任意拓扑网络仿真器说明书

1. 简介

本网络仿真器是一款基于 popnet 的、可对任意拓扑的片上网络（芯粒集成网络）进行仿真的软件。用户只需准备轨迹文件和 DOT 格式的拓扑文件。在仿真时，程序会先读取拓扑文件，获得网络拓扑信息，使用 Floyd-Warshall 算法生成路由表，然后根据路由表，读取轨迹文件进行网络仿真。

2. 使用方法

2.1 编译与依赖

该软件依赖 Boost 库和 cmake。
假设源码目录为$ROOT。

将当前目录切换为$ROOT：

```
cd $ROOT
```

运行安装依赖库的命令：

```
sudo apt install libboost-all-dev
sudo apt install cmake
```

运行编译命令：

```
mkdir build
cd build
cmake ..
make
```

2.2 仿真

2.2.1 程序运行

运行命令：build/popnet -A vertices -c dimension -V vc_cnt -B input_buffer -O output_buffer -F flit_size -L 1000 -T time -r 1 -I trace_file -G topology_file -R routing_algorithm

参数说明：

-A vertices：网络节点数量为 vertices

-c dimension：网络维度为 dimension，在使用任意拓扑功能时恒为 1

-V vc_cnt：虚通道数量为 vc_cnt

-B input_buffer：路由器输入缓冲区的大小为 input_buffer

-O output_buffer：路由器输出缓冲区的大小为 output_buffer

-F flit_size：微片（flit）大小为 flit_size，以 64 比特为单位

-T time：仿真周期数为 time

-I trace_file：轨迹文件路径为 trace_file

-G topology_file：拓扑文件路径为 topology_file

-R routing_algorithm：路由算法为 routing_algorithm，在使用任意拓扑功能时恒为 4

命令示例：build/popnet -A 16 -c 1 -V 3 -B 12 -O 12 -F 4 -L 1000 -T 5.1e7 -r 1 -I test/bs-mesh-16.popnet_trace -G test/mesh_4_4.gv -R 4

2.2.2 轨迹文件

轨迹文件存储了需要注入网络的数据包，格式如下：

send_time source destination packet_size

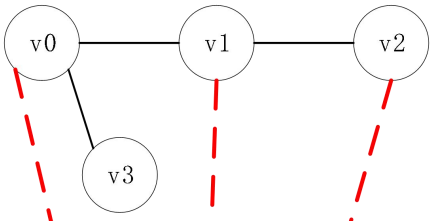
每一行代表一个数据包。从左到右的字段分别为发送时间、源地址、目的地址、数据包大小（以微片数量为单位）。

轨迹文件示例如下：

```
0.000000        0       284     1
3.000000        0       272     1
10.000000       284     34      1
```

2.2.3 拓扑文件

拓扑文件存储了本软件所仿真网络的拓扑。拓扑文件为 DOT 格式的图文件，如图 1。




```
//an undirected graph
graph g
{
    edge[weight=1]
    node[pipeline_stage_delay=1]
    v0--v1--v2
    v0--v3
}
```

图 1 拓扑文件示意图

在拓扑文件中，关键词 graph 表示图。edge[...]、node[...]可以定义边和节点内的属性。边属性 weight 是以周期为单位的链路延迟。节点属性 pipeline_stage_delay 为每级流

水线的时间（单位也是周期）。“--”表示边。在图 1 中，有节点 v0、v1、v2、v3，“v0--v1--v2”为 v0 和 v1 之间、v1 和 v2 之间分别定义了一条边，“v0--v3”为 v0 和 v3 之间定义了一条边。节点名称必须为位于区间[0,n)的数字，其中 n 为节点数量。

