CHƯƠNG 10
LẬP TRÌNH VỚI ĐẶC TÍNH SCALABILITY

ThS. Trần Bá Nhiệm
Website:
sites.google.com/site/tranbanhiem
Email: tranbanhiem@gmail.com

Nội dung

- Giới thiệu
- Google search engine
- Replication & redundancy
- Các ứng dụng Scalable network
- Future proofing
- Thread pooling
- Hiện thực Thread pool
- Tránh deadlocks
- Load balancing

Giới thiệu

- Cung cấp phần mềm cho người dùng thực hiện công việc của họ được gọi là usability, nếu lượng người dùng đến 10000 thì đó là scalability
- Scalability bao gồm nhiều đặc tính của công nghệ phần mềm: ổn định, tin cậy và hiệu quả việc dùng các tài nguyên máy tính

Giới thiệu

- Mục tiêu của hệ thống scalable là phải luôn luôn sẵn sàng để sử dụng tại mọi thời điểm, giữ nguyên mức độ đáp ứng cao đối với mọi tình huống thay đổi lượng người dùng
- Scalability dưới góc độ kiến trúc phần mềm có một vài mở rộng và sửa đổi.

Giới thiệu

- Khi hệ thống phần mềm cần tăng mức độ phức tạp, nó không cần phải "đại tu" lại với mỗi tính năng bổ sung
- Xem xét các vấn đề sau:
  - Thiết kế kiến trúc scalable
  - Làm thế nào bổ sung đặc tính scalability vào ứng dụng như: load balancing, quản lý thread hiệu quả

Google search engine

- Google.com hiện tại là search engine lớn nhất trên Internet
- Cung cấp 200 triệu yêu cầu mỗi ngày
- Có trên 15000 server phân tán trên toàn thế giới
- Có thể được xem là dịch vụ Internet có đặc trưng scalable nhất

Google search engine

- Mỗi server mà Google dùng không phải luôn có sức mạnh lớn hơn một máy tính để bàn trung bình.
- Ngoài ra mỗi server có thể thường xuyên trục trặc, hư hỏng phần cứng
- Nhưng với hệ thống khắc phục rất phức tạp được Google phát minh thì một số server hư không có ảnh hưởng gì đến hiệu suất chung

Google search engine

- Google rất chú trọng đến tỷ số chi phí/hiệu suất
- Nếu để 1 server phục vụ cho 1 yêu cầu từ người dùng thì có khi phải mất hàng tuần tìm kiếm trong hàng ngàn terabyte dữ liệu mới có kết quả.
- Google chia làm 6 nhóm server: Web, document, index, spell, advertisement, Googlebot

Google search engine

- Mỗi server thực hiện công việc đặc thù của nó
- Google dùng một hệ thống DNS tinh vi để chọn Web server thích hợp nhất cho người dùng → tự động điều hướng đến data center gần nhất. Đồng thời hệ thống này cũng tính đến cân bằng tải, có thể điều hướng đến data center khác nếu có tắc nghẽn

Google search engine

- Khi có 1 yêu cầu đến, phần cứng cân bằng tải sẽ chọn 1 cluster các Web server quản lý yêu cầu đó
- Nhiệm vụ duy nhất của các Web server là sửa soạn các HTML cho client, hoàn toàn không thực hiện tìm kiếm, việc đó được ủy thác cho các index server nằm phía sau Web server

Google search engine

- Một cluster các index server gồm hàng trăm máy tính, lưu trữ một phần trong cơ sở dữ liệu có hàng ngàn terabyte.
- Một số bản sao cơ sở dữ liệu cũng có trên nhiều server khác để phòng tránh hư hỏng phần cứng
- Bản thân index là một danh sách các từ và thuật ngữ tương liên với một danh sách các mục tài liệu và độ đo thích hợp

Google search engine

- Một mục tài liệu là một tham chiếu đến 1 trang web hoặc tài liệu khác như PDF, DOC
- Thứ tự kết quả trả về phụ thuộc kết hợp trọng số liên quan đến từ khóa và thứ hạng trang (page rank) của mục tài liệu
- Page rank là độ đo mức độ phổ biến của site, số lượng link ra ngoài site, cấu trúc các link nội bộ,…

