TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

──────── \* ───────

ĐỒ ÁN

**TỐT NGHIỆP ĐẠI HỌC**

NGÀNH CÔNG NGHỆ THÔNG TIN

**NGHIÊN CỨU VỀ NETTY FRAMEWORK VÀ XÂY DỰNG HỆ THỐNG GAME ONLINE TRÊN NỀN TẢNG WEB**

Sinh viên thực hiện : **Nguyễn Xuân Giáp**

Lớp VN – K57

Giáo viên hướng dẫn: **TS** **Nguyễn Tuấn Dũng**

HÀ NỘI 5-2017

# PHIẾU GIAO NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP

1. Thông tin về sinh viên

Họ và tên sinh viên: Nguyễn Xuân Giáp

Điện thoại liên lạc: 0969626066 Email: giapnx.ict@gmail.com

Lớp: VN – K57 Hệ đào tạo: Đại học chính quy

Đồ án tốt nghiệp được thực hiện tại: Chương trình tiên tiến Việt Nhật – Đại học Bách Khoa Hà Nội

Thời gian làm ĐATN: Từ ngày / / đến / /

2. Mục đích nội dung của ĐATN

Đồ án tập trung nghiên cứu về Netty Framwork và xây dựng hệ thống Game Online trên

nền tảng Web. Mục đích của đề tài là nắm được tư tưởng và các thành phần của Netty Framework, từ đó áp dụng kiến thức để xây dựng một Server có thể phục vụ nhiều người cùng chơi online cùng lúc.

3. Các nhiệm vụ cụ thể của ĐATN

- Nghiên cứu mô hình Client – Server.

- Nghiên cứu về Java NIO và Netty Framwork.

- Nghiên cứu mô hình truyền tin qua Socket và Websocket.

- Phân tích, thiết kế hệ thống game online trên nền tảng Web.

- Xây dựng chương trình và kiểm thử.

- Tổng kết và đánh giá.

4. Lời cam đoan của sinh viên:

Tôi - *Nguyễn Xuân Giáp* - cam kết ĐATN là công trình nghiên cứu của bản thân tôi dưới sự hướng dẫn của *học hàm TS Nguyễn Tuấn Dũng*.

Các kết quả nêu trong ĐATN là trung thực, không phải là sao chép toàn văn của bất kỳ công trình nào khác.

|  |  |
| --- | --- |
|  | *Hà Nội, ngày tháng 5 năm 2017*  Tác giả ĐATN  *Nguyễn Xuân Giáp* |

5. Xác nhận của giáo viên hướng dẫn về mức độ hoàn thành của ĐATN và cho phép bảo vệ:

|  |  |
| --- | --- |
|  | *Hà Nội, ngày tháng 6 năm 2017*  Giáo viên hướng dẫn  *TS Nguyễn Tuấn Dũng* |

# LỜI CẢM ƠN

Khoá luận này được em hoàn thành qua quá trình tích luỹ kiến thức, kỹ năng, phương pháp sau năm năm học tập, nghiên cứu trên giảng đường và thực tế tại doanh nghiệp. Để đạt được thành quả này không không chỉ là sự cố gắng của bản thân mà còn nhờ sự quan tâm hướng dẫn, giúp đỡ của nhiều cá nhân trong và ngoài trường.

Em xin gửi lời cảm ơn thầy cô giáo trong trường Đại học Bách Khoa nói chung và các thầy cô trong viện Công nghệ thông tin và truyền thông nói riêng, đã tận tình giảng dạy, trang bị cho em những kiến thức quý báu đồng thời tạo điều kiện cho em được thực tập và hoàn thành đề tài luận văn tốt nghiệp của mình.

Em xin gửi lời cảm ơn tới thầy giáo, TS Nguyễn Tuấn Dũng – Giảng viên bộ môn Khoa học máy tính, viện Công nghệ thông tin và truyền thông đại học Bách Khoa Hà Nội đã tận tình hướng dẫn em trong suốt quá trình làm đồ án tốt nghiệp.

Mặc dù đã cố gắng hết sức và nhận được nhiều sự giúp đỡ nhưng do thời gian, kiến thức và kinh nghiệm thực tế có hạn nên đề tài không tránh khỏi có những thiếu sót. Em rất mong nhận được sự đóng góp ý kiến của các thầy cô để em có thể hoàn thiện hơn nữa đề tài nghiên cứu.

|  |
| --- |
| Hà Nội, tháng 5 năm 2017  Sinh viên: Nguyễn Xuân Giáp |

# 

# TÓM TẮT NỘI DUNG ĐỒ ÁN TỐT NGHIỆP

# Lời dẫn

Nội dung đồ án bao gồm:

**Phần 1: Đặt vấn đề, định hướng và giải pháp:**

**Phần 2: Nền tảng phát triển**

**Phần 3: Netty Framework**

**Phần 4: Phân tích và thiết kế hệ thống**

**Phần 5: Các kết quả đạt được**

**Kết luận**

# ABSTRACT OF THESIS

**[Mục này không bắt buộc những khuyến khích nên có]**

. Điền tóm tắt nội dung của ĐATN bằng tiếng Anh.

.*Font Times New Roman (Unicode), size font 12, Line spacing: multiple 1.2*

*Page setup: Paper size (A4); margin (top: 20mm, bottom: 20mm, right: 20mm,   
left: 35mm)*

.

.

.

.

.

**MỤC LỤC**

**DANH MỤC CÁC BẢNG**

**DANH MỤC HÌNH VẼ**

**DANH MỤC TỪ VIẾT TẮT**

**MỞ ĐẦU**

Chém về NIO, Netty và từ đó áp dụng xây dựng server

**CHƯƠNG 1: ĐẶT VẤN ĐỀ, ĐỊNH HƯỚNG VÀ GIẢI PHÁP**

- Trình bày chi tiết về nhiệm vụ (đề tài) cần thực hiện trong khuôn khổ ĐATN (mô tả bài toán, các vấn đề cần giải quyết);

- Trình bày định hướng giải quyết các vấn đề đặt ra;

- Giới thiệu tóm tắt về cơ sở lý thuyết và công cụ được lựa chọn để giải quyết các vấn đề đặt ra;

**Phần 2**

- Trình bày chi tiết kết quả phân tích, thiết kế hệ thống (giải pháp);

- Trình bày chi tiết kết quả thực hiện, cài đặt và thử nghiệm hệ thống (giải pháp);

- Đánh giá ưu, nhược điểm và khả năng ứng dụng của hệ thống (giải pháp)(thông qua thử nghiệm thực tế hoặc mô phỏng), so sánh với các sản phẩm cùng loại (nếu có).

