UniMon：基于FPGA+CPU的统一流状态管理模型

# 研究背景

公有云需要部署有状态网络功能，例如有状态NAT、有状态防火墙，满足租户的需求。为每种有状态网络功能单独开发流状态维护功能，存在功能冗余。针对该问题，mOS[1]基于软件（用户态）实现了统一流管理功能，不同的网络功能开发者只需要编写各自的报文处理代码，而共用同一套流状态管理逻辑。但mOS处理性能受限于CPU核数量，处理延时较大且不稳定，（而且还需占用服务器的大量CPU资源）。另外，微软在其数据中心发现某一些应用的单条连接会突发性地产生较大流量[2]，而多核并行处理无法保证单条流的处理性能（如果将同一条流的报文分散在不同核上处理，需要解决流状态一致性问题）。

FPGA具有良好的并行性和可重性，被数据中心用于分组处理的加速。例如Microsoft Azure将分组处理卸载到基于FPGA实现的smartNIC上[2]，以获得线速处理（40Gbps），并极大地降低CPU资源占用率和分组处理延时。在该模型中，smartNIC充当cache的作用，即缓存一张大流表（unified flow table），而该流表的处理规则由报文经过VFP[3]多级流表所生成（类似OVS中的Microflow table）。当新到达的报文未能命中smartNIC中的统一流表或流状态发生转化时，需要将其送给软件实现的分组处理平台（VFP），处理性能严重下降。另外，该模型并未提供统一的流状态管理，不同的分组处理模块（每一层VFP Layer）需要维护各自的流状态。

为此，我们提出UniMon（unified connection monitor），一种基于FPGA+CPU架构实现的统一流状态管理。思想相对简单，即CPU负责实现复杂流状态处理和用户编写的处理逻辑，而将基本的TCP流连接管理卸载到FPGA上，以获得加速。

## 需求与解决方案

UniMon的设计需要满足以下四个需求（Requirements）：

**R1**：维护TCP流的状态信息，以提供统一流状态管理。目前，数据中心网络99.91%的流量为TCP，UniMon需要维护多种类型的TCP流状态信息以供有状态网络功能使用。例如，有状态NAT需要知道流两端的主机地址信息；TCP伪装重传则需要知道TCP是否发生重传，以及重传报文和先前报文的内容。

**R2**：减少CPU占用率，减少网络功能处理延时。在数据中心，CPU资源相对宝贵，降低网络功能的CPU占用率，从而部署更多的虚拟机以获得更高的经济效益。另外，为了满足低延时应用的需求，需要降低网络功能的处理延时。

**R3**：具有线速处理能力（40+Gbps）。目前数据中心的链路带宽已由原来的1Gbps增长至40Gbps，并且100Gbps的规范也在制定中。因此，需要UniMon支持线速处理，包括单流的线速处理。

**R4**：为网络功能开发者提供开发接口，简化网络功能开发（代码行数）。

针对上述需求，我们的方案基于FPGA+CPU实现统一流状态管理，并提供良定义的开发接口，对用户掩藏TCP流状态管理的实现细节。我们为UniMon设计了以下3个部件（Components）以满足上述需求。

**C1**：FPGA-base basic connection manager：UniMon将TCP的基本状态卸载到FPGA上，以降低CPU资源占用率，并提高处理性能（**R1 and** **R2**）。另外，在FPGA上统一维护流状态，能够保证流状态一致性，允许将同一条流的报文分散在不同核上，提高单流的吞吐率（**R3**）。

**C2**：CPUb-based completed connection manager：UniMon在CPU上实现完整的TCP流的状态维护（**R1**），解决FPGA维护全流信息的难题。同时支持用户配置流过滤规则，以选择性建立流状态表项，以及所维护的状态类型等，解决存储开销和状态管理开销（**R2**）。

**C3**：Event manager（UniMon networking API）：UniMon采用event-handler开发模式，用户通过注册自定义实现，和事件处理函数开发所需的网络功能（**R4**）。UniMon networking API包含两部分，即FPGA-part API和CPU-part API，FPGA-part API可用于支持网络功能到FPGA的卸载，但支持的基本事件类型相对较少，而CPU-part API支持更加丰富的基本事件类，以及流状态的读取功能。

# UniMon模型设计

## UniMon三层模型



图1 UniMon模型

UniMon模型如图1所示，可以将其分为三层。每一层对应1.1节所述的一个部件，从下往上分别为TCP连接的基本状态管理，TCP连接的完整状态管理和事件管理部件。其中，TCP连接的基本状态管理部件放在FPGA上实现，负责维护TCP连接的基本状态信息（即TCP连接两端主机的状态，如建立连接、半关闭、关闭状态，详细见3.1节硬件设计），以加速TCP连接状态管理。同时，我们在TCP连接的完整状态管理部件（基于单个UA实现）中维护更多的TCP连接状态信息（已发送、确认的报文序列号），以支持更加丰富的有状态网络功能开发。另外，两个TCP连接管理部件除了根据接收报文更新TCP连接状态外，还会根据状态类型、及状态的转化生成多种基本事件，并通告事件管理部件。事件管理部件采用event-handler编程模型，允许用户注册事件处理函数，在接收到TCP连接管理模块通告的基本事件后，查找并执行这些基本事件对应的处理函数，以实现用户所定义的分组处理功能。

