga puede ser enviada solo cuando el canal wl se establezca como libre. (habrá mucho tiempo para enviar el bcp y esperar el tiempo de offset máximo)
    if(horizon[wl] - rfg->getMaxOffset() >= simTime()) //el = Significa que se puede programar una ráfaga al mismo tiempo que el valor del horizonte
     {
```

```
      //Se almacena en ScheduledBurst
      pos = scheduleBursts.insertBurst(rfg, horizon[wl]);

      if(pos == -1)
      { //scheduledBurst está lleno. Se decarta esta ráfaga
        delete msg;
        delete ctrMsg;
        burstDroppedByQueue++;
        return;
      }

    //Inserta la posición de la ráfaga en la cola de scheduledBurst
    myInfo->setBurstId(pos);

    //Se llenan los campos de control(este mensaje viajará a través de todos los estados del sender)
    ctrMsg->setControlInfo(myInfo);
    ctrMsg->setKind(OBS_PROGRAMAR_BCP); //Se Establece en el paso 1: Schedule BCP

    //Programar envío de BCP
    scheduleAt(horizon[wl] - rfg->getMaxOffset(), ctrMsg);

    //Actualizar el valor del horizonte
    horizon[wl] = horizon[wl] + (rfg->getBitLength()/dataRate) + guardTime;

    //Se Registra el valor del horizonte
    horizonVec[wl]->record(horizon[wl]);
    }

   else //Se puede mandar el BCP inmediatamente de esta forma la ráfaga será enviada dentro del offset máximo
    {
     //Se almacena en ScheduledBurst
     pos = scheduleBursts.insertBurst(rfg, simTime() + rfg->getMaxOffset());
     char name[40];
     sprintf(name, "--pos %d", pos);
     EV<< "info"<<name<< endl;

     if(pos == -1)
      { // la cola de ScheduleBursts queue está llena. Descartar
        delete msg;
```

```
        delete ctrMsg;
        burstDroppedByQueue++;
        return;
       }

    //Insertar la posición de la ráfaga en la cola scheduledBurst
    myInfo->setBurstId(pos);

    // LLenar los campos del mensaje de control(Este mensaje pasará a traves de
todos los estados del sender)
    ctrMsg->setControlInfo(myInfo);
    ctrMsg->setKind(OBS_PROGRAMAR_BCP);

    //Programar envío de BCP en este momento
    scheduleAt(simTime(), ctrMsg);

    //Actualizar el valor del horizonte
    horizon[wl] = simTime() + rfg->getMaxOffset() + (rfg->getBitLength()/dataRate) +
guardTime;

    //Registrar el valor del horizonte
    horizonVec[wl]->record(horizon[wl]);
    }
   }
```

B.7 Algoritmo para encontrar el horizonte más cercano.

Este algoritmo realiza un barrido del vector "**horizon**" (el cual contiene los tiempos en los cuales se encontraran libres los canales), y devuelve la posición del menor tiempo que encuentra.

```
int Sender::findNearestHorizon()
{
   int min = 0;
      int i;
      for(i=0;i<numLambdas;i++)
      {
         if(horizon[min] > horizon[i]) //devuelve el menor tiempo que encuentra
            min = i;
```

```
      }
      return min;
}
```

B.8 Algoritmo para la generación y envío de la ráfaga.

Este algoritmo se encarga de generar el BCP, enviarlo y programar el envío de la ráfaga de datos sobre el canal obtenido mediante el algoritmo explicado en B.7.

```
   // El canal de control se encuentra ocupado transmitiendo otro BCP?  ***-->BCP
   if(control_is_busy)
    {
    //Si es así, colocar el BCP actual en la cola (waitingBCP)
    waitingBCP.insert(msg);
    }
   #============================
   // Primer paso: Enviar el ini del mensaje del BCP  ***-->BCP
   //============================
   else
    {
    //Si no lo es, tomar el canal de control
     control_is_busy = true;

     //Crear el mensaje BCP
     BurstControlPacket *bcp = new BurstControlPacket("iniBCP");

     //Inicializar todos los campos del BCP
     bcp->setKind(1); //kind 1 = initial BCP
     bcp->setColorBurst(colour[info->getAssignedLambda()]); //wl