**CHƯƠNG 2: NỀN TẢNG PHÁT TRIỂN**

**1. Mô hình Web Client – Server**

**Mô hình client-server** là một mô hình nổi tiếng trong mạng máy tính, được áp dụng rất rộng rãi và là mô hình của mọi trang web hiện có. Ý tưởng của mô hình này là máy con (đóng vài trò là máy khách) gửi một yêu cầu (request) để máy chủ (đóng vai trò người cung ứng dịch vụ), máy chủ sẽ xử lý và trả kết quả về cho máy khách.

Một mô hình ngược lại là mô hình master-slaver, trong đó máy chủ (đóng vai trò ông chủ) sẽ gửi dữ liệu đến máy con (đóng vai trò nô lệ) bất kể máy con có cần hay không.

**2. Công nghệ sử dụng bên phía Client**

**2.1 HTML5 Websocket**

HTTP 101 (hay đúng hơn, HTTP / 1.0 và HTTP / 1,1). Trong kiến trúc cũ, kết nối được xử lý bởi HTTP / 1.0 và HTTP / 1.1. HTTP là một giao thức để đáp ứng yêu cầu trong một mô hình client / server. Với 1 client (thường là một trình duyệt web) gửi một yêu cầu HTTP đến server, server xử lý và gửi trả kết quả trả lại cho client. Chẳng hạn như một trang HTML, cũng như các thông tin về trang. HTTP cũng được thiết kế cho việc lấy các tài liệu. HTTP / 1.0 là đủ cho một đơn yêu cầu tài liệu từ một server.

Tuy nhiên, như các trang web lớn hơn thì sự tương tác này là lớn hơn và dữ liệu cũng lớn hơn rất nhiều thì tài liệu đơn giản chia sẻ và bắt đầu bao gồm tương tác nhiều hơn đòi hỏi thời gian phản ứng phải nhanh hơn giữa các yêu cầu trình duyệt và đáp ứng server.

Với HTTP, HTTP / 1.1 Với việc thêm vào các kết nối tái sử dụng, trình duyệt có thể khởi tạo một kết nối đến một máy chủ web để lấy các trang HTML, sau đó sử dụng cùng một kết nối để lấy nguồn tài nguyên như hình ảnh, chữ viết, và như vậy. HTTP / 1.1 giảm độ trễ giữa yêu cầu bằng cách giảm số lượng các kết nối đã được thực hiện từ các client đến các máy chủ.

Với HTTP / 1.0 và HTTP / 1.1, có những điểm không hiệu quả xuất phát từ những điểm sau đây:

* HTTP được thiết kế để chia sẻ tài liệu, không phải là các ứng dụng tương tác phong phú.
* Độ trễ lớn, nếu client thực hiện request liên tục lên server, rất nhiều request thừa, không có dữ liệu mới cũng phải gửi, ảnh hưởng lớn đến performance của server.

Về bản chất, HTTP cũng là half duplex, có nghĩa là lưu lượng truyền tin theo một hướng duy nhất tại một thời điểm, điều này gây lãng phí và kém hiệu quả.

Ra đời cách đây không lâu HTML5 Websocket tạo kết nối giữa client và server qua một cổng nào đó. Không sử dụng các request HTTP thông thường, WebSocket mở một kết nối duy nhất, hai chiều và tái sử dụng cùng một kết nối. Tất cả dữ liệu giao tiếp giữa client-server sẽ được gửi trực tiếp qua port này làm cho thông tin được gửi đi nhanh chóng và liên tục khi cần thiết. Web socket giả quyết được tất cả khuyết điểm ở các cách phía trên, nhưng khuyết điểm lớn nhất websocket là không hỗ trợ các trình duyệt cũ.

WebSocket làm giảm độ trễ bởi vì một khi kết nối WebSocket được thành lập, server không cần phải chờ đợi cho một yêu cầu từ client. Tương tự như vậy, client có thể gửi tin nhắn đến server bất cứ lúc nào. Yêu cầu duy nhất này giúp làm giảm đáng kể độ trễ, mà sẽ gửi một yêu cầu trong khoảng thời gian, cho dù thông điệp có sẵn.

Khi một Websocket kết nối được mở thông qua HTTP request đó chính là liên kết giữa WebSockets với những header đặc đặc biệt. Kết nối này sẽ được duy trì nhằm giúp cho người sử dụng có thể viết và nhận dữ liệu bằng Javascript giống như khi họ đang sử dụng một TCP socket đơn thuần.

Websocket là một công nghệ dùng để hỗ trợ cho việc giao tiếp hai chiều giữa sever và client thông qua việc sử dụng một TCP socket nhằm tạo ra một liên kết hiệu quả mà lại ít tốn kém. Với mục đích ban đầu thiết kế ra chỉ để chuyên sử dụng cho các ứng dụng web, nhưng lập trình viên vẫn có thể sử dụng chúng trong bất kỳ loại ứng dụng nào khác.

**2.1.1 So sánh giữa HTTP và WebSocket**

Với kỹ thuật Ajax, khi một dữ liệu được truyền tải đến thông qua giao thức HTTP sẽ chứa quá nhiều dữ liệu không cần thiết đến trong phần header. Với một header request/reponse của HTTP sau khi đã kết nối sẽ có kích cỡ khoảng 871 byte, trong khi đó đối với Websocket kích cỡ này chỉ rơi vào khoảng 2 byte.

Ví dụ thực tế trường hợp trên đó là khi bạn tạo ra một ứng dụng chơi game có thể sẽ có tới 10,000 lượt truy cập cùng một lúc, và mỗi giây họ sẽ phải gửi và nhận dữ liệu từ sever. Chúng ta cùng so sánh với lượng dữ liệu từ header mà giao thức HTTP và WebSocket trong mỗi giây:

* *HTTP: 871x10,000 = 8,710,000 bytes = 69,680,000 bits per second (66 Mbps)*
* *Websocket: 2x10,000 = 20,000 bytes = 160,000 bits per second (0.153 Kbps)*

Qua ví dụ trên có thể thấy, với giao thức HTTP truyền thống chỉ riêng phần header thôi mà đã chiếm một phần dung lượng rất lớn rồi.

Hiện tại Websocket đã được hỗ trợ trên 94% các trình duyệt (trừ Opera Mini). Chi tiết: <http://caniuse.com/#search=websocket>.