## UniMon部件间的接口定义

* + 1. 两个TCP连接管理部件之间的接口定义

在UniMon中，FPGA维护的TCP连接与软件所维护的相同，只是各自维护的连接状态不同，即FPGA维护基本的连接状态；软件负责维护复杂的连接状态（具体维护的状态类型见各自模块设计中的关键数据结构部分）。另外，为保证软硬件中的连接数量一致，连接的创建、删除仅由UniMon的软件负责，而FPGA负责判断、上报超时连接。由于连接的基本状态在FPGA中维护，并且只有部分报文会通过Packet\_in的方式上报给软件（为了降低CPU的处理负载，FPGA会过滤上报的报文类型），导致FPGA与软件所维护的连接状态不同步的情况。为此，我们为FPGA中的连接表增加了上报选项字段，允许FPGA定时（每隔一段时长）或者定期（每接收一定数量报文）上报当前流状态，以更新软件所维护的连接表。

为支持上述软硬件协同处理功能，我们设计了以下17种类型的交互消息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 消息  类型 | 触发条件 | 交互内容 | 所携带的参数 |
| 1 | 上报 | 硬件连接表未命中 | 上报未命中的报文 | flowID = 0 |
| 2 | Every packet（软件配置的上报条件） | 上报新接收的报文 | flowID = *x*，cur\_state = *y*，  previous\_state = *z*， |
| 3 | Connection setup | 上报连接建立报文 | flowID = *x*，cur\_state = *y*， |
| 4 | 表项超时 | 上报超时的连接 | flowID = *x*，cur\_state = *y*，  last packet timestamp = *z* |
| 5 | State change（软件配置的上报条件） | 当状态发生变化时，上报改变状态的报文 | flowID = *x*，cur\_state = *y*，  pre\_state = *z* |
| 6 | Connection end（包括FIN，RST报文） | 上报结束报文 | flowID = *x*，cur\_state = *end* |
| 7 | http packet | 上报http报文 | flowID = *x*，cur\_state = *y* |
| 8 | Every *x* ms（软件配置的上报条件） | 上报连接状态 | flowID = *x*，cur\_state = *y*， |
| 9 | Every *x* pkt（软件配置的上报条件） | 上报连接状态，以及当前接收的报文 | flowID = *x*，cur\_state = *y*， |
| 10 | 配置硬件中的  连接表 | 增加连接 | 为新连接增加新连接表项 | flow\_key = *w*，flowID = *x*，cur\_state = *y*，pkt\_in\_cnd = *z*，  aging\_interval = *v* |
| 11 | 删除连接 | 连接结束（包括超时，接收FIN，RST报文） | flowID = *x*， |
| 12 | 修改连接 | 修改连接状态、上报条件 | flowID = *x*，cur\_state = *y*，pkt\_in\_cnd = *z* |
| 13 | 读取连接 | 读取连接状态 | flowID = *x* |
| 14 | 读响应 | 返回连接状态 | flow\_key = *w*，flowID = *x*，cur\_state = *y*，pkt\_in\_cnd = *z*  aging\_interval = *v*，  pkt\_count = *u*，byte\_count = *r* |
| 15 | 配置hash Table | 修改hash table | Hash table采用2-left hash | left\_right\_tag = *p*，addr = *q*， hash\_entry\_ctx = *s* |
| 16 | 读取hash table | left\_right\_tag = *p*，addr = *q* |
| 17 | 读响应 | hash\_entry\_ctx = *s* |

此外，每一种上报的参数还额外携带pkt\_in\_cdt(pkt\_in\_condition/reason) = 消息类型序号；



图2 FPGA与软件交互消息

* + 1. TCP连接管理（软件部分）与事件管理之间的接口定义

TCP连接管理模块，会根据接收的报文以及该报文所属流的状态，产生8种基本事件（见2.2.3小节），并通告事件管理部件，同时携带流和报文信息。此外，事件管理部件需要根据用户对流过滤规则（BPF格式）以及用户所关心的基本事件类型（bitmap mask）配置TCP连接管理部件中的过滤表。为了保证软件和硬件中流的一致性，事件管理模块需要调用TCP连接管理部件提供的接口函数实现连接状态的配置功能。