Google search engine

- Document server chứa các bản sao của toàn bộ World Wide Web trên đĩa cứng. Thực ra chúng chỉ chứa tiêu đề trang, từ khóa theo ngữ cảnh được index server cung cấp
- Mỗi data center có cluster server riêng của nó, mỗi cluster giữ ít nhất 2 bản sao của web để bảo đảm khắc phục khi server hỏng

Google search engine

- Khi tìm kiếm được thực hiện, các hệ thống ngoại vi cũng thêm nội dung của nó vào trang, gồm: kiểm tra chính tả và quảng cáo
- Trộn tất cả các thứ trên lại theo thứ tự thành kết quả trả về cho client
- Thời gian trả kết quả thường < 1 s

Google search engine

- Google bot (spider) là công cụ chạy đồng thời trên hàng ngàn máy tính cá nhân, rà soát web liên tục để thu thập các thông tin được phép truy cập, lưu nội dung vào document server và cập nhật lại index cùng với mục tài liệu, trọng số liên quan và xếp hạng trang
- Google là kiến trúc scalability tốt nhất

Replication & redundancy

- Giữ một bản sao của hệ thống cho mục đích triển khai nhanh được gọi là redundancy
- Giữ một bản sao của hệ thống đang hoạt động được gọi là replication
- Khi làm việc với dịch vụ dựa trên Internet với mức độ sẵn sàng cao thì giữ hơn 1 bản sao của hệ thống quan trọng là rất cần thiết

Replication & redundancy

- Khi đó nếu phần mềm hoặc phần cứng bị hỏng, bản sao sẽ được dùng để khắc phục nhanh chóng
- Sao lưu không cần giữ trên các máy riêng, có thể dùng RAID
- Cung cấp cơ chế redundancy giữa các máy tính chính là công việc của hệ cân bằng tải (load balancer)

Replication & redundancy

- Replication đóng vai trò cập nhật liên tục các bản sao hệ thống
- Replication đã được tích hợp vào Microsoft SQL

Các ứng dụng scalable network

- Các ứng dụng phía server thường được yêu cầu hoạt động với tối đa công suất
- Efficiency liên quan đến thông lượng của server và số lượng client mà server có thể quản lý
- Các ứng dụng scalable network hướng đến đặc tính efficiency cho các thread

Các ứng dụng scalable network

- Các ví dụ đã trình bày trong khóa học này thường cho thread mới được tạo ra khi client kết nối với server – mặc dù đơn giản nhưng đó không phải là ý tưởng tốt.
- Quản lý bên dưới các thread riêng lẻ tiêu tốn nhiều bộ nhớ và thời gian phục vụ của bộ xử lý hơn socket

Các ứng dụng scalable network

- Ví dụ: server với cấu hình Pentium IV 1,7 GHz, RAM 768-Mb kết nối với 3 client: Pentium II 233 MHz + RAM 128-Mb, Pentium II 350 MHz + RAM 128-Mb, Itanium 733 MHz + RAM 1-Gb thì server chỉ cung cấp được đến giới hạn 1008 thread, thông lượng tối đa là 2Mbps

Các ứng dụng scalable network

- Cơ chế quản lý tốt hơn được gọi là thread pooling
- Khi thread pooling được áp dụng trên server, server có thể quản lý kết nối với 12000 client mà không có trục trặc gì, thông lượng là 1,8Mbps. Với 49000 client, thì số kết nối bị hỏng chỉ chiếm 0,6%, thông lượng là 3,8Mbps, tải CPU đạt 95%

Các ứng dụng scalable network

- Để tăng sức mạnh của server, mô hình threading phải bỏ hoàn toàn. Phải dùng cơ chế asynchronous callback quản lý ở mức hệ điều hành mới giải quyết được
- Thử nghiệm: Với 12000 kết nối, thông lượng đạt 5Mbps. Với 50000 kết nối, thông lượng đạt 4,3Mbps, mức tải CPU là 65%

