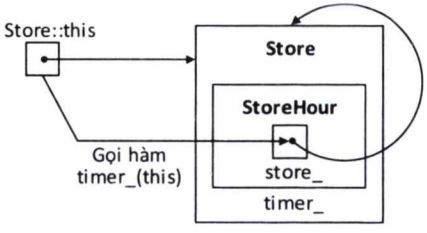
Chương trình 12.7. Khai báo của lớp `Store` và `StoreHour`.

```
      //store.h
1   class Store;
2   class StoreHour : public TimerHandler {
3   public:
4       StoreHour(Store *s) { store_ = s; };
5       virtual void expire( Event *e );
6   protected:
7       Store *store_;
8   };

9   class Store : public TclObject {
10  public:
11      Store() : timer_(this) { hours_ = -1; count_ = 0; };
12      void close(){
13          printf("So luong khach hang trong
                    %2.2f gio mo cua la %d\n", hours_,count_);
14      ;
15      int command(int argc, const char*const* argv);
16  protected:
17      double hours_;
18      int count_;
19      StoreHour timer_;
20  }
```

Tham chiếu chéo giữa đối tượng Store và StoreHour



Hình 12.4. *Sơ đồ biểu diễn tiến trình tham chiếu chéo giữa đối tượng* `Store` *và đối tượng* `StoreHour`

Tiến trình tham chiếu chéo giữa đối tượng `Store` và `StoreHour` được minh họa trong Hình 12.4. Hàm khởi tạo của lớp `Store` khởi tạo biến `timer_` của nó bằng con trỏ `this` của đối tượng `Store` (xem dòng 11). Hàm khởi tạo của lớp `StoreHour` lưu con trỏ đầu vào này vào trong biến `store_`. Vì đối số

đầu vào là một con trỏ trỏ tới đối tượng Store, về mặt bản chất hàm khởi tạo của đối tượng StoreHour thiết lập biến store_ quay lại trỏ vào đối tượng Store.

Để tham chiếu chéo, trình biên dịch cần nhận ra một trong hai lớp trên khi khai báo lớp kia. Nếu bỏ đi dòng 1, trình biên dịch sẽ không nhận ra lớp Store khi biên dịch đến dòng 7 và sẽ báo lỗi lên trên màn hình. Sau khi biên dịch dòng 2, trình biên dịch nhận ra lớp StoreHour và có thể biên dịch dòng 19 mà không gặp lỗi.

Điều quan trọng cần chú ý khi biên dịch các dòng 2-8 là trình biên dịch chỉ nhận ra tên của lớp Store. Nếu ta cố gắng gọi hàm close() của lớp Store thì sẽ gây ra lỗi biên dịch. Đây là lý do tại sao ta lại cần phân tách file mã nguồn và file header trong C++. Cần nhắc lại là vì file header được đưa vào trên đỉnh của file mã nguồn nên trình biên dịch sẽ biên dịch qua file header để nhận ra toàn bộ các biến và các hàm đã định nghĩa trong file header. Vì vậy sau đó trình biên dịch có thể biên dịch file mã nguồn C++ mà không gặp lỗi.

Xác định các hành động khi bộ đếm vượt ngưỡng

Chương trình 12.8: Hàm expire của lớp StoreHour và các lệnh open và new-customer của lớp Store.

```
   //store.cc
1  void StoreHour::expire(Event*) {
2     store_->close();
3  };

4  int Store::command(int argc, const char*const* argv)
5  {
6     if (argc == 3) {
7        if (strcmp(argv[1], "open") == 0) {
8           hours_ = atoi(argv[2]);
9           count_ = 0;
10          timer_.sched(hours_);
11          return (TCL_OK);
12       }
13    } else if (argc == 2) {
14       if (strcmp(argv[1], "new-customer") == 0) {
15          count_++;
16          return (TCL_OK);
17       }
18    }
19    return TclObject::command(argc,argv);
20 }
```

Lớp StoreHour thừa kế từ lớp TimerHandler. Nó xây dựng lại hàm expire(e) (các dòng 1-3 trong Chương trình 12.8). Tại thời điểm bộ đếm vượt ngưỡng, đối tượng StoreHour sẽ gọi hàm close() của đối tượng Store.