为此，我们设计了以下4种类型的交互消息：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 消息  类型 | 触发条件 | 交互内容 | 所携带的参数 |
| 1 | 通告基本事件 | 连接管理部件（软件）根据报文产生基本事件 | 基本事件类型（bitmap表示） | connection指针（其中包含了packet指针，见connection数据结构） |
| 2 | 修改连接状态 | 读取连接 | 读取的连接状态 | flowID = *x* or flow\_key = *y* |
| 3 | 修改连接 | 修改的连接状态 | flowID = *x*，conn\_entry = *z* |
| 4 | 修改过滤规则 | 增加filter entry | 每个sock绑定一个filter entry（filter entry包含了该sock关心的流类型，使用BPF表示，还包含该sock所关心的基本事件类型，使用bitmap表示） | sock = *p*，filter\_entry\_ctx = *q* |



图3连接管理部件（软件）与事件管理之间的消息交互

TCP连接管理部件（软件）接口函数如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 交互对象 | 交互内容 | 函数名称 | 含义 |
| 与FPGA交互 | 报文 | void fast\_ua\_recv() | Fast接口获取报文，所需的信息，包括flow\_key, flowID, cur\_state, pre\_state, pkt\_in\_cnd, aging\_info, counter\_info分别在metadata.user[0]的[127:32], [31:16], [15:0], metadata.user[1]的[127:112], [111:80], [79:64], [63:0].（详细见3.1.2小节） |
| int fast\_ua\_send(struct fast\_packet \*pkt, int pkt\_len) | Fast接口发送报文 |
| 状态 | int fpga\_add\_con\_entry(fpga\_state\_t fstate, uint16\_t flowID) | 增加连接表项 |
| int fpga\_read\_con\_entry(uint16\_t flowID, flow\_key\_t fkey, int type, fpga\_state\_t fpga\_conn) | Type=0，根据flowID读取connection，type=1，根据5-tuple读取connection |
| int fpga\_modify\_con\_entry(fpga\_state\_t fstate, uint16\_t flowID) | 修改连接表项 |
| int fpga\_del\_con\_entry(fpga\_state\_t fstate, uint16\_t flowID) | 删除连接表项 |
| Hash table | int fpga\_modify\_hashTable(int tableID, int entryID, hash\_entry\_t hety) | 修改hash表内容 |
| int fpga\_read\_hashTable(int tableID, int entryID, hash\_entry\_t hety) | 读取hash表内容 |
| 与事件管理部件交互 | 读取connection table | state\_t fast\_read\_connection\_entry(uint16\_t flowID, flow\_key\_t fkey, int type) | Type=0，根据flowID读取state，type=1，根据5-tuple读取state |
| 修改connection表项 | int fast\_modify\_connection\_entry( uint16\_t flowID, state\_t conn\_ety) | 回写状态，也需要调用该函数，以保证FPGA，UA间的一致性 |
| 配置Filter table | int set\_flow\_filter (int sock, filter\_t ft) | 根据过滤条件（ft）配置对应sock的filter entry，成功返回0 |
| 触发基本事件 | int fast\_raise\_event(event\_bitmap\_t eb, state\_t state) | 调用事件处理函数 |
| main调用 | 线程处理 | pthread\_t fast\_unimon\_run() | 实现抓包、连接维护、事件处理功能 |
| wait\_fast\_unimon\_end() | 等待线程结束 |

超时时长采用脚本配置的方式实现。

* + 1. 事件管理部件为上层网络功能开发提供的编程接口

UniMon借鉴mOS编程思想，采用socket id来表示当前操作的流类型（即用socket id来识别当前操作的流），因此，为用户提供的开发接口需要类似socket管理函数。另外，事件管理部件需要为上层网络功能提供自定义事件（过滤函数）、自定义处理函数注册接口。为此，我们设计了如下7个接口函数：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 类型 | 函数名称 | 含义 |
| 1 | Socket相关 | int fast\_socket (int domain, int type) | 创建sock, domain为IPv4 或者 IPv6, type为MONITOR或者PROCESS |
| 2 | int fast\_bind\_monitor\_filter(int sock, monitor\_filter\_t ft) | 为sock绑定流过滤规则 |
| 3 | int fast\_close(int sock) | 关闭sock |
| 4 | 事件注册 | event\_t fast\_define\_event(event\_t ev, filter\_t filt, struct filter\_arg \*arg) | 设置自定义事件 |
| 5 | int fast\_register\_callback(int sock, event\_t ev, int side, callback\_t cb) | 注册事件处理函数 |
| 6 | 报文处理 | int fast\_sendpkt(int sock, struct pkt\_info \*pinfo) | 发送报文 |
| 7 | 获得流状态 | int fast\_getflowstate (int cur\_tag, uint16\_t flowID, connecton\_t \* con\_info) | cur\_tag = CURRENT表示读取当前流状态信息，否则根据flowID读取流状态信息； |