假设要构造一个 4×4 Mesh 网络拓扑，如图 2。

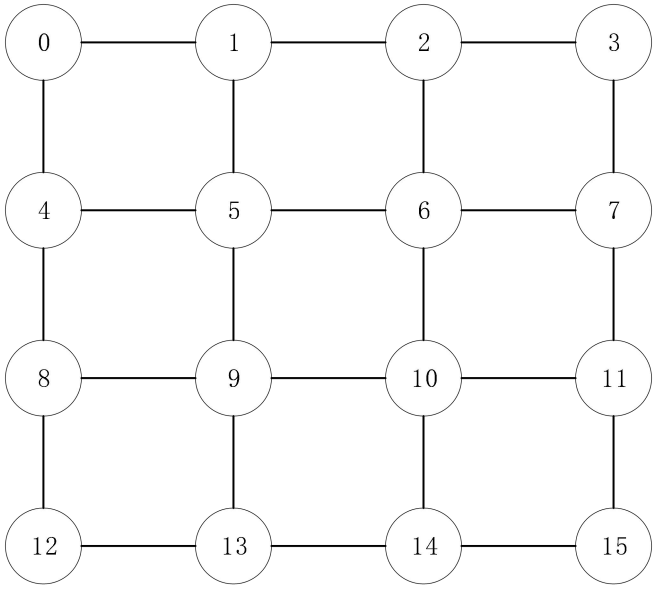


图 2 Mesh 网络拓扑示例

拓扑文件示例如下：

```
graph mesh_4_4
{
    edge[weight=1]
    node[pipeline_stage_delay=1]
    //横线
    0--1--2--3
    4--5--6--7
    8--9--10--11
    12--13--14--15
    //竖线
    0--4--8--12
    1--5--9--13
    2--6--10--14
    3--7--11--15
}
```

以上示例中，graph、edge、node 是关键词。首行给出了图的名称。“edge[weight=1]”定义了后面的边的属性。类似地，“node[pipeline_stage_delay=1]”定义了后面的节点的属性。而“0--1--2--3”则定义了节点 0、节点 1、节点 2、节点 3，以及连接节点 0 和节点 1、节点 1 和节点 2、节点 2 和节点 3 的三条边。后面的语句“0--4--8--12”也定义了连接节点 0 和节点 4、节点 4 和节点 8、节点 8 和节点 12 的三条边。但是，由于节点 0、节点 4、节点 8、节点 12 在前面的语句中已经定义，这一语句不会定义新的节点。