Thread pooling

- Mỗi máy tính có số lượng giới hạn thread có thể được xử lý tại mỗi thời điểm. Phụ thuộc vào số tài nguyên tiêu thụ thì con số này có thể khá thấp
- Để tăng số lượng thread hoặc số client kết nối đồng thời, giảm nguy cơ hệ thống sụp đổ thì lựa chọn đầu tiên là Thread pooling

Thread pooling

- Thread có thể cải thiện khả năng đáp ứng của các ứng dụng, mỗi thread tiêu tốn < 100% thời gian phục vụ của bộ xử lý
- Các hệ điều hành đa tác vụ chia sẻ tài nguyên CPU hiện có giữa các thread bằng cách chuyển nhanh giữa chúng, tạo ra cảm giác chúng chạy song song. Tần số chuyển có thể đến 60 lần/giây hoặc cao hơn nếu số lượng thread lớn

Thread pooling

- Thread nào tạm ngưng để chờ một sự kiện nào đó không tiêu tốn tài nguyên CPU, nhưng vẫn tiêu tốn tài nguyên bộ nhớ kernel.
- Số lượng tối ưu các thread cho một ứng dụng nào đó phụ thuộc vào hệ thống
- Thread pool là công cụ hữu ích để tìm ra con số tối ưu đó

Thread pooling

- Xem xét ví dụ sau để thấy tác động của việc không dùng Thread pool:

```
public static void IncrementThread()
{
  while(true)
  {
    myIncrementor++;
    long ticks = DateTime.Now.Ticks – startTime.Ticks;
    lock (this)
    {
      lblIPS.Text = "Increments per second:" +
      (myIncrementor / ticks) * 10000000;
    }
  }
}
```


Thread pooling

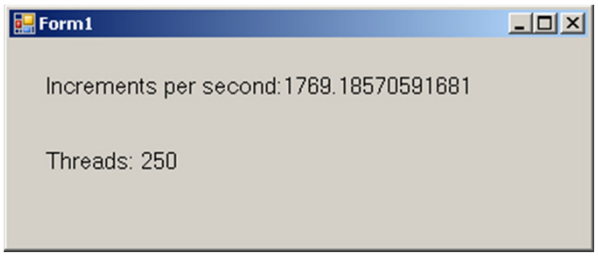
- Đoạn code trên khai báo biến public có tên MyIncrementor, sau đó đọc thời gian hệ thống, cập nhật màn hình để hiển thị thời gian tăng lên theo đơn vị giây. Phát biểu lock được dùng để bảo đảm không có thread nào khác can thiệp cập nhật màn hình tại cùng thời điểm, nếu không thì kết quả không thể tiên đoán được

Thread pooling

- Khi thread thực thi nó hoạt động với tốc độ 235 lần tăng/giây, khi thread này được khởi tạo 1000 lần và chạy đồng hành, nó tiêu tốn 60MB bộ nhớ stack, làm cho toàn hệ thống suy giảm hiệu suất.
- Với thread pool, chỉ với một số sửa đổi nhỏ trong hàm IncrementThread() và project chúng ta đã cải thiện hiệu suất một cách hết sức rõ rệt

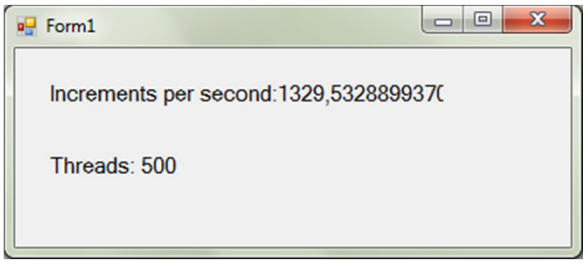
Hiện thực Thread pooling

- Thử nghiệm trên máy tính ảo cài đặt Windows Server 2003, CPU Core2 Dou (2 x 2GHz), RAM 512MB



Hiện thực Thread pooling

- Thử nghiệm trên máy tính cài đặt Windows 7 Pro, CPU Core2 Dou (2 x 2GHz), RAM 4GB