图4 用户配置事件管理部件的消息交互

网络功能编程示例（利用上述接口函数编写有状态防火墙）

|  |
| --- |
| int main(){  # initial fast state manager and event manager;  fast\_unimon\_init();  # sock\_init; choose process mode;  int lsock = fast\_socket( AF\_INET, FAST\_SOCK\_PROCESS\_STREAM );  # binding filter to socket;  fast\_bind\_monitor\_socket( lsock, “dst host 10.0.0.2” );  # resigter handler function  fast\_register\_callback(lsock, FAST\_ON\_CONN\_START, CLI\_SIDE,  check\_request\_side); // only allowing inner -> outer  # register user-defined event  event\_t eventOutOfRecvWind = fast\_define\_event(FAST\_ON\_PKT\_IN,  check\_pkt\_seq, NULL)  fast\_register\_callback(lsock, eventOutOfRecvWind, BOTH\_SIDE,  drop\_pkt); // drop out-of recv window packets  # recv and process packet  pthread\_t fum = fast\_unimon\_run();  # close socket;  wait\_fast\_unimon\_end(&fum)  fast\_close(lsock);  return 0;  } |

## UniMon支持的基本事件

UniMon同样采用event-handler的编程模式，并提供多种类型的基本事件。用户可以基于基本事件编写过滤函数来实现自定义事件。基本事件类型如下：

|  |  |  |
| --- | --- | --- |
| 事件类型 | | 含义 |
| 1 | **FAST \_ON\_PKT\_IN** | 接收到报文 |
| 2 | **FAST \_ON\_CONN\_START** | 连接建立请求（syn） |
| 3 | **FAST \_ON\_CONN\_SETUP** | 完成三次握手 |
| 4 | **FAST \_ON\_TIMEOUT** | 流超时（长时间未到达新报文） |
| 5 | **FAST \_ON\_TCP\_STATE\_CHANGE** | 流状态发生转化 |
| 6 | FAST \_ON\_REXMIT | 发现重传报文 |
| 7 | **FAST \_ON\_CONN\_END** | 完成4次结束握手，或流超时，或接收RST报文 |
| 8 | **FAST\_ON\_HTTP\_PKT** | 接收到http报文 |

**加粗的事件**表示，硬件也支持的基本事件类型。但值得注意的是，硬件中的连接管理部件同样会产生7种基本事件，但仅供硬件中的模块使用，不会上报给软件的连接管理部件。

# 各部件详细设计

## FPGA中的TCP连接管理部件

本节主要介绍在FPGA中实现的TCP连接管理部件的整体框架（3.1.1节），负责接收数据报文，并根据接收的报文维护基本的TCP连接状态（3.1.2小节），产生7类基本事件，供后续UDA使用。然后，介绍子模块间的连接关系和接口设计（3.1.3小节）。最后，介绍TCP连接表所采用的2-left hash和超时判断逻辑（3.1.4小节）。

### TCP连接管理部件（FPGA部分）整体设计

TCP连接管理部件（FPGA）的整体设计如图5所示，主要包含5部分：

|  |  |
| --- | --- |
| 模块名称 | 功能 |
| Connection table（双端口RAM） | 用于保存流连接信息 |
| Connection searcher | 实现连接表的查找、更新功能（包含hash查找功能），产生packet\_in消息（报文触发） |
| Build-in event generator | 产生基本事件类型（采用bitmap表示），需要汇聚报文触发的基本事件以及超时模块触发的超时事件 |
| Connection table configuration | 实现连接表的配置、读取、删除操作，供软件使用 |
| Connection out-time inspector | 判断连接是否超时，产生packet\_in消息（超时触发） |



图5 TCP连接管理模块设计（FPGA）

* + 1. 主要数据结构

首先，是连接表格式，具体如图6所示，主要包含8个成员（共），即flowID, flow\_key, state, action, Next\_index, Aging\_tag, packet\_in\_cnd, aging\_info, counter\_info。其中flowID等同于RAM的addr，可以省略存储；flow\_key采用四元组表示，并按照从大到小的顺序排列；state包含该连接的方向（client or server），连接状态，两端的状态；action则是对该流的处理（例如pkt\_in first or forwarding or drop）；而Next\_index用于指向下一个hash链索引（构造冲突链），aging\_tag内部表示已超时，不可配置（仅供内部使用）；Pkt\_in\_cnd包含7类上报条件；Aging info用于判断连接是否超时；counter info则是统计该流所接收的报文数和字节数。FPGA目前支持的最大规则数量为512条，对应的虚拟地址空间为（0x3000\_0000-0x3000\_01ff），其中[12:4]表示规则addr，[3:0]用于拼接完整表项。



图6 连接表数据结构

其次，上报消息格式，具体如图7所示，包含metadata和packet两部分，其中metadata[1:2]携带连接信息。



图7 上报消息结构

* + 1. 模块连接关系和接口信号设计

我们在3.1.1节中介绍了5个子模块功能，其连接关系和接口信号定义如图8所示。



图8 内部子模块的连接关系和接口信号定义

* + 1. 子模块详细设计

5个模块中，Connection table采用双端口RAM实现；Connection table configuration模块则实现Connection table（RAM）的读写功能；基本事件生成模块，用于汇聚报文触发的基本事件和超时触发的超时事件，起到了多路选择器的功能，设计相对简单。这里着重介绍以下两个模块的设计。