Tạo giao tiếp OTcl

Ta thực hiện việc liên kết lớp C++ Store với lớp OTcl cùng tên bằng cách sử dụng lớp ánh xạ StoreClass trong Chương trình 12.9. Các dòng 4-20 trong Chương trình 12.8 cho ta thấy các lệnh giao tiếp OTcl open{hours} và new-customer{} với hours là đối số đầu vào biểu diễn số giờ mở cửa. Lệnh OTcl open{hour} (dòng 8-11) sẽ được gọi khi cửa hàng mở cửa. Dòng 8 lưu trữ thời gian mở cửa trong biến hours_. Dòng 9 thiết lập lại số lượng khách tới cửa hàng giá trị bằng 0. Dòng 10 thiết lập đối tượng timer_ sẽ vượt ngưỡng sau "hours_" giờ. Lệnh OTcl new-customer{} sẽ được gọi khi một khách hàng đi vào cửa hàng. Trong dòng 15, lệnh này đơn giản chỉ thực hiện việc tăng giá trị biến count_ lên 1 đơn vị. Cần nhắc lại là, tại thời điểm bộ đếm thời gian vượt ngưỡng, bộ đếm này sẽ gọi hàm close() qua con trỏ store_ và in ra số giờ mở cửa (hours_) cũng như số lượng khách hàng đã đến cửa hàng (count_) trong ngày (xem hàm expire(e) trong dòng 2 của Chương trình 12.8).

Chương trình 12.9: Lớp ánh xạ StoreClass thực hiện liên kết giữa lớp Store trong C++ và lớp Store trong OTcl.

```
      //store.cc
1  static class StoreClass : public TclClass {
2     public:
3        StoreClass() : TclClass("Store") {}
4        TclObject* create(int, const char*const*) {
5           return (new Store);
6        }
7  } class_store;
```

Chạy thử chương trình

Sau khi xây dựng file store.cc và store.h, ta đưa file store.o vào Make File và chạy lệnh make tại thư mục gốc của NS2 để thêm các lớp Store và StoreHour vào trong NS2 (xem Mục 2.7).

Tiếp theo ta xây dựng file mô phỏng Tcl có tên là store.tcl để chạy thử chương trình

```
//store.tcl
1 set ns [new Simulator]
2 set my_store [new Store]
3 $my_store open 10.0
4 $ns at 1 "$my_store new-customer"
5 $ns at 5 "$my_store new-customer"
6 $ns at 6 "$my_store new-customer"
7 $ns at 8 "$my_store new-customer"
8 $ns at 11 "$my_store new-customer"
9 $ns run
```

Cuối cùng ta chạy file script store.tcl và thu được kết quả như sau:

```
>>ns store.tcl
So luong khach hang trong 10.0 gio mo cua la 4
```

Từ đoạn mã tcl trên, khi dòng 2 tạo một đối tượng `Store`, NS2 sẽ tự động tạo đối tượng C++ ánh xạ có tên là `Store`. Dòng 3 gọi lệnh `open` với giá trị tham số đầu vào là 10.0 với ý nghĩa mở cửa hàng trong vòng 10 giờ. Từ Chương trình 12.8, lệnh OTcl `open{10.0}` báo cho bộ đếm thời gian liên kết với nó sẽ vượt ngưỡng sau 10 giờ và đặt lại giá trị của biến `count_` trở về 0. Các dòng 4-8 gọi lệnh `new-customer{}` tại các giờ thứ 1, 5, 6, 8 và 11. Mỗi một dòng trên sẽ tăng số lượng khách hàng (biến `count_`) lên 1 đơn vị. Tại thời điểm giờ thứ 11, biến `count_` đúng ra sẽ có giá trị bằng 5. Tuy nhiên chương trình lại hiển thị số lượng khách hàng tới cửa hàng là bằng 4 bởi vì bộ đếm thời gian đã vượt ngưỡng và gọi hàm `close()` tại thời điểm giờ thứ 10.