     //Incluye tiempo de llegada de la ráfaga relativo: la diferencia entre la llegada de iniBCP e iniBurst
     //Como iniBCP será enviado en ste instate, este tiempo es: sendTime(burst) - current simTime
     bcp->setDeltaArriveBurst(scheduleBursts.retrieveSendTime(info->getBurstId()) - simTime());

     //Llenar todos los campos del BCP
     bcp->setIdBurstifier(info->getIdBurstifier()); //índice o identificador del módulo burstifier al que corresponden
     bcp->setNumSeq(info->getNumSeq()); //Contador de ráfagas
```

```
    bcp->setIdSend(getId()); //índice o identificador del módulo, en este caso sender
    bcp->setLabel(info->getLabel()); //se establece en el .ini y es una etiqueta para cada burstifier
    bcp->setBurstSize(scheduleBursts.retrieveBurstSize(info->getBurstId()));
    bcp->setByteLength(BCPSize);
    bcp->setNumSaltos(numSaltos); //se fija el valor de numero de saltos con el contenido de la variable numSaltos
#==============================
   // Envio del ini BCP
#==============================
   //Enviar el BCP al canal de control(el último)
     send(bcp,"out",numLambdas);

     //Retransmitir el mensaje recibido(ctlMsg = *msg here)
     msg->setKind(OBS_PROGRAMAR_FIN_BCP);

     //Programar el envío de Schedule endBCP
     int BCPSizeInBits = BCPSize*8;
     scheduleAt(simTime()+(BCPSizeInBits/dataRate),msg);
     }
    }
#==============================
  // Segundo paso: Enviar el fin del mensaje del BCP y programar el envío de la ráfaga.  ***-->BCP
  #==============================
   else if(msg->getKind() == OBS_PROGRAMAR_FIN_BCP)
    {
     info = check_and_cast<BurstSenderInfo *>(msg->getControlInfo());
     BurstControlPacket *bcp = new BurstControlPacket("endBCP"); //mensaje //creado del BCP

     //llenar el mensaje BCP con info
     bcp->setKind(2); //kind 2 = end BCP
     bcp->setIdBurstifier(info->getIdBurstifier());    //índice o identificador del módulo burstifier al que corresponden
     bcp->setNumSeq(info->getNumSeq());             //contador de ráfagas
#==============================
     // Envio del end BCP
#==============================
     //Envia endBCP al canal de control
     send(bcp,"out",numLambdas);
```

```
      //Retransmitir el mensaje recibido (ctlMsg = *msg here)
      msg->setKind(OBS_PROGRAMAR_RAFAGA);

      //Programar envío de BurstIni
      scheduleAt(scheduleBursts.retrieveSendTime(info->getBurstId()),msg);

      control_is_busy = false; //El canal de control está libre ahora...

      //Escoger un BCP si hay alguno disponible
      if(!waitingBCP.empty())
        {
         //Colocar el BCP fuera de la cola y programarlo
         cMessage *bcp_ini = (cMessage*)waitingBCP.pop();
         scheduleAt(simTime(),bcp_ini);
        }
      }
     #=============================
     // Tercer paso: Enviar el inicio de la ráfaga óptica  ***-->BURST

    else if(msg->getKind() == OBS_PROGRAMAR_RAFAGA)
     {
      info = check_and_cast<BurstSenderInfo *>(msg->getControlInfo());
      Burst *burst= scheduleBursts.retrieveBurst(info->getBurstId());

      //Llenar los campos de la ráfaga
      burst->setName("iniBurst");
      burst->setKind(1);              //kind 1= send burst

#=============================
    // Envio del ini Burst
#=============================
      send(burst,"out",info->getAssignedLambda());

#=============================
    // Actualiza el contador de envíos
#=============================
      burstSend++;  burstSendCore++;

    //Retransmitir el mensaje recibido(ctlMsg = *msg here)
      msg->setKind(OBS_PROGRAMAR_FIN_RAFAGA);
    //Se programa el final de la ráfaga
      scheduleAt(simTime()+(burst->getBitLength()/dataRate),msg);
```

```
   }
#============================
   // Cuarto y último paso: Enviar el final de la ráfaga    ***-->BURST
   #============================
   else if(msg->getKind() == OBS_PROGRAMAR_FIN_RAFAGA)
    {
     info = check_and_cast<BurstSenderInfo *>(msg->removeControlInfo());
     Burst *burst = new Burst("endBurst");