Hiện thực Thread pooling

- Tạo project mới, gồm 1 form, 2 textbox có tên lblThreads và lblIPS. Biến đánh dấu thời gian bắt đầu startTime. Mỗi thread thực hiện cộng thêm 1 vào biến public là myIncrementor – giúp chúng ta tính được hiệu suất toàn phần.
- Cải tiến thứ 1:

```
private static System.Windows.Forms.Label lblIPS;
private static System.Windows.Forms.Label
lblThreads;
```

Hiện thực Thread pooling

- Cải tiến thứ 2 trong hàm InitializeComponent(), thay vì dùng this.lblIPS chúng ta sửa thành lblIPS để truy xuất các thuộc tính:

```
lblIPS = new System.Windows.Forms.Label();
lblThreads = new System.Windows.Forms.
Label();
.v.v.
```

- Tương tự như vậy đối với lblThreads

Hiện thực Thread pooling

- Cải tiến thứ 3 trong Form1:

```
public static double myIncrementor;
public DateTime startTime;
delegate void InfoMessageDel(String info);
```

- Cải tiến thứ 4: Như chúng ta đã biết trong các chương trước, .NET không cho thread can thiệp trực tiếp vào các đối tượng mà phải thông qua hàm InfoMessageDel(…) để tránh xung đột

Hiện thực Thread pooling

- Nội dung hàm InfoMessageDel(…)

```
public static void InfoMessage(String info)
{
if (lblIPS.InvokeRequired) {
      InfoMessageDel method = new
InfoMessageDel(InfoMessage);
      lblIPS.Invoke(method, new object[] { info });
      return;
   }
   lblIPS.Text = info;
}
```


Hiện thực Thread pooling

- Xử lý sự kiện Form_Load(…)

```
private void Form1_Load(object sender, EventArgs e)
{
   int workerThreads = 0;
   int IOThreads = 0;
   ThreadPool.GetMaxThreads(out workerThreads, out IOThreads);
   lblThreads.Text = "Threads: " + workerThreads;
   for (int threads = 0; threads < workerThreads; threads++)    {
      ThreadPool.QueueUserWorkItem(new
WaitCallback(Increment));
   }
   startTime = DateTime.Now;
}
```

Hiện thực Thread pooling

- Nội dung hàm Increment(Object stateInfo)

```
void Increment(Object stateInfo)
{
    while (true){
        myIncrementor++;
        long ticks = DateTime.Now.Ticks - startTime.Ticks;
        lock (this){
            String s = "Increments per second:" +
            (myIncrementor / ticks) * 10000000;
            InfoMessage(s);
        }
    }
}
```


Tránh các deadlock

- Chúng ta hình dung một ứng dụng thực hiện lấy dữ liệu từ website vào lưu trữ vào trong cơ sở dữ liệu. Người dùng có thể dùng ứng dụng này để truy vấn vào cơ sở dữ liệu hoặc website. Ba tác vụ được thực hiện trên các thread riêng biệt. Và đồng thời với bất kỳ lý do gì thì 2 thread không được truy xuất vào website tại cùng thời điểm

Tránh các deadlock

- Thread đầu tiên có thể thực hiện tác vụ:
  - Chờ để truy cập vào website
  - Ngăn chặn thread khác truy cập vào website
  - Chờ để truy cập vào cơ sở dữ liệu
  - Ngăn chặn thread khác truy cập vào cơ sở dữ liệu
  - Lấy dữ liệu xuống, ghi vào cơ sở dữ liệu
  - Bỏ việc ngăn chặn thread khác truy cập vào website và cơ sở dữ liệu

Tránh các deadlock

- Thread thứ 2 có thể thực hiện tác vụ:
  - Chờ để truy cập vào cơ sở dữ liệu
  - Ngăn chặn thread khác truy cập vào cơ sở dữ liệu
  - Đọc từ cơ sở dữ liệu
  - Thực hiện thread thứ 3 và chờ cho nó hoàn tất
  - Bỏ việc ngăn chặn thread khác truy cập vào cơ sở dữ liệu