1. **Hash查找算法设计（查找模块）**

采用2-left hash算法，首先将关键字输入两个hash函数获得两个hash值。然后分别查找两个hash表，并比较hash表中的简化的4元组信息（16b），能够获得目标表项所在的索引值（16b）。最后，搜索目索引值所指的hash链，获得匹配的连接表项。

1. **超时模块设计**

双端口RAM中的一个端口用于connection table配置和超时维护，超时维护主要在配置功能空闲时执行，超时的轮询的周期为100ms。当发现表项超时，则标记aging\_tag位，后续匹配的报文都不在修改该连接状态，而是上报给软件。

## CPU中的TCP流管理模块

本节主要介绍在CPU（UA）中实现的TCP连接管理部件的整体框架（3.2.1节），负责接收FPGA上报的消息（包含流状态和接收的报文），并根据该信息更新TCP连接表状态（3.2.2小节），产生9类基本事件，并通告事件管理部件。然后，介绍子模块的详细介绍（3.2.3小节）。最后，编写了CUP实现TCP流管理模块的伪代码（3.2.4小节）。

### TCP连接管理部件（CPU部分）整体设计

TCP流管理模块的设计如图9所示，主要包含4部分：

|  |  |
| --- | --- |
| 模块名称 | 功能 |
| Filter (Table) | 实现流过滤，（用户通过fast\_bind\_monitor\_filter函数绑定过滤条件），以及该流所关心的基本事件类型（使用bitmap表示），每个sock对应一个filter表项 |
| Connection searcher | 实现流状态表查询和更新操作（另外发现FPGA上报流超时，则在Connnection table中删除该流表项） |
| Event generator | 生成基本事件并送给事件管理模块 |
| Configuration | 用于实现配置功能（包括Filter table，Flow state table，和硬件中Connection table） |



图9 TCP流管理模块设计

### 主要数据结构

Flow state table数据结构如下：

|  |
| --- |
| struct tcp\_stream{  SOCKQ\_HEAD(socket\_t) socks; /\* socks which interest this flow \*/  uint16\_t flowID;  uint32\_t hash\_idx;  uint32\_t stream\_type; /\* Monitor or Process mode \*/  uint32\_t saddr; /\* in network order \*/  uint32\_t daddr; /\* small one \*/  uint16\_t sport;  uint16\_t dport;  uint8\_t direction; /\* in or out \*/  struct tcp\_endPoint\_vars \*rcvvar;  struct tcp\_endPoint\_vars \*sndvar;    uint8\_t state; /\* request, established, closed \*/  uint8\_t close\_reason; /\* close reason \*/  uint16\_t filter\_event\_bitmap\_mask; /\* a flow can be interested by many sockets \*/  uint8\_t pkt\_in\_cdt; /\* pkt\_in condition for FPGA\*/    pkt\_info\_t last\_pkt;  int dropPkt\_tag;  uint32\_t last\_active\_ts;  uint32\_t pkt\_count;  uint32\_t byte\_count;  };  typedef struct tcp\_stream \*state\_tj;  struct tcp\_endPoint\_vars  {  SOCKQ\_HEAD(tcp\_seq) MESseq; /\* message seq\*/  SOCKQ\_HEAD(tcp\_seq) ACKseq; /\* ack seq\*/  uint16\_t window;  uint32\_t seq\_range\_from;  uint32\_t seq\_range\_to;  uint32\_t state;  }; |

filter table数据结构：

|  |
| --- |
| struct filter\_entry{  int sock; /\* socket id \*/  monitor\_filter\_t filter\_bpf;  uint16\_t filter\_event\_bitmap\_mask;  }; |

### 子模块详细设计

1. **Filter (Table)**：

Filter的功能包括两方面，一方面是实现流过滤，而过滤条件是fast\_bind\_monitor\_filter函数所绑定的过滤条件；另一方面，针对流类型得到关心该流的基本事件类型（使用bitmap表示）。

Filter table中的每一条表项对应一个socket，其表项数据结构包含两个成员，即基于BPF描述的过滤条件和bitmap表示的感兴趣基本事件类型。值得注意的是，filter功能只需要对FPGA未命中的流表进行过滤（即flowID = 0），而其他命中的报文一定是感兴趣的报文，因为这些报文所属的流的首报文已经过滤。