**2.1.2 Ưu nhược điểm của Websocket**

+ Ưu điểm:

* Giao tiếp thời gian thực hiệu quả hơn, tiết kiệm băng thông, CPU, và giảm độ trễ.
* WebSockets cung cấp khả năng giao tiếp hai chiều mạnh mẽ, có độ trễ thấp và dễ xử lý lỗi.
* API cũng rất dễ sử dụng trực tiếp mà không cần bất kỳ các tầng bổ sung nào, so với Comet, thường đòi hỏi một thư viện tốt để xử lý kết nối lại, thời gian chờ timeout, các Ajax request (yêu cầu Ajax), các tin báo nhận và các dạng truyền tải tùy chọn khác nhau (Ajax long-polling và jsonp polling).
* Không cần phải có nhiều kết nối như phương pháp Comet long-polling và cũng không có những nhược điểm như Comet streaming.
* Websocket cho thấy sự đơn giản:
  + Sử dụng Ajax với long polling qua HTTP có ưu điểm là hỗ trợ nhiều trình duyệt, tạo ra ứng dụng thời gian thực nhưng việc giữ request làm tăng tải cho server, đối với các server bình thường, xử lý đồng bộ một thread cho một request, thì nếu có nhiều request sau đó, sẽ gây ra chậm.
  + WebSocket cho phép bạn đơn giản hóa đáng kể hướng kết nối thông tin liên lạc trong các ứng dụng thời gian thực.

+ Nhược điểm:

* Chưa được hỗ trợ trên trình duyệt Opera Mini và một số trình duyệt cũ của IE, Firefox, Chrome
* Không có phạm vi yêu cầu nào. Do WebSocket là một TCP socket chứ không phải là HTTP request, nên không dễ sử dụng các dịch vụ có phạm vi yêu cầu, như SessionInViewFilter của Hibernate. Hibernate là một framework kinh điển cung cấp một bộ lọc xung quanh một HTTP request. Khi bắt đầu một request, nó sẽ thiết lập một contest (chứa các transaction và liên kết JDBC) được ràng buộc với luồng request. Khi request đó kết thúc, bộ lọc hủy bỏ contest này.

**2.1.3 Các thành phần của WebSocket**

**2.1.3.1 Các thuộc tính**

|  |  |
| --- | --- |
| **Thuộc tính** | **Miêu tả** |
| Socket.readyState | Thuộc tính chỉ đọc **readyState** biểu diễn trạng thái kết nối. Nó có các giá trị sau:   * Giá trị 0 chỉ rằng kết nối vẫn chưa được thành lập. * Giá trị 1 chỉ rằng kết nối đã thành lập và có thể giao tiếp. * Giá trị 2 chỉ rằng kết nối đang qua handshake. * Giá trị 3 chỉ rằng kết nối đã được đóng hoặc không thể được mở. |
| Socket.bufferedAmount | Thuộc tính chỉ đọc **bufferedAmount** biểu diễn số byte của UTF-8 mà đã được xếp hàng bởi sử dụng phương thức send() |

**2.1.3.2 Các sự kiện**

|  |  |  |
| --- | --- | --- |
| **Sự kiện** | **Event Handler** | **Miêu tả** |
| open | Socket.onopen | Sự kiện này xảy ra khi kết nối Socket được thành lập. |
| message | Socket.onmessage | Sự kiện này chỉ xảy ra khi Client nhận dữ liệu từ Server. |
| error | Socket.onerror | Sự kiện này xảy ra khi có bất kỳ lỗi nào phát sinh trong quá trình giao tiếp |
| close | Socket.onclose | Sự kiên này xảy ra khi kết nối được đóng |

**2.1.3.3 Phương thức**

|  |  |
| --- | --- |
| **Phương thức** | **Miêu tả** |
| Socket.send() | Phương thức send(data) truyền tải dữ liệu sử dụng kết nối đó. |
| Socket.close() | Phương thức close() sẽ được sử dụng để kết thúc bất kỳ kết nối đang tồn tại. |

**2.2 jQuery**

**3 Công nghệ sử dụng bên phía Server**

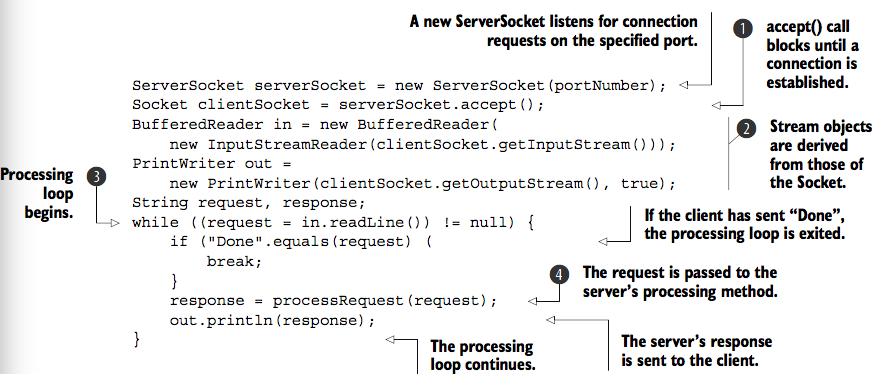
**3.1 Netty – asynchronous and event-driven**

In this first chapter, we’ll begin with background on high-performance networking, particularly its implementation in the Java Development Kit (JDK).

**3.1.1 Mạng (Networking) trong Java**

Từ những ngày đầu của mạng - Networking, các nhà phát triển đã dành rất nhiều thời gian để tìm hiểu sự phức tạp của những thư viện socket ngôn ngữ C và giải quyết các vấn đề của họ trên các hệ điều hành khác nhau. Các phiên bản Java đầu tiên (1995-2002) đã giới thiệu vẻ ngoài hướng đối tượng để giấu một số chi tiết khốc liệt, tuy nhiên việc tạo ra một giao thức client / server phức tạp vẫn đòi hỏi rất nhiều mã lệnh để làm cho nó hoạt động trơn tru.

Các API Java đầu tiên (java.net) chỉ hỗ trợ các chức năng chặn (blocking) được cung cấp bới các thư viện socket của hệ thống. Ví dụ về thực hiện Socket API căn bản:

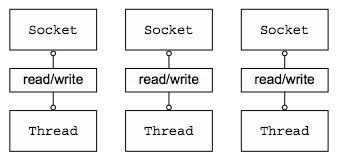


Hình x: Ví dụ về Blocking I/O

Một số điểm quan trọng nhất:

* Phương thức *accept ()*sẽ chặn (block) cho đến khi kết nối được thiết lập trên *ServerSocket* ❶, sau đó trả về một *Socket* mới cho giao tiếp giữa client và server. *ServerSocket* sẽ tiếp tục lắng nghe các kết nối đang đến.
* Một *BufferedReader* và một *PrintWriter* xuất phát từ đầu vào và đầu ra của *Socket* ❷. Các khuôn (form) đọc văn bản từ một luồng đầu vào ký tự, sau đó in định dạng đại diện của các đối tượng cho một luồng đầu ra văn bản.
* Phương thức *readLine* () đọc chuỗi kỹ tự đến.
* Yêu cầu (request) của client được xử lý ở ❹.