     //Se llenan los campos de la ráfaga
     burst->setKind(2);                          //kind 2= end burst
     burst->setIdBurstifier(info->getIdBurstifier()); //índice o identificador del módulo
burstifier al que corresponden
     burst->setSecNum(info->getNumSeq());          //contador de ráfagas

#============================
     // Envio del end Burst
    #============================
     send(burst,"out",info->getAssignedLambda());

     //Borrar ráfaga de las lista programada
     scheduleBursts.removeBurst(info->getBurstId());

     //Remover la información de control y los mensajes contenedores
     delete msg;
     delete info;
   } }
```

B.9 Algoritmo para la elección de la compuerta de salida de la ráfaga en el OXC.

```
if(msg->getKind() == 1){ //Inicio de la ráfaga.

       take(msg);//toma posesión
       Burst *recvBurst = check_and_cast < Burst*> (msg);
       numSaltos = recvBurst->getNumSaltos() + 1;
       recvBurst->setNumSaltos(numSaltos);
   }
```

//Algoritmo sencillo: verifica si la compuerta de entrada; tiene una conexión programada y manda el mensaje a la compuerta de salida asignada.

```
cGate *gate = msg->getArrivalGate();

if(schedulingTable[gate->getIndex()] == -1) delete msg; //Compuerta de salida no asiganada. Se descarta la ráfaga
else
  sendDelayed(msg,OXCDelay,"out",schedulingTable[gate->getIndex()]);
```

B.10 Algoritmo para el desensamble de la ráfaga.

```
if(dynamic_cast< Burst *> (msg) == NULL){ delete msg; return; } //El paquete de control de la ráfaga no debería pasar en este punto

if(msg->getKind() == 1){ //Inicio de la ráfaga. Coloca este mensaje en la cola

  take(msg); //toma posesión
  Burst *recvBurst = check_and_cast < Burst*> (msg);
  receivedBursts.push_back(recvBurst);
  //Adiciona la ráfaga en el contador de ráfagas recibidas
  recvBursts++;

  listSize++;
  VlistSize.record(listSize);

}
else if(msg->getKind() == 2){ //Final de la ráfaga. Se busca el inicio del mensaje y se desemnsambla.
  Burst *recvBurst = check_and_cast < Burst*> (msg);
  int bId,nSeq; // valores de ID de la ráfaga

  bId = recvBurst->getIdBurstifier();
  nSeq = recvBurst->getSecNum();

  list<Burst*>::iterator i;
  Burst* actElem;
  //Busca desde el comienzo porque la ráfaga que se está buscando es probablemente una de la últimas
  for(i = receivedBursts.begin(); i != receivedBursts.end(); i++){
    actElem = *i;
    if((actElem->getIdBurstifier() == bId) && (actElem->getSecNum() == nSeq)) break; //Burst found!
  }
```

```
    if(i != receivedBursts.end()){ //Si el iterador superior no llega hasta el final...significa que la ráfaga fue encontrada.
      Burst *burstIni= check_and_cast< Burst* > (*i);
    cMessage *tempPack;
      while(burstIni->hasMessages()){ //Libera los paquetes hasta que la cola de burstIni queue esté vacia
        tempPack = burstIni->retrieveMessage();
        //TODO: Envía el paquete a un buffer intermedio por lo tanto, todos los paquetes no serían liberados a la red eléctrica al mismo tiempo.
        send(tempPack,"out");
      }
      //Limpieza
      delete msg;
      delete burstIni;
      i = receivedBursts.erase(i);