Filter采用串型搜索方式实现，最大的filter entry数量为32，即最大同时支持32个网络功能。

1. **Connection searcher**：

采用与硬件相同的hash查找算法，针对新流创建新连接表项，而超时表项则需删除。同时，根据需求维护序列号状态。

1. **Event generator**：

在硬件7中基本事件类型的基础上，判断是否发生重传事件，构造8种基本事件bitmap，并调用事件管理模块提供的事件触发函数。

1. **Configuration**：

读取软件中的连接表（调用connection searcher提供的功能函数），以及配置硬件的连接表。

每个模块所包含的基本功能函数如下：

|  |  |  |
| --- | --- | --- |
| 头文件 | 函数名 | 功能 |
| configuration.h | int fast\_modify\_connection\_entry( uint16\_t flowID, state\_t conn\_ety) | 回写状态，也需要调用该函数，以保证FPGA，UA间的一致性 |
| int fpga\_add\_conn\_entry(fpga\_state\_t fstate, uint16\_t flowID) | 增加连接表项 |
| fpga\_state\_t fast\_read\_conn\_entry(uint16\_t flowID, flow\_key\_t fkey, int type) | Type=0，根据flowID读取connection，type=1，根据5-tuple读取connection |
| int fast\_modify\_conn\_entry(fpga\_state\_t fstate, uint16\_t flowID) | 修改连接表项 |
| int fast\_del\_conn\_entry(fpga\_state\_t fstate, uint16\_t flowID) | 删除连接表项 |
| int fast\_modify\_hashTable(int tableID, int entryID, hash\_entry\_t hety) | 修改hash表内容 |
| recvSendPkt.h | int recv\_packet\_from\_fast\_os (metadata\_t meta, pkt\_info\_t pinfo) | 获取报文 |
| int recv\_packet\_from\_fast\_os (metadata\_t meta, pkt\_info\_t pinfo) | 发送报文 |
| common.h | flow\_key\_t get\_5\_tuple(pkt\_info\_t *pinfo*); | 从报文/Meta中获取5-tuple信息 |
| uint16\_t get\_flowID(metadata\_t *meta*); | 从metadata中获取flowID |
| genEvent.h | event\_bitmap\_t update\_state(state\_t fstate, pkt\_info\_t pinfo); | 更新流状态，并生成基本事件（bitmap） |
| managerConnection.h | connection\_t fast\_read\_connection\_table(uint16\_t flowID, flow\_key\_t fkey, int type) | Type=0，根据flowID读取state，type=1，根据5-tuple读取state |
| int fast\_modify\_conn\_table(state\_t fstate, uint16\_t flowID) | 修改状态表项 |
| int fast\_del\_ua\_conn\_entry(state\_t fstate, uint16\_t flowID) | 删除状态表项 |
| int fast\_add\_ua\_conn\_entry(state\_t fstate, uint16\_t flowID) | 删除状态表项 |
| filter.h | int lookup\_filter\_table(  filter\_table\_t *ftable*, flow\_key\_t *fkey*, sock\_series\_t sockList, uint16\_t event\_bitmap\_mask); | 过滤报文，获得对该流感兴趣的socks，以及bitmap\_mask |

### TCP流管理模块整体实现伪代码

|  |
| --- |
| /\*receive packet and metadata \*/  {metadata\_t *metadata*, pkt\_info\_t *pinfo*} =  recv\_packet\_from\_fast\_os();  /\* get 5\_tuple info and flowID of packet \*/  flow\_key\_t *fkey* = get\_5\_tuple(pkt\_info\_t *pinfo*);  uint16\_t *flowID* = get\_flowID(metadata\_t *meta*);  /\* match with filter\_condition of every sock \*/  lookup\_filter\_table( filter\_table\_t *ftable*, flow\_key\_t *fkey*  sock\_series\_t *socks*\_*heade,*uint16\_t event\_bm);  /\* lookup and update flow state table \*/  if(*flowID* != 0)  state\_t *fstate* = fast\_read\_flowState\_entry(  uint16\_t *flowID*, flow\_key\_t *fkey*, int *0*));  else  state\_t *fstate* = fast\_read\_flowState\_entry(  uint16\_t *flowID*, flow\_key\_t *fkey*, int *1*));  *fstate*->pinfo = *pinfo*;  event\_bitmap\_t *evb* = update\_state(state\_t *fstate*,  pkt\_info\_t *pinfo*);  /\* raise event \*/  fast\_raise\_event(event\_bitmap\_t *evb*, sock\_series\_t  *socks*\_*header*, state\_t *fstate*); |

## CPU中的事件管理模块

本节主要介绍事件管理部件的整体框架（3.3.1节），负责接收连接管理部件通告的基本事件类型（bitmap表示），并根据该信息查询event-handler树表（3.3.2小节），生成自定义事件，调用callback函数。然后，介绍子模块的详细介绍（3.3.3小节），并举例说明了自定义事件生成树的构建和工作过程（3.3.4）。最后，编写了实现事件管理部件的伪代码（3.3.5小节）。

### 事件管理部件整体设计

事件管理部件的设计如图10所示，主要包含4部分：

|  |  |
| --- | --- |
| 模块名称 | 功能 |
| UDE generator | 根据基本事件类型生成用户自定义事件 |
| Event Handler | 根据事件类型执行相应动作（查询event-handler table） |
| Register | 实现用户自定义事件、处理函数等注册功能，并配置Flow State Manager中的filter table |