Tránh các deadlock

- Thread thứ 3 có thể thực hiện tác vụ:
  - Chờ để truy cập vào website
  - Ngăn chặn thread khác truy cập vào website
  - Đọc từ website
  - Bỏ việc ngăn chặn thread khác truy cập vào website
- Thread nào chạy cũng hoàn tất và không xuất hiện lỗi gì, tuy nhiên nếu ngay lúc

Tránh các deadlock

- thread 2 đọc từ cơ sở dữ liệu, trong khi đó thread 1 chờ để truy xuất vào cơ sở dữ liệu thì thread 3 sẽ bị treo
- Thread 3 không bao giờ hoàn tất bởi vì thread 1 sẽ không bao giờ được truy cập vào cơ sở dữ liệu cho đến khi thread 2 thỏa mãn điều kiện là thread 3 hoàn tất → deadlock

Tránh các deadlock

- Deadlock có thể tránh được nhờ bỏ việc ngăn chặn truy cập vào cơ sở dữ liệu trước khi thực hiện thread 3, hoặc dùng một vài phương pháp khác, nhưng vấn đề deadlock làm ảnh hưởng và cần có thiết kế lại threading để tránh lỗi bên trong ứng dụng

Load balancing

- Load balancing có nghĩa là chia khối lượng công việc giữa nhiều server bằng cách chuyển theo tỷ lệ phần trăm các yêu cầu đến từng server đó
- Phương pháp đơn giản nhất là DNS round-robin – mỗi DNS server có một số điểm đăng nhập cho cùng địa chỉ IP. Khi client yêu cầu DNS nó sẽ được nhận một

Load balancing

- trong số các địa chỉ IP để kết nối vào. Trở ngại lớn là nếu 1 server hỏng, 50% số client sẽ không nhận được dữ liệu. Tác động tương tự cũng gặp bên phía client, khi ứng dụng sẽ kết nối với một địa chỉ IP mà server lỗi khi trả dữ liệu về
- Có thể dùng phần mềm đặc biệt Microsoft Network Load Balancing Service (NLBS)

Load balancing

- NLBS cho phép nhiều máy tính hoạt động từ 1 địa chỉ IP. Bằng cách kiểm tra trạng thái của các dịch vụ như IIS trên mỗi máy tính trong cluster, mọi máy khác có thể chọn để loại trừ khỏi cluster cho đến khi chính nó thay đổi, hoặc người kỹ thuật viên can thiệp vào. Các máy tính không thực sự dùng cùng IP mà thực tế các IP được luân phiên để tạo ra tác động đó

Load balancing

- NLBS thích hợp cho các cluster nhỏ từ 4-5 server với 10-8000 client
- Ý tưởng giải pháp trên đã được hiện thực thành server ảo như Local Director của Cisco. Máy này nằm giữa router và gia đình server. Tất cả yêu cầu đến nó được nạp trực tiếp vào 1 trong 8000 client nằm phía sau.

Load balancing

- Không có giải pháp nào trong DNS round-robin, Cisco Local Director, NLBS có thể cung cấp khả năng load balancing linh hoạt. Vì thế nếu chúng ta có nhiều server với cấu hình phần cứng khác nhau thì load balancing tối ưu là trách nhiệm của mình
- Có 2 cách cung cấp load balancing theo ý muốn: bằng phần mềm hoặc phần cứng

Load balancing

- Giải pháp phần cứng có thể đạt được với một ít tùy chỉnh và 1 router. Đặc thù của nó là router có thể nhanh chóng xác định làm thế nào chuyển port và các yêu cầu được quản lý như thế nào khi các thiết lập cấu hình thay đổi. Đây là giải pháp đòi hỏi một số kinh nghiệm, nhưng chi phí thấp

Load balancing

- Giải pháp phần mềm được áp dụng cho hệ thống mà thời gian xử lý yêu cầu của client thực sự lớn hơn thời gian chuyển dữ liệu trên mạng. Cần xem xét dùng server thứ hai để chia sẻ việc xử lý. Chúng ta có thể lập trình cho các client kết nối vào để chuyển không liên tục giữa các server, nhưng nếu phần mềm bên client đã triển khai thì không thực hiện được.