      listSize--;
      VlistSize.record(listSize);
    }
    else{
      printf("<OBS_BurstDissasembler><t=%s> Error!! burst with id=(%d,%d) not found!\n",simTime().str().c_str(),bId,nSeq);
      delete msg;
    }
  }
```

ANEXO C. RESULTADOS DE SIMULACIÓN

C.1 Resultados obtenidos de probabilidad de bloqueo en función del tráfico en la red para tiempo de simulación de 0.8 s.

Tabla B.1 Probabilidad de bloqueo vs Tráfico

Tráfico (E)	RWA				RSA			
	2 λ (100 GHz)		4 λ (200 GHz)		8 slots (100 GHz)		16 slots (200 GHz)	
	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps
1	0,388380	0,247270	0,123430	0,015190	0,093020	0,009450	0,015930	0,001540
0,9	0,386960	0,241110	0,117400	0,014540	0,088940	0,008930	0,015350	0,001430
0,8	0,386490	0,234850	0,112860	0,013510	0,084300	0,008060	0,015020	0,001340
0,7	0,387440	0,228470	0,107700	0,012530	0,080900	0,007790	0,014670	0,001220
0,6	0,377630	0,221810	0,102540	0,011360	0,077120	0,007210	0,013110	0,001120
0,5	0,377060	0,216240	0,097450	0,010780	0,072820	0,006730	0,012910	0,001140
0,4	0,376580	0,209870	0,092720	0,010060	0,069700	0,006220	0,010680	0,001020
0,3	0,367510	0,204720	0,089950	0,009610	0,066930	0,006080	0,009380	0,000907
0,2	0,367020	0,200720	0,085090	0,008760	0,063950	0,005570	0,006080	0,000610
0,1	0,356940	0,194420	0,081330	0,008380	0,060670	0,005190	0,003890	0,000305

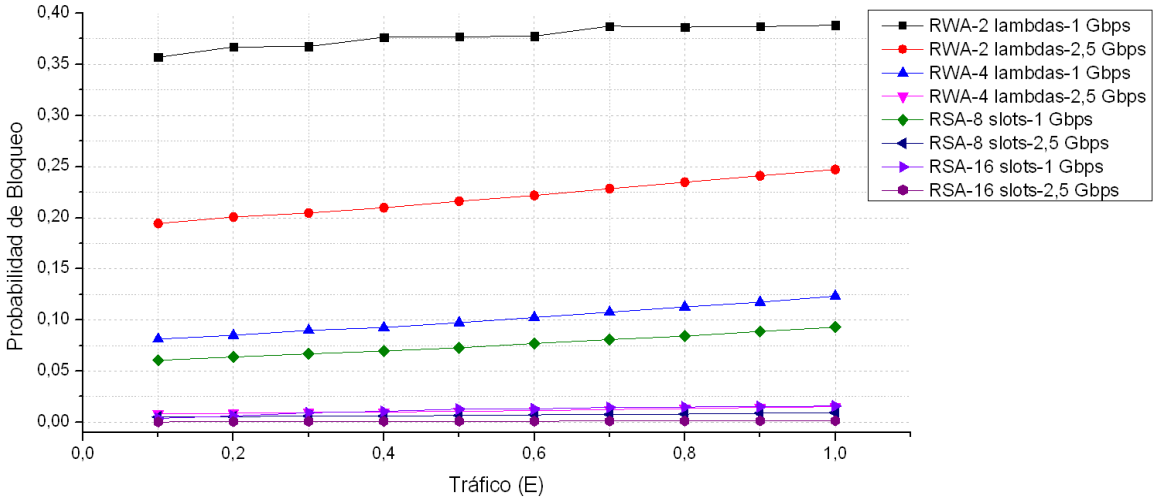


Figura B.1 Probabilidad de bloqueo vs tráfico, ts = 0.8 s [1].

C.2 Resultados obtenidos de retardo extremo a extremo en función del tráfico en la red.