3. 仿真原理

3.1 事件驱动仿真

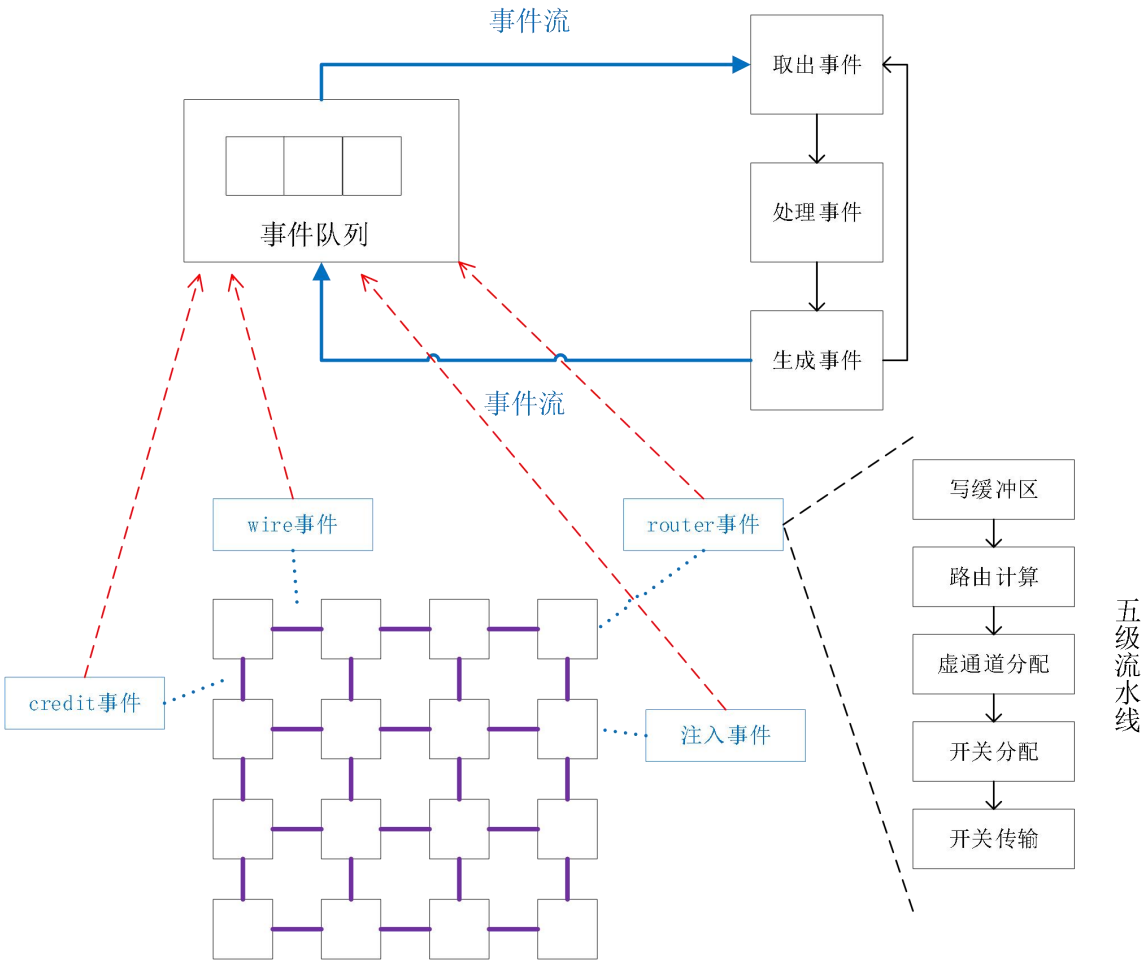


图 3 事件驱动仿真示意图

该软件是一个事件驱动的网络仿真器。程序有一个按时间排序的事件队列。程序不断地从队列中取出事件并将其处理，如图 3。事件分为注入、router、credit、wire 共 4 种，分别有以下含义和处理办法：