12.1.4. Hướng dẫn triển khai bộ đếm Timer trong NS2

Bây giờ ta sẽ tổng kết lại tiến trình định nghĩa một bộ đếm mới. Giả sử rằng ta muốn định nghĩa một lớp bộ đếm `Timer` mới có tên là `StoreHour` và đối tượng `Store` là đối tượng có nhiệm vụ khởi động, khởi động lại và hủy bỏ đối tượng `StoreHour` cũng như thực hiện các hành động khi bộ đếm vượt ngưỡng. Khi đó tiến trình xây dựng các lớp ở trên bao gồm các bước sau:

Đối với lớp StoreHour

- Bước 1: Thiết kế cấu trúc lớp
  - Thừa kế lớp `StoreHour` từ lớp `TimerHandler`.
  - Khai báo một con trỏ (`store_`) trỏ tới lớp `Store`. Các hàm public của lớp `Store` có thể được truy cập thông qua con trỏ này.
- Bước 2: Tham chiếu tới lớp `Store` trong hàm khởi tạo
- Bước 3: Xác định các hành động xảy ra khi bộ đếm vượt ngưỡng trong hàm `expire(e)`

```
  ### KHỞI ĐỘNG LẠI NS2 ###
9 >> ns
10    >>$defaultRNG seed
11    1
12    >>$defaultRNG next-random
13    729236
14    >>$defaultRNG next-random
15    1193744747
16    >>exit

  ### KHỞI ĐỘNG LẠI NS2 ###
17    >>ns
18    >>$defaultRNG seed 101
19    >>$defaultRNG next-random
20    72520690
21    >>$defaultRNG next-random
22    308637100
23    >>exit
```

Trong lần chạy thứ nhất (dòng 1-8), biến `defaultRNG` được sử dụng để sinh ra hai số ngẫu nhiên. Trong dòng 2, thủ tục `seed` trả về giá trị hạt giống hiện tại đã được thiết lập (mặc định bằng 1). Các dòng 4 và 6 sử dụng thủ tục `next-random{}` để sinh ra hai số ngẫu nhiên với kết quả là 729239 và 1193744747. Cuối cùng, dòng 8 thực hiện việc thoát khỏi NS2.

Các dòng 9-16 lặp lại tiến trình trong các dòng 1-8. Trong các dòng 10-11, ta thấy rằng giá trị hạt giống vẫn bằng 1. Và kết quả cho thấy rằng, giá trị số ngẫu nhiên thứ nhất và thứ hai vẫn là 729239 và 1193744747. Về mặt bản chất, kết quả giữa lần chạy thứ nhất và lần chạy thứ hai là giống nhau. Để có được kết quả khác, ta cần thay đổi giá trị hạt giống.

Các dòng 17-22 cho ta thấy trong lần chạy thứ ba, giá trị hạt giống đã được thay đổi là 101. Khi đó kết quả của số ngẫu nhiên thứ nhất và thứ hai đã thay đổi là 72520690 và 308637100.

Các đặc điểm chính đối với cơ chế hạt giống trong NS2 là:

- Hạt giống chỉ định vị trí bắt đầu trong chuỗi số giả ngẫu nhiên chính là một đặc tính của RNG.

- Để sinh ra hai kết quả mô phỏng độc lập, mỗi lần mô phỏng cần thiết lập một giá trị hạt giống khác nhau.
- Khi khởi tạo, NS2 tạo ra biến `defaultRNG` là bộ sinh số ngẫu nhiên mặc định với hạt giống mặc định bằng 1. NS2 mặc định sẽ sinh ra các kết quả mô phỏng giống nhau cho mọi lần chạy.
- Khi giá trị hạt giống bằng 0, RNG sẽ thay thế giá trị hạt giống bằng giá trị thời gian và bộ đếm hiện tại. Mặc dù có xu hướng độc lập nhưng giữa hai lần chạy vẫn có thể lấy các giá trị hạt giống như nhau và do đó sinh ra kết quả giống nhau. Để đảm bảo kết quả giữa các lần chạy là độc lập, ta phải thay đổi giá trị hạt giống RNG.
- NS2 gieo hạt cho một đối tượng RNG mới là vị trí bắt đầu của chuỗi số ngẫu nhiên kế tiếp nên các đối tượng RNG khác nhau là hoàn toàn độc lập.