图5 事件管理模块设计

### 主要数据结构

Event table数据结构如下，并采用树形结构维护

|  |
| --- |
| typedef struct filter\_arg {  void \*arg;  size\_t len;  } filter\_arg\_t;  typedef struct \_tree\_node\_t {  filter\_t ft; // 过滤函数  callback\_t cb; // 回调函数  event\_t ev; // 事件id  struct filter\_arg arg; // 过滤函数变量  uint32\_t is\_in\_raiseq:1;  TREE\_NODE(\_tree\_node\_t) link; // link in the tree  TREE\_NODE(\_tree\_node\_t) invk; // inverse link: used for invoking callbacks  } tree\_node\_t;  #define \_TREE\_NODE(type, qual) \  struct { \  qual type \*tn\_parent; /\* parent \*/ \  qual type \*tn\_first; /\* first child \*/ \  qual type \*tn\_last; /\* last child \*/ \  qual type \*tn\_younger; /\* younger sibling \*/ \  qual type \*tn\_older; /\* older sibling \*/ \  }  #define TREE\_NODE(type) \_TREE\_NODE(struct type,)  typedef struct \_stree\_t {  int ref\_cnt;  uint64\_t id;  tree\_node\_t \*root;  } stree\_t; |

### 子模块详细设计

**1）Register:**

Register模块提供自定义事件接口函数和事件处理接口函数，以便用户自定义事件和事件回调函数，同时，它提供过滤功能函数接口，用于进行流状态管理模块的规则配置。对内提供操作事件树的功能函数，如树节点的添加、查询、修改和删除，用于根据用户的定义来生成事件树，以便UDE generator模块和Event handler模块根据事件树来工作。

1. **UDE generator:**

当接收到一个基本事件后，查询该基本事件所对应的事件树，按照树结构，根据过滤函数产生新事件，并根据事件处理函数处理该新事件。在自定义事件产生的过程中，若某个节点的事件根据过滤函数并不能产生，则不再遍历该节点的子节点。

1. **Event handler:**

当一个事件发生后，查找该事件对应的事件处理回调函数，然后调用该回调函数执行。

模块主要包含以下处理函数：

|  |  |  |
| --- | --- | --- |
| 头文件 | 函数名称 | 功能 |
| event\_filter.h | int set\_event\_filter (int sock, event\_bitmap\_t evbit\_mask) | 绑定事件过滤bitmap |
| event\_bitmap\_t get\_event\_filter (int sock) | 获取事件过滤bitmap |
| event.h | event\_t gen\_user\_defined\_event(int sock, event\_t ev, filter\_t filt, struct filter\_arg \*arg) | 生成自定义事件 |
| filter\_t find\_filter(int sock, event\_t ev) | 查询过滤函数 |
| callback\_t find\_handler(int sock, event\_t ev) | 获取处理函数 |
| int set\_user\_defined\_event(int sock, event\_t ev, filter\_t ft, event\_t next\_ev) | 设置自定义函数 |
| tree.h | tree\_node\_t \* tree\_node\_new() | 生成树节点 |
| void tree\_node\_del(tree\_node\_t \*node) | 删除一个树节点 |
| void tree\_del\_recursive(tree\_node\_t \*node) | 删除树节点以及它的子节点 |
| void tree\_node\_add(tree\_node\_t \*node) | 添加一个树节点 |
| tree\_node\_t \* tree\_search(tree\_node\_t \*root, event\_t ev) | 根据事件ID查询树节点 |
| sock.h | int fast\_socket(int domain, int type) | 创建一个sock |
| int mtcp\_close(int sock); | 关闭一个sock |

### 事件管理工作实例

1. 用户自定义事件（user\_defined\_event）模块

假设基本事件为e1和e2,用户调用自定义事件生成函数，分别为：

fast\_define\_event(e1, f1, arg1);

fast\_define\_event(e1, f2, arg2);

fast\_define\_event(e2, f3, arg3);

fast\_define\_event(e3, f4, arg4);

执行后，事件e1根据过滤函数f1生成事件自定义事件e3，同样的e1通过f2生成事件e4，e2通过f3生成e5, e3通过f4生成e7。生成的这些自定义事件会加入到事件生成树当中。



图6 事件生成树结构（一）

（2）事件处理（event\_handle）模块

调用注册回调函数对所关心的事件实现处理函数，这些处理函数主要用来实现用户的应用逻辑。

如用户关心（1）中的e3、e4、e5、e6和e7，那么就要分别定义这四个事件的处理函数，具体实现如下：

fast\_register\_callback(e3, cb3);

fast\_register\_callback(e4, cb4);

fast\_register\_callback(e5, cb5);

fast\_register\_callback(e6, cb6);

fast\_register\_callback(e7, cb7);