（1）注入事件：新数据包注入网络。

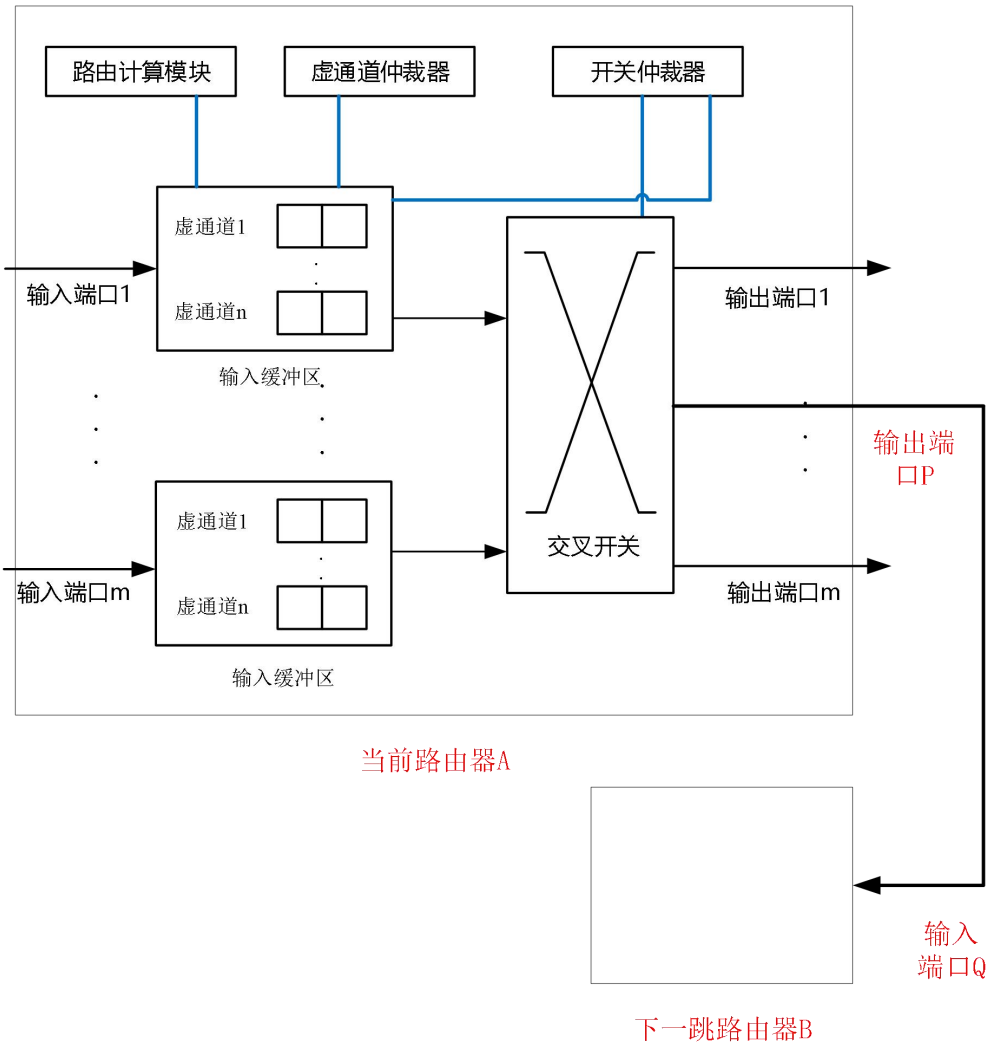


图 4 路由事件、传输事件、确认事件和路由算法的预设场景：数据包从路由器 A 传到路由器 B

（2）router 事件：执行路由算法。预设场景如图 4，程序执行以下步骤：

A. 执行 3.3 小节的路由算法，得出路由器 B 的地址和输出端口 P。

B. 通过虚通道仲裁器为数据包分配虚通道，避免冲突。

C. 通过开关仲裁器为数据包在交叉开关中分配通路。

D. 将数据包写入端口 P 的输出缓冲区。

（3）credit 事件：通知上一跳路由器可以接收数据包。预设场景如图 4，程序执行以下步骤：

A. 将数据包打包成传输事件，插入到事件队列中。

B. 将数据包从端口 P 的输出缓冲区移除。

（4）wire 事件：模拟数据包在链路（link）中传输的过程。预设场景如图 4，程序将数据包写入端口 Q 的输入缓冲区。

程序处理事件的流程图如图 5。