12.2.3. Triển khai RNG trong miền OTcl và C++

NS2 triển khai lớp C++ RNG (được liên kết với lớp OTcl có cùng tên) để sinh ra các số ngẫu nhiên (xem Chương tình 12.10). Trong mọi trường hợp, ta không cần phải tìm hiểu về chi tiết của MRG32k3a. Vì vậy, phần này chỉ đưa ra các điểm chính khi cần cấu hình và triển khai bộ sinh số ngẫu nhiên trong miền OTcl và C++. Bạn đọc có thể tìm hiểu chi tiết về cách xây dựng bộ MRG32k3a trong các file `~ns/tools/rng.cc,h`.

Chương trình 12.10. Lớp ánh xạ `RNGClass` liên kết giữa lớp `RNG` trong miền OTcl và lớp `RNG` trong miền C++.

```
      //~/ns/tools/rng.cc
1  static class RNGClass : public TclClass {
2  public:
3     RNGClass() : TclClass("RNG") {}
4     TclObject* create(int, const char*const*) {
5        return(new RNG());
6     }
7  } class_rng;
```

Các lệnh và thủ tục OTcl của lớp RNG trong miền OTcl

Trong miền OTcl, lớp RNG định nghĩa các lệnh OTcl sau:

- `seed{}`: Trả về giá trị hạt giống hiện tại của `RNG`
- `seed{n}`: Thiết lập giá trị n cho hạt giống của `RNG`
- `next-random{}`: Trả về một số ngẫu nhiên
- `next-substream{}`: Tiến tới vị trí bắt đầu của chuỗi số ngẫu nhiên tiếp theo
- `reset-start-substream{}`: Trả về vị trí bắt đầu của chuỗi số ngẫu

nhiên hiện tại.

- `normal{avg std}`: Trả về một số ngẫu nhiên theo phân bố thường với giá trị trung bình `avg` và độ lệch `std`.
- `lognormal{avg std}`: Trả về một số ngẫu nhiên theo phân bố hàm `logarit` với giá trị trung bình `avg` và độ lệch `std`.

Được định nghĩa trong file `~ns/tcl/lib/ns-random.tcl`, các thủ tục sau sinh số ngẫu nhiên với phân bố hàm mũ và phân bố bất thường:

- `exponential{mu}`: Trả về một số ngẫu nhiên theo phân bố hàm mũ với kỳ vọng là mu
- `uniform{min max}`: Trả về một số ngẫu nhiên theo phân bố bất thường trong khoảng `[min, max]`.
- `integer{k}`: Trả về một số ngẫu nhiên theo phân bố bất thường trong tập `{0,1,..., k-1}`.

Các hàm C++

Trong miền C++, các hàm chính của lớp `RNG` bao gồm:

- `set_seed(n)`: Nếu `n=0` thì thiết lập hạt giống của `RNG` là thời gian và bộ đếm hiện tại. Ngược lại, thiết lập giá trị hạt giống là `n`.
- `seed()`: Trả về giá trị hạt giống hiện tại của `RNG`.
- `next()`: Trả về một số nguyên ngẫu nhiên trong tập `{0,1,...,MAX_INT}`.
- `next_double()`: Trả về một số thực ngẫu nhiên trong khoảng `[0,1]`.
- `reset_start_substream()`: Di chuyển về vị trí bắt đầu của chuỗi số ngẫu nhiên hiện tại.
- `reset_next_substream()`: Di chuyển về vị trí bắt đầu của chuỗi số ngẫu nhiên tiếp theo.
- `uniform(k)`: Trả về một số nguyên ngẫu nhiên theo phân bố bất thường trong tập `{0,1,...,k-1}`.
- `uniform(r)`: Trả về một số thực ngẫu nhiên theo phân bố bất thường trong khoảng `[0, r]`.
- `uniform(a, b)`: Trả về một số thực ngẫu nhiên theo phân bố bất thường trong khoảng `[a, b]`.
- `exponential(k)`: Trả về một số ngẫu nhiên theo phân bố hàm mũ với kỳ vọng `k`.
- `normal(avg, std)`: Trả về một số ngẫu nhiên theo phân bố thường với giá trị trung bình `avg` và độ lệch `std`.
- `lognormal(avg, std)`: Trả về một số ngẫu nhiên theo phân bố hàm logarit với giá trị trung bình `avg` và độ lệch `std`.