执行后，会将回调函数写入事件生成树中，树的结果如下：



图7 事件生成树结构（二）

（3）产生自定义模块（UDE generator）

当解析一个报文时产生的基本事件e1和e2。对于事件e1来说，它首先判断它是否有处理回调函数，发现没有，后分析子节点产生自定义事件。在例子中，e1事件根据过滤函数f1判断是否产生e3，这里假设产生了e3，查看e3拥有处理回调函数，则对事件进行处理。依次按照分析树产生事件，处理事件。

当一个报文的事件产生、处理完毕后，从FAST接口获取下一个报文事件信息进行处理。

### 事件管理模块整体实现伪代码

|  |
| --- |
| /\* 创建事件生成树（由用户在main函数中调用）\*/  **if** (fast\_define\_event or fast\_register\_callback)  create\_update\_event\_tree();  /\* 处理已触发的基本事件（由状态管理模块调用）\*/  fast\_raise\_event(event\_bitmap\_t *evb*, sock\_series\_t *socks*\_*header*, state\_t *fstate*);{  /\* 获得底层传上来的基本事件 \*/  event\_bitmap\_t *evb\_filted* = lookup\_event\_filter\_table(  event\_filter\_table\_t *ftable*, event\_bitmap\_t *evb*);  /\* 根据事件生成树生成事件、处理事件；\*/  **for** *each\_bit* **in** *evb\_filter*:  **if** (*each\_bit*)  *curNode* = event\_root\_node[*each\_bit*];  **for** *each\_son\_node* **in** *curNode*:  **if** (*cunNode*->tn\_first)  *curNode* = *each\_son\_node-*>tn\_first;  **else**  process\_event(*each\_son\_node->cb*);  *curNode* = *curNode*->parent;  } |

# 开发案例介绍

## 有状态防火墙

* + 1. 有状态防火墙功能需求

有状态的防火墙主要实现以下四个功能：

（1）防止tcp半连接攻击。

（2）内网可以访问外网，外网不能访问内网。

（3）只允许http 报文通过（用于web服务器的防护）。

（4）窗口外的报文丢弃。

* + 1. 功能实现

针对功能需求，分四个方面实现功能：

（1）分析基本事件FAST \_ON\_TIMEOUT，设置过滤函数，检查该流的连接状态，若连接状态为半连接，则产生TCP半连接事件TCP\_HALF\_CONNECTION。在该事件的处理函数中对该事件的发生次数进行计数，在单位时间内发生的次数超过了阈值，则将该地址设置于黑名单，阻止其通过。

（2）分析基本事件FAST \_ON\_CONN\_START，设置过滤函数，检查TCP请求是否是外网到内网地址的请求，若是，则产生OUT\_TO\_IN事。在该事件的处理函数中设置标记，阻止报文通过。

（3）在检测到基本事件FAST\_ON\_HTTP后，在基本事件FAST\_ON\_HTTP事件的处理函数中设置该报文允许通过。

（4）分析基本事件FAST \_ON\_PKT\_IN，设置过滤函数，检查窗口大小和序列号，判断报文是否为窗口外的报文，若是，则产生OUT\_WIN\_PKT事件。当产生OUT\_WIN\_PKT事件后，丢弃该报文。

* + 1. 事件类型

实现有状态的防火墙需要产生以下7种事件：

|  |  |  |
| --- | --- | --- |
| 事件类型 | | 含义 |
|  | FAST \_ON\_PKT\_IN | 接收到报文 |
| FAST \_ON\_CONN\_START | 连接建立请求（syn） |
| FAST \_ON\_TIMEOUT | 流超时（长时间未到达新报文） |
| FAST\_ON\_HTTP | http报文 |
| 自定义事件 | TCP\_HALF\_CONNECTION | 接收到syn报文并回复syn-ack，但未收到ack报文，同时连接超时 |
| OUT\_TO\_IN | 外网向内网发送连接请求 |
| OUT\_WIN\_PKT | 报文序列号超过接收窗口 |

## TCP流量计费（伪装重传）

# 附录

关于HTTP报文的特征匹配

|  |
| --- |
| pattern pattern = Pattern.compile("[a-zA-Z]{3,7} .\* HTTP/1.[0,1]");  //匹配所有的HTTP 请求报文  matcher matcher = pattern.matcher(string);  pattern pattern2 = Pattern.compile("^HTTP/1.[0,1] [0-9]{0,3} \*");  //匹配所有的HTTP 响应报文  matcher matcher = pattern.matcher(string); |

上图是HTTP\_REQUEST报文的基本情况，报文头部为方法（get/post/head/put/delete/connect/options/trace等7种）+URI+http/版本号，即"[a-zA-Z]{3,7} .\* HTTP/1.[0,1]"；另一种是HTTP\_RESPOND报文，其头部为状态行，即http/版本号+状态码（三位数字）+描述（比如continue/ok等），可以描述为"^HTTP/1.[0,1] [0-9]{0,3} \*"