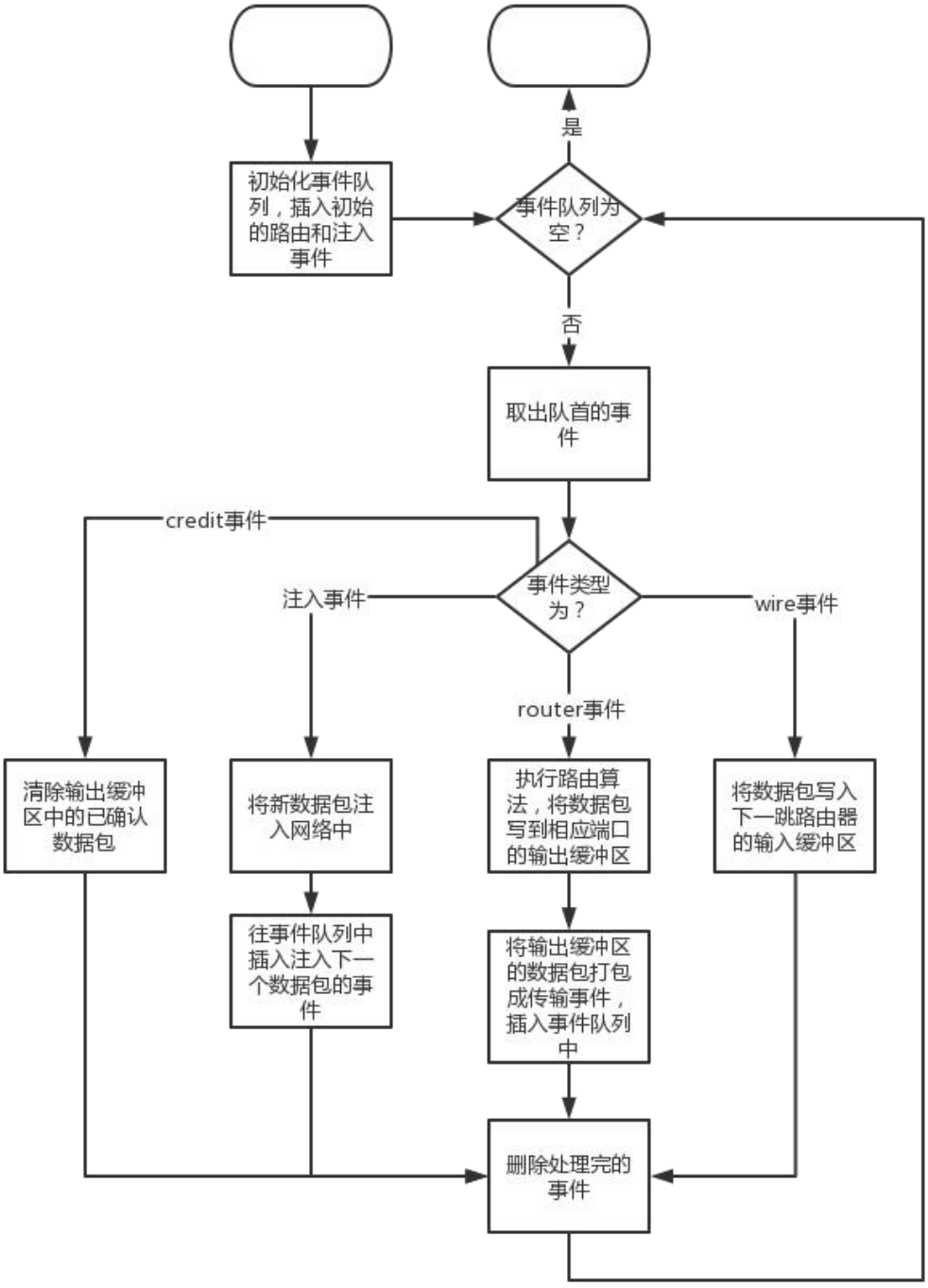


图 5 事件处理流程图

3.2 端口映射表和路由表

为了进行任意拓扑的网络仿真，在初始化时，程序需要根据拓扑文件建立端口映射表，然后用 Floyd-Warshall 算法计算路由表。

建立端口映射表的步骤如下：

（1）读取拓扑文件，获得网络拓扑。网络拓扑中有链路（边）的列表 edges，其长度为

e。

（2）遍历所有边。索引变量为 i。对每条链路执行以下步骤：

A. 假设当前链路连接了路由器（节点）s 和 d。程序让路由器 s 新增一个端口 p，让路由器 d 新增一个端口 q。

B. 程序在路由器 s 的端口映射表中增加条目(p, d, q)，在路由器 d 的端口映射表中增加条目(q, s, p)。

建立端口映射表的程序框图为图 6。最终，每个路由器都有类似表 1 的端口映射表。在实际运行中，程序会生成以邻接路由为键、条目位置为值的散列表，将该散列表作为索引，加快查询速度。