Đoạn mã này sẽ chỉ xử lý một kết nối tại một thời điểm. Để quản lý nhiều client đồng thời, cần cấp phát *Thread* mới cho mỗi client *Socket* mới, như thể hiện trong hình xx.



Hình x: Đa kết nối sử dụng blocking I/O

Vấn đề gặp phải khi giải quyết theo hướng tiếp cận này.

Thứ nhất, tại bất kỳ thời điểm nào nhiều thread có thể được ngủ đông (không làm việc), chỉ cần chờ đợi dữ liệu đầu vào hoặc đầu ra xuất hiện trên tuyến đường. Đây có thể là sự lãng phí tài nguyên.

Thứ hai, mỗi thread yêu cầu phân bổ bộ nhớ ngăn xếp (stack memory) có kích thước mặc định từ 64 KB đến 1 MB, tùy thuộc vào hệ điều hành.

Thứ ba, ngay cả khi một máy ảo Java (JVM) có thể hỗ trợ một số lượng lớn các thread, thì chi phí chuyển đổi phần đầu của phạm vi sẽ trở nên rắc tốn kém khi đạt đến giới hạn, ví dụ khi đạt đến 10.000 kết nối.

Mặc dù cách tiếp cận này có thể chấp nhận được đối với một số lượng nhỏ các client, nhưng trong trường hợp cần tài nguyên để hỗ trợ 100.000 hoặc nhiều kết nối đồng thời thì chúng sẽ gặp rất nhiều khó khăn với hướng tiếp cận này. Và đã có một sự thay thế.

**3.1.1.2 Java NIO (Non-blocking I/O)**

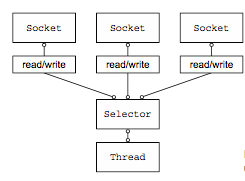
Các thư viện native socket đã bao gồm các lời gọi không chặn (*non-blocking*), cho phép kiểm soát hiệu quả hơn việc sử dụng các tài nguyên mạng.

* Sử dụng *setockopt* () cấu hình các socket sao cho các lời gọi đọc / ghi sẽ trả về ngay nếu không có dữ liệu.
* Có thể đăng ký một tập hợp các *non-blocking* socket bằng API thông báo sự kiện của hệ thống để xác định liệu có bất kỳ dữ liệu nào có sẵn để đọc hoặc ghi hay không.

Java hỗ trợ *non-blocking* I / O được giới thiệu vào năm 2002, với JDK 1.4 gói java.nio.

**3.1.1.3 Selectors**

Ở hình x, cho thấy thiết kế non-blocking đã loại bỏ được hầu hết các các nhược điểm được mô tả ở trên.



Hình x: Non-blocking sử dụng Selector

Lớp java.nio.channels.Selector là mấu chốt của việc triển khai non-blocking I/O của Java. Nó sử dụng API thông báo sự kiện để chỉ ra, trong số các non-blocking socket, socket nào đã sẵn sàng cho I/O. Bởi vì bất kỳ thao tác đọc hoặc ghi có thể được kiểm tra trạng thái hoàn thành của nó bất cứ lúc nào trong chỉ một thread, như thể hiện trong hình 1.2, có thể xử lý nhiều kết nối đồng thời.

Nhìn chung, mô hình này cung cấp quản lý tài nguyên tốt hơn nhiều so với mô hình blocking I/O:

* Nhiều kết nối có thể được xử lý với ít thread hơn, và do đó chi phí quản lý bộ nhớ sẽ ít hơn rất nhiều.
* Các thread có thể xác định lại mục tiêu cho các nhiệm vụ khác khi không có xử lý I/O.

**3.1.2 Tổng quan về Netty Famework**

Trong lĩnh vực mạng, Netty được đánh giá là framework ưu việc nhất cho Java. Khai thác sức mạnh của các API nâng cao của Java, Netty tập trung vào những gì quan trọng nhất của sản phẩm.

Các đặc điểm chính của Netty framework

|  |  |
| --- | --- |
| Đặc điểm | Tính năng |
| Thiết kế  - Design | API được thống nhất cho nhiều kiểu vận chuyển (transport), cả blocking và non-blocking.  Mô hình thread đơn giản nhưng mạnh mẽ.  Kết nối các thành phần logic để hỗ trợ tái sử dụng. |
| Dễ sử dụng  - Ease of use | Đầy đủ Javadoc hỗ trợ và nhiều mẫu ví dụ.  Không yêu cầu về ràng buộc phải trên JDK 1.6+ (ngoài một số tính năng tùy chọn có thể yêu cầu Java 1.7+). |
| Hiệu suất  - Performance | Thông lượng tốt hơn và độ trễ thấp hơn các API core Java. Giảm tiêu thụ tài nguyên nhờ tính kết hợp lại (pooling) và tái sử dụng (reuse). Tối thiểu sao chép bộ nhớ. |
| Mạnh mẽ  - Robustness | Không có *OutOfMemoryError* do kết nối chậm, nhanh hoặc quá tải. Loại bỏ tỷ lệ đọc/ghi không đồng đều của các ứng dụng NIO trong các mạng tốc độ cao. |
| Bảo mật  - Security | Hỗ trợ SSL / TLS và StartTLS.  Có thể sử dụng trong các môi trường hạn chế như Applet hoặc OSGI. |
| Hướng cộng đồng  - Community-driven | Phát hành sớm và thường xuyên. |

Bảng x: Tóm tắt các tính năng của Netty.

Có rất nhiều các công ty lớn như Apple, Twitter, Facebook, Google, Square và Instagram đã sử dụng framwork Netty trong quá trình phát triển của họ. Ví du, Framework Finagle của Twitter dựa vào Netty để liên lạc giữa các hệ thống, hay Facebook sử dụng Netty trong Nifty – một triển khai các client và server Thrift trên Netty. Netty đã được hưởng lợi từ các dự án này, tăng cường cả phạm vi và tính linh hoạt của nó thông qua việc thực hiện các giao thức như FTP, SMTP, HTTP, và Web-Socket, có thể truyền gói tin theo cả nhị phân – binary và văn bản - text.

**3.1.2.1 Không đồng bộ (Asynchronous) và điều khiển sự kiện (event-driven)**

Một hệ thống không đồng bộ và điều khiển sự kiện thể hiện một hành vi đặc biệt, nó có thể phản ứng lại các sự kiện xảy ra bất cứ lúc nào và theo bất kỳ thứ tự nào.