14.2.4. Tính ngẫu nhiên trong kịch bản mô phỏng

Trong hầu hết mọi trường hợp, một mô phỏng thường rơi vào một trong ba kịch bản sau đây:

Thiết lập theo quyết định

Loại mô phỏng này thường được sử dụng cho việc gỡ rối (debug). Mục đích của nó là phát hiện các điểm lỗi lập trình trong mã nguồn mô phỏng hoặc để tìm hiểu các hoạt động phức tạp của một mô hình mạng nào đó. Trong cả hai trường hợp này, nên chạy chương trình dưới chế độ thiết lập theo quyết định và lặp lại việc sinh ra kết quả giống nhau. NS2 mặc định thiết lập giá trị cho hạt giống trong mô phỏng bằng 1. Do đó thiết lập theo quyết định là thiết lập mặc định cho mô phỏng bằng NS2.

Thiết lập ngẫu nhiên đơn chuỗi

Dạng đơn giản nhất của phân tích thống kê là chạy một chương trình mô phỏng nhiều lần và tính toán các độ đo thống kê như giá trị trung bình và/hoặc độ lệch tiêu chuẩn. NS2 mặc định sử dụng bộ sinh số ngẫu nhiên `defaultRNG` với hạt giống là "1" để sinh số ngẫu nhiên. Trong một hệ thống phân tích thống kê, ta cần sinh ra các kết quả khác biệt. Do đó, ta cần thiết lập giá trị hạt giống khác nhau ở các lần chạy. Trong thiết lập ngẫu nhiên đơn chuỗi, ta chỉ cần một bộ RNG. Do đó, có thể đạt được sự khác biệt giữa các lần chạy khác nhau một cách đơn giản bằng cách thiết lập giá trị hạt giống khác nhau ở các lần chạy khác nhau bằng lệnh `$defaultRNG <n>` với `<n>` là giá trị hạt giống.

Thiết lập ngẫu nhiên đa chuỗi

Trong một vài trường hợp, ta cần nhiều hơn một biến ngẫu nhiên độc lập trong chương trình mô phỏng. Ví dụ như ta có thể cần sinh ra các giá trị ngẫu nhiên cho thời gian giữa các gói tin gửi đi cũng như kích thước của gói tin. Hai biến này nên độc lập với nhau và không nên nằm trong cùng một chuỗi số ngẫu nhiên. Khi đó ta cần tạo ra hai bộ `RNG` độc lập với nhau bằng cách sử dụng lệnh "`new RNG`". Do NS2 gieo hạt mỗi bộ `RNG` với mỗi chuỗi số ngẫu nhiên khác nhau (xem Mục 12.2.2) nên xử lý ngẫu nhiên của các bộ `RNG` sẽ độc lập với nhau.

Ví dụ 12.3. Giả sử thời gian giữa các gói tin được gửi được phân bố theo hàm lũy thừa với giá trị trung bình bằng 5 và và kích thước gói tin được phân bố theo luật phân bố bất thường trong khoảng [100, 5000]. Hãy in ra 5 giá trị ngẫu nhiên đầu tiên của thời gian giữa các gói tin được gửi và kích thước gói tin