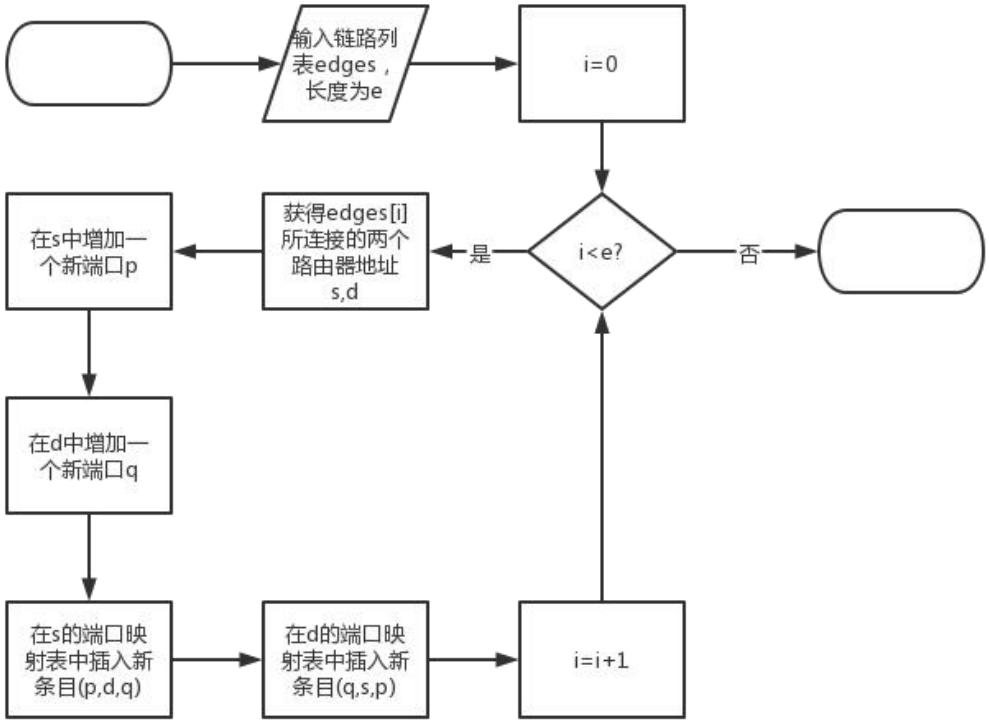


图 6 端口映射表的生成流程图

表 1 路由器的端口映射表示例

端口号	邻接路由	邻接端口号
p_1	v_1	q_1
p_2	v_2	q_1
P	B	Q
……	……	……

假设网络拓扑的邻接矩阵为$weight$，路由器（节点）的数量为v，路由器的转发延迟数组为$router_delay$，程序使用 Floyd-Warshall 算法建立全局路由表的步骤如下：

（1） 将距离矩阵$dist$所有元素初始化为$+\infty$。将路由表$next_hop$的所有元素初始化为-1。

（2） 遍历矩阵$dist$的元素。设索引变量分别为i, j，不断执行步骤 3-4。

（3） 令$dist(i,j) = router_delay(i) + weight(i,j) + router_delay(j)$。

（4） 若$dist(i,j) \neq + \infty$，则令$next_hop(i,j) = j$。

（5） 对于路由器$k=0$到$v-1$，执行步骤6-8。

（6） 对于路由器$i=0$到$v-1$，执行步骤7-8。

（7） 对于路由器$j=0$到$v-1$，执行步骤8。

（8） 依次检查经过路由器k的i到j的路径是否比现有从i到j的路径更短：

A. 令$t=dist(i,k)+dist(k,j)$。

B. 若$i\neq j$且 $j\neq k$且$i\neq k$，则$t=t-router_delay(k)$。

C. t为经过路由器k的i到j的路径的距离。若$t<dist(i,j)$，则令$dist(i,j)=t$，$next\_hop(i,j)=next\_hop(i,k)$。

（9） 输出距离矩阵$dist$和路由表$next_hop$。

路由表生成算法如表2，最终生成的路由表如表3。

表 2 路由表生成算法

输入：邻接矩阵$weight$、路由器数量v、路由器转发延迟列表$router_delay$

输出：距离矩阵$dist$、全局路由表$next_hop$

1. 将距离矩阵$dist$所有元素初始化为$+\infty$
2. For $i=0$ to $v-1$ do
3. For $j=0$ to $v-1$ do
4. $dist(i,j)=router\_delay(i)+weight(i,j)+router\_delay(j)$
5. If $dist(i,j)=+\infty$ then
6. $next\_hop(i,j)=j$
7. End if
8. End for
9. End for
10. For $k=0$ to $v-1$ do
11. For $i=0$ to $v-1$ do
12. For $j=0$ to $v-1$ do
13. $t=dist(i,k)+dist(k,j)$
14. If $i\neq j$ and $j\neq k$ and $i\neq k$ then
15. $t=t-router\_delay(k)$
16. End if
17. If $t<dist(i,j)$ then
18. $dist(i,j)=t$
19. $next\_hop(i,j)=next\_hop(i,k)$
20. End if
21. End for
22. End for
23. End for
24. Return $dist,next\_hop$