Khả năng này rất quan trọng để đạt được mức độ cao nhất của khả năng mở rộng, được định nghĩa là "khả năng của một hệ thống, mạng hoặc quy trình để xử lý một lượng công việc ngày càng tăng hoặc khả năng mở rộng để đáp ứng sự tăng trưởng đó”.

Kết nối giữa bất đồng bộ (asynchrony) và khả năng mở rộng (scalability) là gì?

* Các lời gọi non-blocking network không cần chờ đợi cho tới khi hoàn thành một công việc hay một hành động. Asynchronous I/O được xây dựng trên tính năng này và tiến thêm một bước nữa: phương pháp không đồng bộ trả về ngay lập tức và thông báo cho người dùng khi nó hoàn thành, trực tiếp hoặc sau đó.
* Selector cho phép chúng ta giám sát nhiều kết nối cho các sự kiện với ít thread hơn.

Với non-blocking I/O chúng ta có thể xử lý số lượng rất nhiều các sự kiện nhanh hơn và hiệu quả hơn nhiều so với blocking I/O.

**3.1.3 Giới thiệu các thành phần cốt lỗi của Netty.**

Các khối chính xây dựng lên Netty Framework:

* Channels
* Callbacks
* Futures
* Events and handlers

**3.1.3.1 Channels**

Channel là một cấu trúc cơ bản của Java NIO. Nó đại diện cho một kết nối mở với một thực thể như thiết bị phần cứng, tệp tin, network socket hoặc một phần của chương trình có khả năng thực hiện một hoặc nhiều thao tác I/O khác biệt, ví dụ như đọc hoặc ghi.

Channel như một phương tiện cho dữ liệu đến (gửi đến) và gửi đi (ra bên ngoài). Như vậy, nó có thể được mở, đóng, kết nối hoặc ngắt kết nối.

**3.1.3.2 Callbacks**

Một callback chỉ đơn giản là một phương thức, một tham chiếu đã được cung cấp cho một phương thức khác. Callbacks được sử dụng trong một loạt các tình huống lập trình và đại diện cho một trong những cách phổ biến nhất để thông báo cho một bên quan tâm rằng một hoạt động đã hoàn thành.

Netty sử dụng callbacks khi xử lý sự kiện. Khi callback được kích hoạt, sự kiện có thể được xử lý bởi một thực thi của giao diện ChannelHandler. Ví dụ: khi một kết nối mới đã được thiết lập thì ChannelHandler callback phương thức channelActive () sẽ được gọi và sẽ in một tin nhắn.



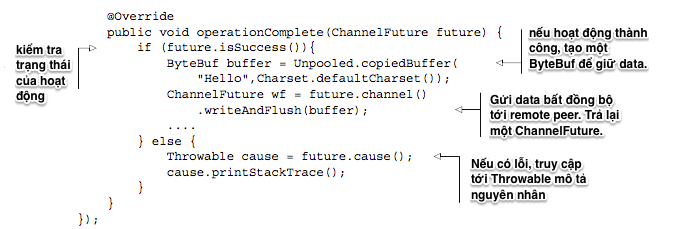
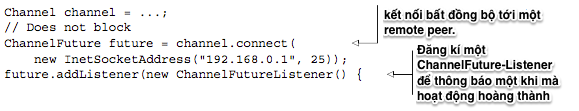
Hình x: ChannelHandler được kích hoạt bởi một callback

**3.1.3.3 Futures**

Một Future cung cấp một cách khác để thông báo cho một ứng dụng khi một hoạt động đã hoàn thành. Đối tượng này hoạt động như một trình giữ chỗ cho kết quả của hoạt động không đồng bộ. Nó sẽ hoàn thành tại một số thời điểm trong tương lai và cung cấp quyền truy cập tới kết quả.

JDK có interface java.util.concurrent.Future, nhưng được cung cấp các cài đặt cho phép kiểm tra thủ công dù hoạt động đã hoàn thành hay để ngăn chặn cho đến khi nó được thực hiện. Điều này là gây nhiều trở ngại, vì vậy Netty cung cấp việc triển khai riêng của nó, ChannelFuture, để sử dụng khi một thao tác không đồng bộ được thực hiện.

ChannelFuture cung cấp các phương pháp bổ sung cho phép chúng ta đăng ký một hoặc nhiều thể hiện (instances) của ChannelFutureListener. Phương thức callback của listener, operationComplete () được gọi khi hoạt động đã hoàn tất. Listener có thể xác định xem hoạt động có thành công hay không. Nếu không thành công sẽ phát sinh ra ngoại lệ (Exception). Như vậy, cơ chế thông báo được cung cấp bởi ChannelFutureListener loại bỏ việc phải kiểm tra theo cách thủ công sự hoàn thành của hoạt động.



Hình x: Hoạt động của callback

Callbacks và Futures là các cơ chế bổ sung. Kết hợp chúng tạo thành một trong những khối then chốt chính để xây dựng Netty.

**3.1.3.4 Events and handlers**

Netty sử dụng các sự kiện riêng biệt để thông báo về những thay đổi của trạng thái hoặc tình trạng hoạt động. Điều này cho phép kích hoạt hành động thích hợp dựa trên sự kiện đã xảy ra. Những hành động như vậy có thể bao gồm

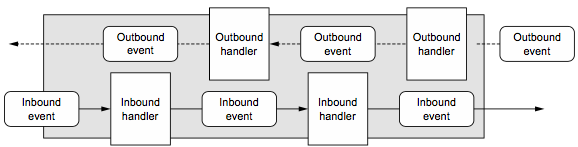
* Logging
* Data transformation
* Flow-control
* Application logic

Netty là networking framework, vì vậy các sự kiện được phân loại theo luồng dữ liệu gửi đến hoặc đi ra. Các sự kiện có thể được kích hoạt bởi dữ liệu gửi đi hoặc sự thay đổi trạng thái liên quan bao gồm

* Active or inactive connections
* Data reads
* User events
* Error events

Sự kiện đi ra (outbound) là kết quả của một hoạt động, cái sẽ kích hoạt một hành động trong tương lai, có thể là

* Mở hoặc đóng một kết nối tới một remote peer
* Writing hoặc flushing dữ liệu tới socket



Hình x: Luồng data vào/ra thông qua một chuỗi của ChannelHandlers

**CHƯƠNG 3: CÁC THÀNH PHẦN CỦA NETTY FRAMEWORK**

Trong chương này, nghiên cứu cụ thể các thành phần của Netty một cách chi tiết hơn, chú ý đến cách họ hợp tác để hỗ trợ tốt nhất kiến trúc này.