Load balancing

- Phần mềm load balancing chắc chắn phải gánh vác tránh nhiệm phân tải nhưng không thực hiện được thì có nghĩa là giải pháp này không đáp ứng với mọi tình huống xảy ra
- Hiện thực phần mềm load balancing hơi giống với proxy server. Nó chấp nhận yêu cầu từ Internet và chuyển sang server nào được nó chọn

Load balancing

- Các yêu cầu đã được chuyển phải có header là HOST thay đổi để cho biết đích mới, ngược lại server có thể hủy yêu cầu. Trình load balancing có thể chuyển dựa trên bất kỳ tiêu chuẩn nào như: tải của CPU, bộ nhớ được dùng,… Nó cũng được dùng để điều khiển lỗi, nếu server ngừng, load balancing sẽ tự động điều hướng lưu thông sang các server còn hoạt động khác (dùng giải thuật round-robin)

Minh họa load balancing

- Mục tiêu: xây dựng phần mềm load balancing giữa 3 web server. Các yêu cầu từ client sẽ được điều hướng đến load-balancing server và sau đó được phân kênh đến 1 trong 3 server trên, cùng với thông báo phản hồi
- Tạo project mới, có: 1 form, 1 textbox tên tbStatus với thuộc tính multiline = true

Minh họa load balancing

- Khởi tạo 2 biến public:

  public int port;

  public int site;
- Khai báo hàm: delegate void InfoMessageDel(String info);
- Khi ứng dụng chạy, nó khởi tạo 1 thread chờ vô hạn định với các kết nối TCP từ bên ngoài:

Minh họa load balancing

```
private void Form1_Load(object sender,
EventArgs e)
{
    Thread thread = new Thread(new
ThreadStart(ListenerThread));
    thread.Start();
}
```


Minh họa load balancing

- Hàm ListenerThread thực hiện lắng nghe trên port 8889 và chờ kết nối đến. Khi nó nhận kết nối, nó khởi tạo một thực thể mới của class WebProxy (được xây dựng riêng) và khởi động method run của class này. Thiết lập các thuộc tính clientSocket, UserInterface để thực thể trên tham chiếu đến form và socket chứa yêu cầu của client

Minh họa load balancing

```
public void ListenerThread()
{
    port = 8889;
    TcpListener tcplistener = new
TcpListener(port);
    reportMessage("Listening on port " + port);
    tcplistener.Start();
    while (true){
        WebProxy webproxy = new WebProxy();
```


Minh họa load balancing

```
webproxy.UserInterface = this;
        webproxy.clientSocket =
tcplistener.AcceptSocket();
        reportMessage("New client");
        Thread thread = new Thread(new
ThreadStart(webproxy.run));
        thread.Start();
    }
}
```

Minh họa load balancing

- Hàm reportMessage() dùng phương thức lock nhằm ngăn chặn thread khác chen vào dẫn đến kết quả không đoán trước được:

```
public void reportMessage(string msg)
{
    lock (this) {
        string s = msg + "\r\n";
        InfoMessage(s);
    }
}
```


Minh họa load balancing

- Công việc load balancing nằm trong phần hiện thực giải thuật trong hàm getMirror(). Để nhằm mục đích minh họa thì giải thuật này có cơ chế tương đối đơn giản, chúng ta có thể dùng những phương pháp khác hiện đại hơn để phục vụ cho nhu cầu thực tế

Minh họa load balancing

```
public string getMirror()
{
    string Mirror = "";
    switch (site)
    {
        case 0:
            Mirror = "uk.php.net";
            site++;
            break;
        case 1:
```


Minh họa load balancing

```
            Mirror = "ca.php.net";
            site++;
            break;
        case 2:
            Mirror = "au.php.net";
            site = 0;
            break;
    }
    return Mirror;
}
```