Để thực hiện yêu cầu này, ta xây dựng đoạn mã Tcl như sau:

```
1 $defaultRNG seed 101
2 set arrivalRNG [new RNG]
3 set sizeRNG [new RNG]

4 set arrival_ [new RandomVariable/Exponential]
5 $arrival_ set avg_ 5
6 $arrival_ use-rng $arrivalRNG

7 set size_ [new RandomVariable/Uniform]
8 $size_ set min_ 100
9 $size_ set max_ 5000
10    $size_ use-rng $sizeRNG

11    puts "Inter-arrival time Packet size"
12    for {set j 0} {$j < 5} {incr j} {
13        puts [format "%-8.3f %-4d" [$arrival_ value] \
                  [expr round([$size_ value])]]
14    }
```

Kết quả hiện lên trên màn hình sẽ như sau:

```
Inter-arrival time  Packet size
1.048               1880
7.919               116
8.061               3635
4.675               2110
7.201               1590
```

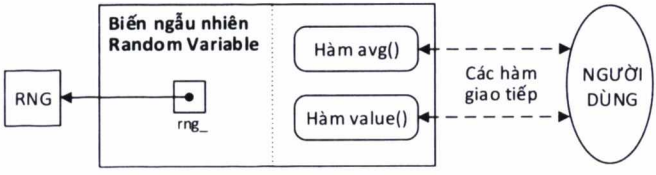
Trong đoạn mã Tcl ở trên: Dòng 4 tạo ra biến ngẫu nhiên `arrival_` phân bố theo hàm mũ với các tham số được thiết lập trong dòng 5-6. Dòng 6 tạo ra biến ngẫu nhiên `size_` phân bố theo luật phân bố bất thường với các tham số được thiết lập trong dòng 8-10. Các dòng 11-14 thực hiện việc in ra 5 giá trị ngẫu nhiên do biến `arrival_` và `size_` sinh ra. Trong Mục 12.2.5 ta sẽ thấy rằng lệnh OTcl "`value`" của lớp `RandomVariable` trả về một số ngẫu nhiên và lệnh "`use-rng`" được sử dụng để chỉ định một `RNG` cho một biến.

Theo mặc định, bộ sinh số ngẫu nhiên `defaultRNG` được sử dụng để sinh

ra số ngẫu nhiên cho cả biến arrival_ và size_. Trong trường hợp này, dòng 2 và 3 tạo ra hai RNG độc lập có tên là arrivalRNG và sizeRNG. NS2 chỉ định hai biến này là các RNG cho biến arrival_ và size_ bằng lệnh use-rng tương ứng trong dòng 6 và 10. Do các đối tượng RNG được tạo ra độc lập nên các biến ngẫu nhiên arrival_ và size_ cũng độc lập với nhau.

12.2.5. Biến ngẫu nhiên

Trong NS2, một biến ngẫu nhiên là một module sinh ra các giá trị ngẫu nhiên có thống kê theo một luật phân bố nào đó. Nó sử dụng một bộ RNG để sinh các số ngẫu nhiên và chuyển đổi các số ngẫu nhiên thành các giá trị phù hợp với một luật phân bố cho trước. Điều này được thực hiện trong lớp trừu tượng C++ có tên là RandomVariable. Hình 12.6 sau thể hiện sơ đồ lớp và khai báo của lớp được đưa ra trong Chương trình 12.11.



Hình 12.6. Sơ đồ lớp RandomVariable

Chương trình 12.11. Khai báo của lớp RandomVariable.

```
      //~/ns/tools/ranvar.h
1  class RandomVariable : public TclObject {
2  public:
3      virtual double value() = 0;
4      virtual double avg() = 0;
5      int command(int argc, const char*const* argv);
6      RandomVariable();
7      int seed(char *);
8  protected:
9      RNG* rng_;
10 };
```

Ta thấy rằng trong Chương trình 12.11 lớp RandomVariable chứa một con trỏ có tên là rng_ (dòng 9) trỏ đến một đối tượng RNG và hai hàm ảo nguyên thủy là value() (dòng 3) và avg() (dòng 4). Hàm value() sinh ra các giá trị số ngẫu nhiên, chuyển đổi các giá trị sinh ra theo một luật phân bố cho trước và trả về các giá trị đã được chuyển đổi cho đối tượng gọi hàm. Hàm avg() trả về giá trị trung bình của phân bố. Vì đây là hai hàm ảo nguyên thủy nên chúng