**3.1 Channel, EventLoop và ChannelFuture**

**3.1.1 Interface Channel**

Các hoạt động I/O cơ bản (bind (), connect (), read () và write () được cung cấp bởi network transport. Trong mạng Network dựa trên Java, cấu trúc cơ bản là lớp Socket. Interface Channel của Netty cung cấp một API làm giảm đáng kể sự phức tạp của việc làm việc trực tiếp với Sockets. Ngoài ra, Channel là gốc của một hệ thống phân cấp lớp mở rộng có nhiều thể hiện được xác định trước:

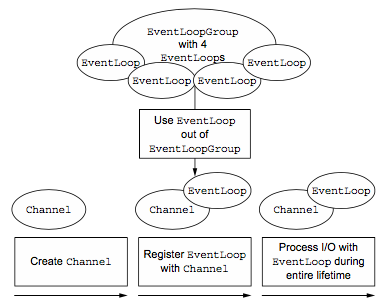
* EmbeddedChannel
* LocalServerChannel
* NioDatagramChannel
* NioSctpChannel
* NioSocketChannel

**3.1.2 Interface EventLoop**

EventLoop định nghĩa sự trừu tượng cốt lõi của Netty để xử lý các sự kiện xảy ra trong suốt vòng đời của một kết nối. Hình x minh họa mối quan hệ giữa các kênh, EventLoops, Threads, và EventLoopGroups.

Những mối quan hệ:

* Một EventLoopGroup chứa một hoặc nhiều EventLoops.
* Một EventLoop bị ràng buộc bởi một Thread duy nhất trong suốt vòng đời của nó.
* Tất cả các sự kiện I / O được xử lý bởi một EventLoop được xử lý trên Thread dành riêng của nó.
* Channel được đăng ký trong suốt vòng đời với một EventLoop duy nhất.
* Một EventLoop duy nhất có thể được gán cho một hoặc nhiều Channel.



Hình x: Channels, EventLoops và EventLoopGroups

**3.1.3 Interface ChannelFuture**

Tất cả các hoạt động I/O trong Netty đều không đồng bộ. Do một thao tác không thể trở lại ngay lập tức, vì vậy cần một cách để xác định kết quả của nó vào một thời điểm sau. Với mục đích này, Netty cung cấp ChannelFuture, phương thức addListener () của nó sẽ đăng ký một ChannelFutureListener để được thông báo khi một thao tác đã hoàn thành (đã hoặc chưa thành công).

**3.2 ChannelHandler and ChannelPipeline**

Ở phần này, chúng ta sẽ xem xét chi tiết hơn các thành phần quản lý lưu lượng dữ liệu và thực hiện xử lý logic trong các ứng dụng.

**3.2.1 Interface ChannelHandler**

Từ quan điểm của nhà phát triển ứng dụng, thành phần chính của Netty là ChannelHandler, nó đóng vai trò là vùng chứa cho tất cả phần logic của ứng dụng áp dụng cho việc xử lý dữ liệu trong và ngoài. Điều này là có thể bởi vì các phương thức ChannelHandler được kích hoạt bởi các sự kiện mạng (nơi thuật ngữ "sự kiện" được sử dụng rất rộng). Trên thực tế, ChannelHandler có thể được dành cho hầu hết mọi loại hành động, chẳng hạn như chuyển đổi dữ liệu từ một định dạng này sang dạng khác hoặc xử lý những ngoại lệ bị ném ra trong quá trình xử lý.

Ví dụ: ChannelInboundHandler là một subinterface sẽ triển khai thực hiện một cách thường xuyên. Loại này nhận các sự kiện và dữ liệu gửi đến để xử lý logic của ứng dụng. Mặt khác cũng có thể xóa dữ liệu từ một ChannelInboundHandler khi gửi phản hồi đến một client kết nối. Logic của ứng dụng sẽ thường nằm trong một hoặc nhiều ChannelInboundHandlers.

**3.2.2 Interface ChannelPipeline**

ChannelPipeline cung cấp vùng chứa cho một chuỗi các ChannelHandler và định nghĩa một API để truyền lưu lượng các sự kiện đi vào và đi ra dọc theo chuỗi. Khi một Channel được tạo, nó sẽ được tự động được gán cho ChannelPipeline riêng của mình.

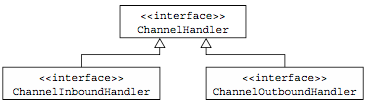
Các ChannelHandler được cài đặt trong ChannelPipeline như sau:

* Một thực thi *ChannelInitializer* được đăng ký với *ServerBootstrap*.
* Khi *ChannelInitializer*.*initChannel* () được gọi, *ChannelInitializer* cài đặt một tập các *ChannelHandler* tùy chỉnh trong pipeline.
* Trình *ChannelInitializer* sẽ tự xóa chính nó khỏi *ChannelPipeline*.

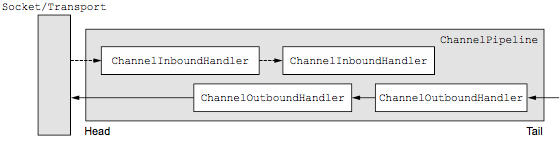
Phần tiếp theo sẽ trình bày sâu hơn vào mối quan hệ cộng sinh giữa *ChannelPipeline* và *ChannelHandler* để kiểm tra xem điều gì sẽ xảy ra với dữ liệu khi gửi hoặc nhận nó.

*ChannelHandler* đã được thiết kế đặc biệt để hỗ trợ một phạm vi sử dụng rộng rãi và có thể như một kho chứa chung cho bất kỳ mã nào xử lý các sự kiện (bao gồm cả dữ liệu) đến và đi qua *ChannelPipeline*. Điều này được minh họa trong hình x, cho thấy nguồn gốc của *ChannelInboundHandler* và *ChannelOutboundHandler* từ *ChannelHandler*.

Sự di chuyển của một sự kiện thông qua pipeline là công việc của *ChannelHandler,* đã được cài đặt trong quá trình khởi tạo hoặc giai đoạn khởi động của ứng dụng. Các đối tượng này nhận các sự kiện, thực thi xử lý logic mà chúng đã được thực hiện và truyền dữ liệu tới trình xử lý kế tiếp trong chuỗi. Thứ tự thực hiện được xác định theo thứ tự chúng được thêm vào.



Hình x: Phân cấp class ChannelHandler



Hình x: ChannelPipeline

Hình x cũng chỉ ra rằng cả hai bộ xử lý dữ liệu đi vào và đi ra có thể được cài đặt trong cùng một đường ống. Nếu tin nhắn hoặc bất kỳ sự kiện đi đến nào được đọc, nó sẽ bắt đầu từ đầu của đường ống và được chuyển đến *ChannelInboundHandler* đầu tiên. Bộ xử lý này có thể hoặc không thể sửa đổi dữ liệu, phụ thuộc vào chức năng cụ thể của nó, sau đó dữ liệu sẽ được chuyển tới *ChannelInboundHandler* tiếp theo trong chuỗi. Cuối cùng, các dữ liệu sẽ tới cuối của đường ống, tại điểm mà tất cả các xử lý được chấm dứt.