表 3 全局路由表的示例

当前地址	目的地址	下一跳地址
v_0	v_1	v_3
v_0	v_2	v_4
A	D	B

……	……	……

3.3 路由算法

程序中只有一个路由表，由所有路由器共享。在路由过程中，程序先后查找路由表和端口映射表，以将微片传输给下一跳路由器。假设待转发的数据包 F 的源地址为 S，目的地址为 D，数据包 F 从路由器 A 传到下一跳路由器 B，预设场景如图 4，路由器转发数据包的具体步骤如下：

（1）程序从事件队列中取出一个路由事件，依次对所有路由器执行路由算法。

（2）路由器 A 从输入缓冲区取出数据包，根据它的地址 A 和数据包 F 的目的地址 D 从全局路由表（如表 1）中找出路由器 B 的地址。

（3）路由器 A 根据路由器 B 的地址，从它的端口映射表（如表 3）找出输出端口号 P 和邻接端口号 Q。路由器 A 将数据包 F 写入端口 P 的输出缓冲区。

（4）路由器 B 的端口 Q 的输入缓冲区有空间。路由器 B 产生 credit 事件。

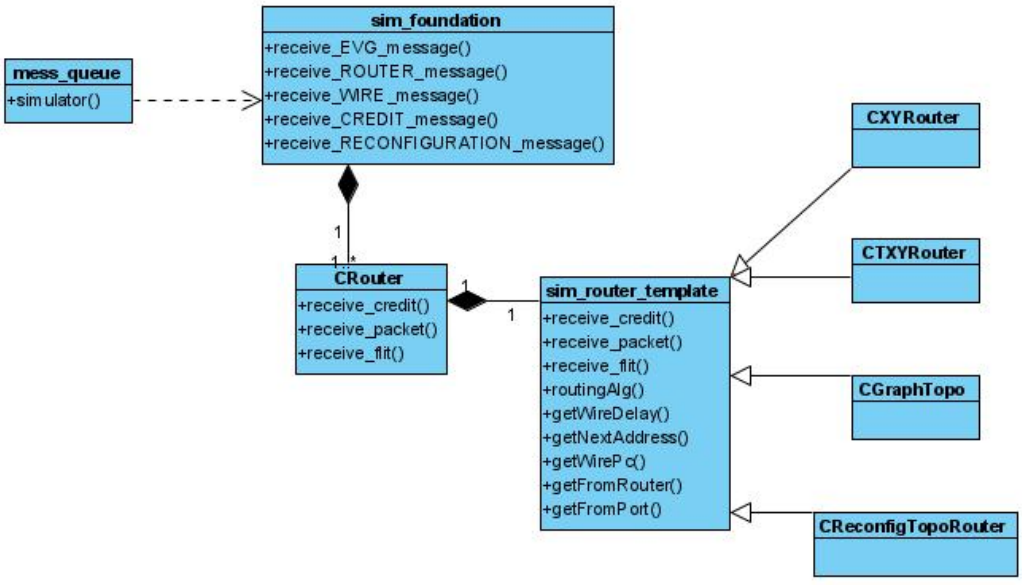
（5）路由器 A 收到路由器 B 的 credit 事件，得知路由器 B 的端口 Q 的输入缓冲区有空间。

（6）路由器 A 往事件队列中插入一个 wire 事件。该事件有路由器 B 的地址和输入端口号 Q。

(7) 程序处理上一步产生的 wire 事件。数据包 F 被写入路由器 B 的端口 Q 的输入缓冲区。

附录：对 popnet 的重要修改

修改后，本软件的类图如下：



图表 1 本软件的类图

mess_queue 是一个单例类。这个类维护一个存有事件的优先队列，在这个队列中，事件按时间顺序排列。这个类不断从队列中取出最早的事件，调用 sim_foundation 相应的函数处理。

sim_foundation 也是一个单例类。这个类保存了多个 CRouter 对象，模拟整个网络。sim_router_template 模拟网络中的路由器,CRouter 则是 sim_router_template 的包装类。为了支持多种拓扑和路由算法，sim_router_template 的路由相关函数 routingAlg、getWireDelay、getNextAddress、getWirePc、getFromRouter、getFromPort 都被定义为虚函数。在 sim_router_template 的子类 CXYRouter、CTXYRouter、CGraphTopo、CReconfigTopoRouter 中，这些函数被重写以支持不同的网络拓扑和路由算法。其中，类 CGraphTopo 实现了第 3 节基于端口映射表和路由表的路由算法。