Tabla B.2 Retardo extremo a extremo de ráfagas vs Tráfico

Tráfico (E)	RWA				RSA			
	2 λ (100 GHz)		4 λ (200 GHz)		8 slots (100 GHz)		16 slots (200 GHz)	
	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps
1	0,106210	0,026330	0,005290	0,004110	0,014130	0,004980	0,004110	0,003720
0,9	0,101820	0,025520	0,005240	0,004100	0,013620	0,004820	0,004100	0,003720
0,8	0,097830	0,025160	0,005230	0,004100	0,013090	0,004700	0,004100	0,003720
0,7	0,093980	0,024980	0,005140	0,004080	0,012700	0,004670	0,004100	0,003710
0,6	0,089810	0,024140	0,005000	0,004070	0,012420	0,004680	0,004080	0,003710
0,5	0,086800	0,023410	0,004860	0,004060	0,012010	0,004600	0,004070	0,003710
0,4	0,082330	0,022970	0,004800	0,004050	0,011350	0,004490	0,004070	0,003720
0,3	0,078270	0,022730	0,004750	0,004030	0,011270	0,004430	0,004060	0,003710
0,2	0,074440	0,022100	0,004640	0,004020	0,010960	0,004510	0,004060	0,003710
0,1	0,070960	0,021860	0,004600	0,004000	0,010330	0,004350	0,004050	0,003710

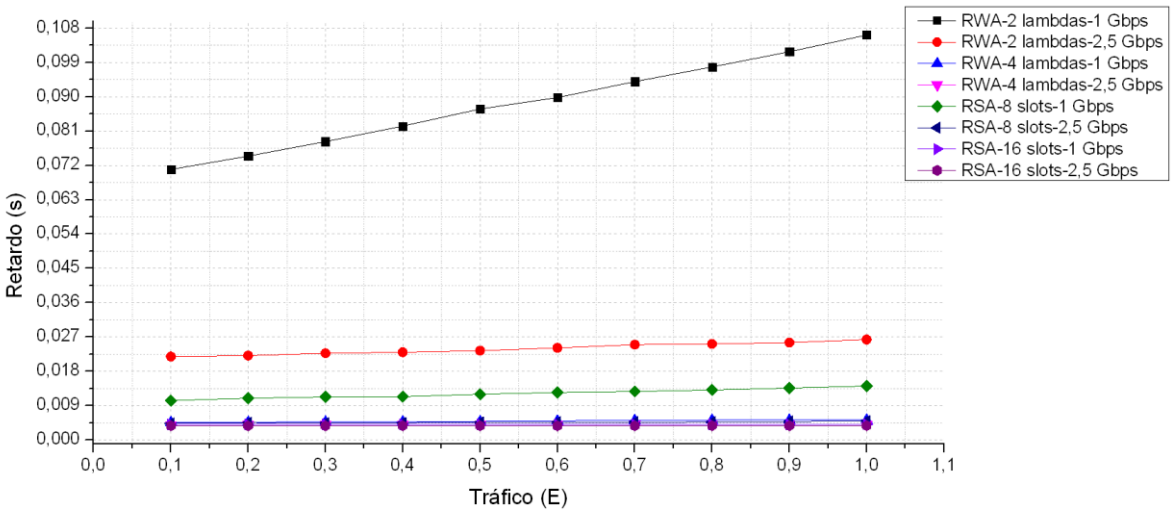


Figura B.2 Retardo extremo a extremo vs tráfico, ts = 0.8 s [1].

C.3 Número de ráfagas generadas, recibidas y perdidas en las redes para un tiempo de simulación de 0.8 s.

Tabla B.3 Ráfagas generadas, recibidas y perdidas en la red.

Red	Número de canales	Velocidad (Gbps)	Generadas			Recibidas			Perdidas		
			Alto	Medio	Bajo	Alto	Medio	Bajo	Alto	Medio	Bajo
RWA/WDM	2 lambdas	1	529576	228792	60826	323899	142524	39115	205677	86268	21711
		2,5	529576	228792	60826	398628	179318	49000	130948	49474	11826
	4 lambdas	1	529576	228792	60826	464210	206496	55879	65366	22296	4947