Sự di chuyển dữ liệu ra bên ngoài (tức là dữ liệu đang được ghi) là giống hệt nhau. Trong trường hợp này, luồng dữ liệu từ đuôi qua chuỗi các ChannelOutboundHandler cho đến khi nó đạt tới đầu. Ngoài điểm này, dữ liệu gửi đi sẽ đi đến Socket, từ đó sẽ được gửi tới client tương ứng.

Do hoạt động dữ liệu đến và đi là khác biệt, nên dù được trộn lẫn trong cùng một ChannelPipeline thì cũng không xảy ra xung đột. Thông qua hai thể hiện của *ChannelHandler* là *ChannelInboundHandler* và *ChannelOutboundHandler,* sẽ đảm bảo rằng dữ liệu được truyền chỉ giữa các bộ xử lý cùng loại.

Khi một *ChannelHandler* được thêm vào một *ChannelPipeline*, nó được gán một *ChannelHandlerContext*, đại diện cho sự ràng buộc giữa *ChannelHandler* và *ChannelPipeline*.Và phần lớn đối tượng này được sử dụng để ghi dữ liệu đi.

**3.2.3 Encoders và Decoders**

Khi gửi hoặc nhận tin nhắn với Netty, có xảy ra sự chuyển đổi dữ liệu. Một tin nhắn gửi đến sẽ được giải mã. Có nghĩa là, được chuyển đổi từ các byte sang một định dạng khác, thường là một đối tượng Java. Nếu tin nhắn được gửi đi, nó sẽ được mã hoá sang byte từ định dạng hiện tại của nó. Lý do cho cả hai chuyển đổi là dữ liệu network luôn luôn là một loạt byte.

Có nhiều abstract classe được cung cấp cho các bộ mã hóa và giải mã, tương ứng với các nhu cầu cụ thể. Tất cả các class mã hóa / giải mã cung cấp bởi Netty sẽ được thực thi tương ứng trên cả *ChannelInboundHandler* và *ChannelOutboundHandler*.

Ví dụ như *ByteToMessageDecoder* hoặc *MessageToByteEncoder*. Trong trường hợp đặc biệt, có thể sử dụng *ProtobufEncoder* và *ProtobufDecoder*, được cung cấp để hỗ trợ bộ đệm giao thức của Google.

**3.3 Transport trong Netty**

**3.3.1 NIO – non-blocking I/O**

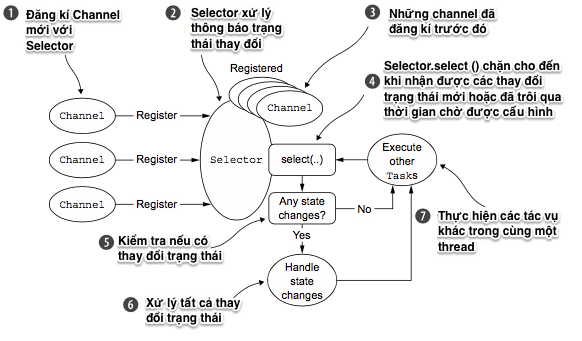
Khái niệm cơ bản phía sau *Selector* là đăng ký yêu cầu được thông báo khi tình trạng của *Channel* thay đổi. Những thay đổi của trạng thái có thể là

* Một *Channel* mới được chấp nhận và đã sẵn sàng.
* Một *Channel* đã hoàn tất kết nối.
* Một *Channel* có dữ liệu đã sẵn sàng để đọc.
* Một *Channel* sẵn sàng cho việc ghi dữ liệu.

Sau khi ứng dụng phản ứng với sự thay đổi trạng thái, selector được đặt lại và lặp lại quá trình, chạy trên một thread, kiểm tra sự thay đổi và đáp ứng chúng.

Các hằng số trong bảng 4.3 mô tả các mẫu bit được xác định bởi lớp *java*.*nio*.*channels*.*SelectionKey*. Các mẫu này được kết hợp để xác định tập hợp các thay đổi trạng thái mà ứng dụng yêu cầu thông báo.

|  |  |
| --- | --- |
| Tên | Mô tả |
| OP\_ACCEPT | Yêu cầu thông báo khi kết nối mới được chấp nhận và một *Channel* được tạo. |
| OP\_CONNECT | Yêu cầu thông báo khi kết nối được thiết lập. |
| OP\_READ | Yêu cầu thông báo khi dữ liệu đã sẵn sàng để được đọc từ *Channel*. |
| OP\_WRITE | Yêu cầu thông báo khi có thể ghi thêm dữ liệu vào *Channel*. Xử lý khi bộ đệm socket được hoàn thành đầy đủ, thường xảy ra khi dữ liệu được truyền nhanh hơn so với remote peer có thể xử lý. |



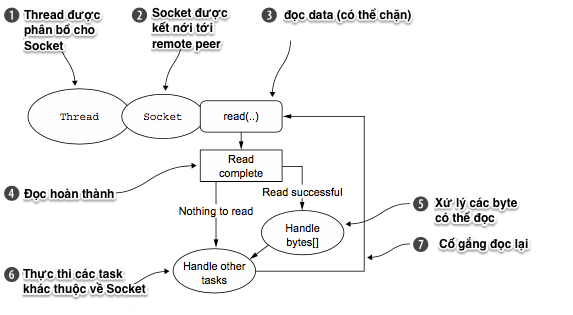
Hình x: Lựa chọn và xử lý thay đổi trạng thái

**3.3.2 OIO – old blocking I/O**

Sự thực thi OIO transport của Netty thể hiện một sự thỏa hiệp: nó được truy cập thông qua transport API thông thường, nhưng vì nó được xây dựng theo cơ chế ngăn chặn (blocking) của java.net nên nó không phải là bất đồng bộ (asynchronous). Tuy nhiên nó rất phù hợp với một số mục đích sử dụng nhất định.

Ví dụ, bạn có thể cần phải nhập mã thừa kế sử dụng các thư viện thực hiện cuộc gọi chặn (như JDBC) và có thể không thực tế để chuyển đổi logic thành không chặn. Thay vào đó, bạn có thể sử dụng vận chuyển OIO của Netty trong thời gian ngắn, và chuyển mã của bạn sau đó đến một trong những chuyến vận chuyển không đồng bộ thuần túy. Hãy xem nó hoạt động như thế nào.