这些可重写的路由相关函数如下：

```
class sim_router_template
{
protected:
    virtual void routingAlg(const add_type&dst,const add_type&src,long
s_ph,long s_vc);
    virtual time_type getWireDelay(long port);
    virtual void getNextAddress(add_type&nextAddress,long port);
    virtual long getWirePc(long port);
    virtual void getFromRouter(add_type&from,long port);
    virtual long getFromPort(long port);
public:
    void receive_credit(long port, long vc);
    void receive_packet();
    void receive_flit(long port, long vc, flit_template & flit);
};
```

routingAlg 是路由函数，会计算数据包的输出端口和虚通道。此函数实现了 3.3 小节中路由器转发数据包步骤的第 2 步和第 3 步。

getWireDelay 返回与端口 port 相连的链路的延迟。此函数从网络拓扑中获得链路的延迟。

getNextAddress 计算端口 port 所连接的路由器地址，并写入 nextAddress。此函数从本路由器的端口映射表中找出与本地端口 port 相连的邻接路由器地址。

getWirePc 计算与端口 port 相连的对方路由器的端口。此函数从本路由器的端口映射表中找出与本地端口 port 相连的邻接端口。

getFromRouter 计算端口 port 的数据包的来源路由器的地址，并写入 from。此函数从本路由器的端口映射表中找出与本地端口 port 相连的邻接路由器地址。

getFromPort 计算端口 port 的数据包的来源路由器的端口。此函数从本路由器的端口映射表中找出与本地端口 port 相连的邻接端口。

receive_credit 实现了 3.3 小节中路由器转发数据包步骤的第 5 步。

receive_packet 实现了路由器注入新数据包的过程。

receive_flit 实现了 3.3 小节中路由器转发数据包步骤的第 7 步。

在 CGraphTopo 类中，以上的虚函数根据路由表和端口映射表返回相应的值。路由表和端口映射表是程序初始化时根据网络拓扑生成的。

关键代码：

```
//s_ph 为输入端口
```

```
void CGraphTopo:: routingAlg(const add_type&dst,const add_type&src,long
s_ph,long s_vc,
    TIntMatrix&routing_table,unique_ptr<std::unordered_map<TAddressNumbe
r,TAddressNumber>[]>&next_hop_port_map)
{
    //从全局路由表中找出下一跳地址
    TAddressNumber
nextAdd=routing_table[getAddressNumber()][dst.front()];
    //从端口映射表的索引中找出输出端口
    auto it=next_hop_port_map[getAddressNumber()].find(nextAdd);
    if(it==next_hop_port_map[getAddressNumber()].end()){
        cerr<<"Routing error: dst "<<dst.front()<<" in
"<<getAddressNumber()<<" next: "<<nextAdd<<endl;
        Sassert(false);
    }
    TAddressNumber port=it->second;
    //尝试输出端口的全部虚通道
    addRoutingForDifferentVC(input_module_,s_ph,s_vc,port,configuration::
ap().virtual_channel_number());
}

time_type CGraphTopo:: getWireDelay(long port)
{
    //从拓扑中找出延迟
    return portMap[getAddressNumber()][port].linkDelay;
}

void CGraphTopo:: getNextAddress(add_type&nextAddress,long port)
{
    //初始化变量
    neighbourAddress.clear();
    //从端口映射表中找出下一跳地址
    neighbourAddress.push_back(portMap[getAddressNumber()][port].neighb
our);
}

long CGraphTopo:: getWirePc(long port)
{
    //从端口映射表中找出输出端口
    return portMap[getAddressNumber()][port].neighbourPort;
}

void CGraphTopo::getFromRouter(add_type&from,long port)
{
```

```
    //初始化变量
    neighbourAddress.clear();
    //从端口映射表中找出上一跳地址
    neighbourAddress.push_back(portMap[getAddressNumber()][port].neighb
our);

}
long CGraphTopo::getFromPort(long port)
{
    //从端口映射表中找出输入端口
    return portMap[getAddressNumber()][port].neighbourPort;
}
```