		2,5	529576	228792	60826	521532	226326	60316	8044	2466	510
RSA/FlexGrid	8 slots	1	529576	228792	60826	480315	212131	57136	49261	16661	3690
		2,5	529576	228792	60826	524572	227252	60510	5004	1540	316
	16 slots	1	529576	228792	60826	521140	225838	60589	8436	2954	237
		2,5	529576	228792	60826	528760	228531	60807	816	261	19

C.4 Resultados obtenidos de probabilidad de bloqueo en función del tiempo de simulación, para tráficos alto (1 E), medio (0.5 E) y bajo (0.1 E).

Tabla B.4 Probabilidad de bloqueo vs tiempo de simulación para trafico alto.

Tiempo (s)	RWA				RSA			
	2 λ (100 GHz)		4 λ (200 GHz)		8 slots (100 GHz)		16 slots (200 GHz)	
	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps
0,2	0,387770	0,245190	0,121850	0,015170	0,092090	0,009140	0,014730	0,001452
0,4	0,387850	0,246300	0,123130	0,014980	0,093060	0,009210	0,014820	0,001461
0,6	0,388170	0,246740	0,123330	0,015080	0,093010	0,009400	0,015880	0,001493
0,8	0,388380	0,247270	0,123430	0,015190	0,093020	0,009450	0,015930	0,001540

Tabla B.5 Probabilidad de bloqueo vs tiempo de simulación para trafico medio.

Tiempo (s)	RWA				RSA			
	2 λ (100 GHz)		4 λ (200 GHz)		8 slots (100 GHz)		16 slots (200 GHz)	
	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps
0,2	0,376270	0,216280	0,096670	0,010380	0,072220	0,006810	0,012800	0,001136
0,4	0,376780	0,215910	0,096710	0,010770	0,072790	0,006750	0,012870	0,001141
0,6	0,376670	0,216170	0,097620	0,010840	0,072530	0,006670	0,012870	0,001144
0,8	0,377060	0,216240	0,097450	0,010780	0,072820	0,006730	0,012910	0,001140