Trong API java.net, bạn thường có một chủ đề chấp nhận các kết nối mới đến ServerSocket đang nghe. Một socket mới được tạo ra cho sự tương tác với peer, và một thread mới được phân bổ để xử lý lưu lượng truy cập. Điều này là cần thiết vì bất kỳ hoạt động I / O trên một ổ cắm cụ thể có thể chặn bất cứ lúc nào. Xử lý nhiều bao đựng bằng một sợi đơn có thể dễ dàng dẫn đến một hoạt động chặn trên một ổ cắm buộc lên tất cả những người khác là tốt.



Hình x: Logic xử lý OIO

Làm thế nào Netty có thể hỗ trợ NIO với cùng một API được sử dụng cho asynchronous transport - vận chuyển bất đồng bộ. Câu trả lời là Netty sử dụng cờ SO\_TIMEOUT Socket, trong đó xác định số mili giây tối đa để chờ cho một thao tác I / O hoàn thành. Nếu hoạt động không hoàn thành trong khoảng thời gian quy định, một *SocketTimeoutException* sẽ được ném ra. Netty bắt ngoại lệ này và tiếp tục vòng lặp xử lý. Trong lần chạy *EventLoop* tiếp theo, nó sẽ thử lại. Đây là cách duy nhất để một asynchronous framework như Netty có thể hỗ trợ OIO. Hình x minh hoạ logic này.

**CHƯƠNG 4: PHÂN TÍCH VÀ THIẾT KẾ HỆ THỐNG**

**3.1 Phân tích hệ thống**

**3.2 Thiết kế hệ thống**

**3.2.1 xx**

**3.2.2 Xây dựng biểu đồ lớp**

**3.2.2.1 Xây dựng biểu đồ lớp cho Server**

Server được xây dựng gồm có 4 thành phần chính được chia thành 4 packet:

* Game: Gói quy định các thành phần trong trò chơi.
* Server: Gói xử lý giao tiếp với Client thông qua các message, là khối trung tâm điều hành mọi hoạt động của hệ thống.
* Message: Gói định dạng các tin nhắn đến và đi.
* Common: Gói chứa thông tin cài đặt cho hệ thống và các thành phần sử dụng chung nhau.

**\* Thiết kế gói Server**

Gói Server bao gồm các lớp phục vụ cho việc quản lý các kết nối từ Client tới Server và quản lý các tin nhắn đi đến và đi ra.

Hình x: Biểu đồ lớp có trong gói Server.

|  |  |  |
| --- | --- | --- |
| STT | Tên lớp | Chức năng |
| 1 | CaroChessServer |  |
| 2 | WebSocketServerPipelineFactory |  |
| 3 | CaroChessServerHandler |  |

Bảng x: Bảng mô tả các lớp có trong gói Server

\* **Thiết kế gói Game**

Gói Game bao gồm các lớp định nghĩa các thành phần có trong trò chơi cờ Caro.

Hình x: Biểu đồ lớp có trong gói Game.

|  |  |  |
| --- | --- | --- |
| STT | Tên lớp | Chức năng |
| 1 | Game |  |
| 2 | GameBoard |  |
| 3 | Player |  |

Bàng x: Bảng mô tả các lớp có trong gói Game.

\* **Thiết kế gói Message**

Gói Message bao gồm các lớp định dạng các tin nhắn đến và đi.

Hình x: Biểu đồ lớp có trong gói Message.

|  |  |  |
| --- | --- | --- |
| STT | Tên lớp | Chức năng |
| 1 | MessageType |  |
| 2 | MessageBean |  |
| 3 | HandshakeMessageBean |  |
| 4 | IncomingCreateRoomMessageBean |  |
| 5 | IncomingJoinRoomMessageBean |  |
| 6 | IncomingTurnResponseMessageBean |  |
| 7 | IncomingExitGameMessageBean |  |
| 8 | OutgoingCreateRoomMessageBean |  |
| 9 | OutgoingRoomListMessageBean |  |
| 10 | OutgoingJoinRoomMessageBean |  |
| 11 | OutgoingQuickStartMessageBean |  |
| 12 | TurnMessageBean |  |
| 13 | OutgoingTurnResponseMessageBean |  |
| 14 | GameOverMessageBean |  |
|  |  |  |

\* **Thiết kế gói Common**

**3.2.2.2 Xây dựng biểu đồ lớp cho Client**

**CHƯƠNG 5: KẾT QUẢ ĐẠT ĐƯỢC**

**KẾT LUẬN**

# TÀI LIỆU THAM KHẢO

*Mẫu danh mục tài liệu tham khảo*

\* Bài báo đăng trên tạp chí khoa học : Tên tác giả, tên bài báo, tên tạp chí, volume, từ trang đến trang (nếu có), nhà xuất bản, năm xuất bản.

1. Hovy E, Automated Discourse Generation Using Discourse Structure Relations, Artificial Intelligence, Elsevier Science Publishers, 63: 341-385, 1993.

\* Sách : Tên tác giả, tên sách, volume (nếu có), lần tái bản (nếu có), nhà xuất bản, năm xuất bản.

1. Peterson L. L. and Davie B. S. , Computer Networks: A Systems Approach, 2nd ed., Mogran-Kaufmann, 1999.
2. Nguyễn Thúc Hải, Mạng máy tính và các hệ thống mở, Nhà xuất bản giáo dục, 1999.

\* Tập san Báo cáo Hội nghị Khoa học : Tên tác giả, tên báo cáo, tên hội nghị, ngày (nếu có), địa điểm hội nghị, năm xuất bản.

1. Poesio M. and Di Eugenio B., Discourse Structure and Anaphoric Accessibility, In Proc. of the ESSLLI Workshop on Information Structure, Discourse Structure and Discourse Semantics, Helsinki, 2001.

\* Đồ án tốt nghiệp, Luận văn Thạc sĩ, Tiến sĩ : Tên tác giả, tên đồ án/luận văn, loại đồ án/luận văn, tên trường, địa điểm, năm xuất bản.

1. Knott D., A Data-Driven Methodology for Motivating a Set of Coherence Relations, Ph.D. Thesis, University of Edinburgh, UK, 1996.

\* Tài liệu tham khảo từ Internet : Tên tác giả (nếu có), tựa đề, cơ quan (nếu có), địa chỉ trang web, thời gian lần cuối truy cập trang web.

1. Berners-Lee T., Hypertext Transfer Protocol (HTTP), CERN, <URL:ftp:/info.cern.ch/pub/www/doc/http-spec.txt.Z>, last visited May 2010.
2. Princeton University, WordNet, http://www.cogsci.princeton.edu/~wn/index.shtml, last visited May 2010.