Minh họa load balancing

- Hàm InfoMessage() thực hiện công việc ghi thông báo vào textbox như đã giới thiệu ở các chương trước, ngoài ra còn bổ sung thêm các tính năng mới theo yêu cầu của phần mềm này nhằm mục đích hiển thị các thông điệp trong textbox và trượt tự động về cuối để cho người dùng dễ theo dõi

Minh họa load balancing

```
public void InfoMessage(String info)
{
   if (tbStatus.InvokeRequired){
      InfoMessageDel method = new
InfoMessageDel(InfoMessage);
      tbStatus.Invoke(method, new object[] { info });
      return;
   }
   tbStatus.Text += info;
   tbStatus.SelectionStart = tbStatus.Text.Length;
   tbStatus.ScrollToCaret();
}
```

Minh họa load balancing

- Bước kế tiếp là xây dựng WebProxy class.
- Click phải vào project → Add → Class: đặt tên lớp mới là WebProxy.cs
- Khai báo 2 biến:

  public Socket clientSocket;

  public Form1 UserInterface;
- Khai báo các phương thức quan trọng run, getMirror

Minh họa load balancing

```
public void run()
{
    string sURL = UserInterface.getMirror();
    byte[] readIn = new byte[1024];
    int bytes = clientSocket.Receive(readIn);
    string clientmessage = Encoding.ASCII.GetString(readIn);
    clientmessage = clientmessage.Substring(0, bytes);
    int posHost = clientmessage.IndexOf("Host:");
    int posEndOfLine = clientmessage.IndexOf("\r\n",
posHost);
    clientmessage =
      clientmessage.Remove(posHost, posEndOfLine -
posHost);
```

Minh họa load balancing

```
    clientmessage =
      clientmessage.Insert(posHost, "Host: " + sURL);
    readIn = Encoding.ASCII.GetBytes(clientmessage);
    if (bytes == 0) return;
    UserInterface.reportMessage("Connection from:" +
      clientSocket.RemoteEndPoint + "\r\n");
    UserInterface.reportMessage
      ("Connecting to Site:" + sURL + "\r\n");
    relayTCP(sURL, 80, clientmessage);
    clientSocket.Close();
}
```


Minh họa load balancing

- Phương thức trên thực hiện đọc 1024 byte từ HTTP request và định dạng là ASCII
- Bỏ phần HOST header của HTTP và thay bằng HOST header trỏ đến server được xác định bởi hàm getMirror()
- Sau khi hoàn thành, chuyển điều khiển cho relayTCP để hoàn thành công việc truyền dữ liệu từ người dùng đến web server

Minh họa load balancing

```
public void relayTCP(string host, int port, string cmd)
{
    byte[] szData;
    byte[] RecvBytes = new byte[Byte.MaxValue];
    Int32 bytes;
    TcpClient TcpClientSocket = new TcpClient(host,
port);
    NetworkStream NetStrm =
TcpClientSocket.GetStream();
    szData =  System.Text.Encoding.ASCII.GetBytes(
cmd.ToCharArray());
    NetStrm.Write(szData, 0, szData.Length);
```


Minh họa load balancing

```
  while (true) {
    try   {
      bytes = NetStrm.Read(RecvBytes, 0, RecvBytes.Length);
      clientSocket.Send(RecvBytes, bytes, SocketFlags.None);
      if (bytes <= 0) break;
    }
    catch  {
      UserInterface.reportMessage("Failed connect");
      break;
    }
  }
}
```

Minh họa load balancing

- Hàm trên thực hiện việc truyền dữ liệu. Nó mở kết nối TCP ở port 80 và sau đó gửi phần HTTP header đã chỉnh sửa cho client. Ngay sau đó, nó vào vòng lặp, đọc từng 256 byte dữ liệu từ web server và gửi cho client.
- Nếu có bất kỳ lỗi gì xảy ra, vòng lặp ngắt và hàm kết thúc

Minh họa load balancing

Bài tập

- Cài đặt các chương trình đã minh họa trong bài giảng của chương bằng ngôn ngữ C# hoặc VB.NET