Tabla B.6 Probabilidad de bloqueo vs tiempo de simulación para trafico bajo.

Tiempo (s)	RWA				RSA			
	2 λ (100 GHz)		4 λ (200 GHz)		8 slots (100 GHz)		16 slots (200 GHz)	
	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps	1 Gbps	2.5 Gbps
0,2	0,357040	0,193020	0,080520	0,008090	0,061170	0,005310	0,003760	0,000000
0,4	0,356270	0,193910	0,080380	0,008340	0,060470	0,005200	0,003990	0,000010
0,6	0,356540	0,194120	0,080910	0,008260	0,060600	0,005180	0,003900	0,000007
0,8	0,356940	0,194420	0,081330	0,008380	0,060670	0,005190	0,003890	0,000305

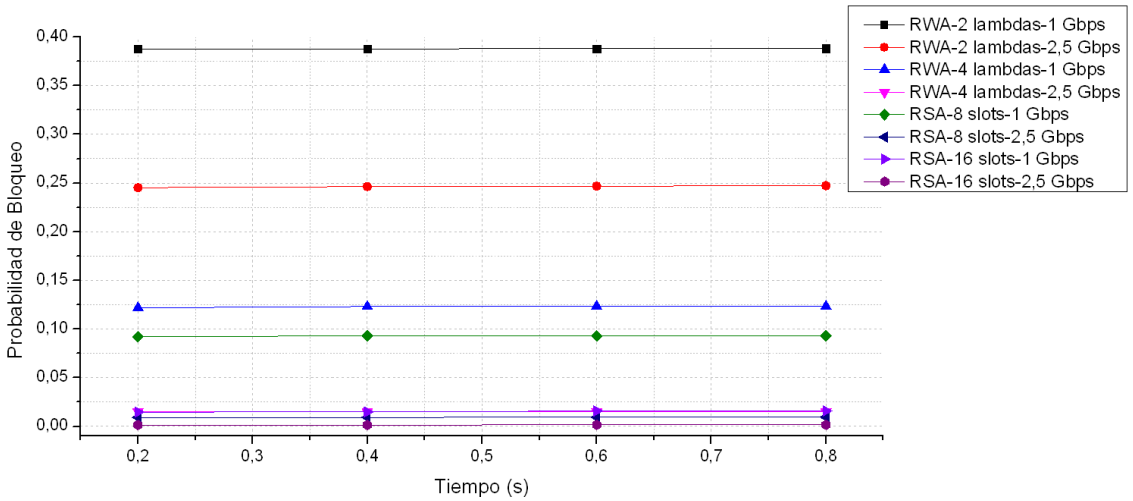


Figura B.3 Probabilidad de bloqueo vs tiempo de simulación para tráfico alto [1].

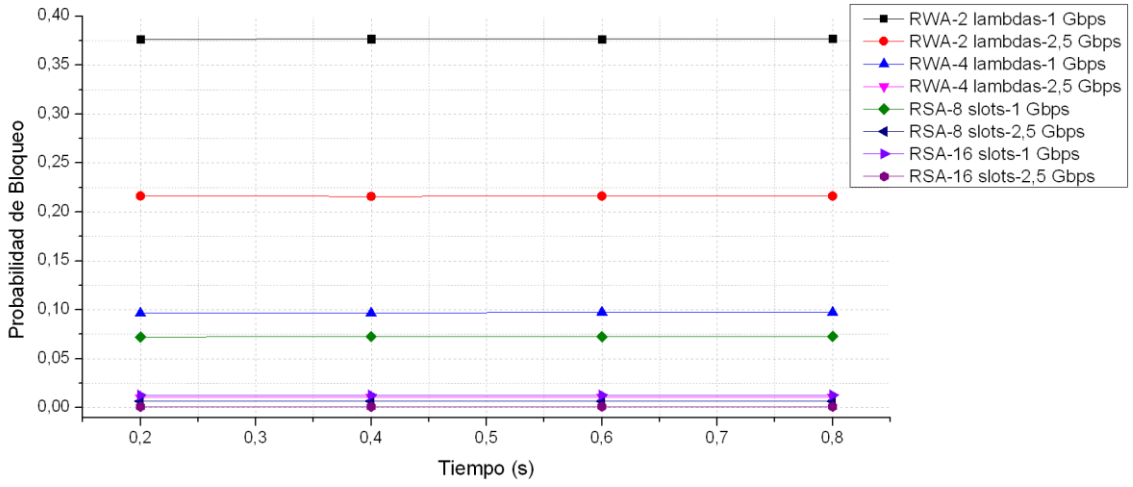


Figura B.4 Probabilidad de bloqueo vs tiempo de simulación para tráfico medio [1].

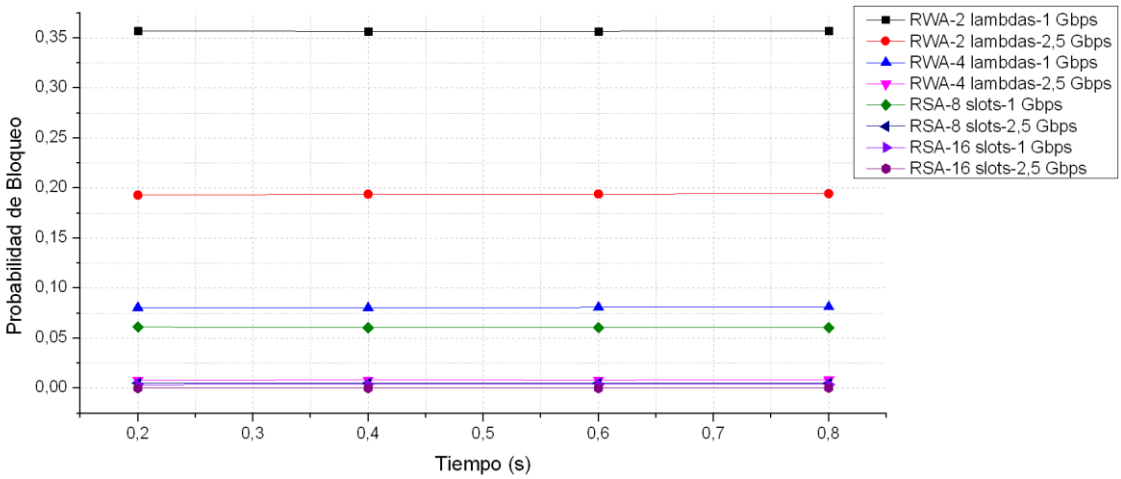


Figura B.5 Probabilidad de bloqueo vs tiempo de simulación para tráfico bajo [1].

REFERENCIAS

[1] A. Sevilla y E. Zúñiga, "*Análisis comparativo del desempeño de algoritmos RSA/RWA sobre una red óptica basada en la topología NSFNeT*", Tesis de pregrado, Universidad del Cauca, Popayán, Colombia, 2017.

[2] S. Shrinivas et al. "Applications of graph theory in computer science an overview", International Journal of Engineering Science and Tecnology, ISSN: 0975-5462, 2010.

[3] M. Claverol, E. Simo, "Matemática discreta, teoría de grafos", Barcelona, España: Departamento de matemáticas aplicada, EPSEVG-UPC, 2006.

[4] J. López, "Métodos matemáticos", I. 978-84-693-4783-6, Ed., Castellón: Universidad Jaume, 2010.