ThriftUsageObjectiveC-new

Before using Thrift for Objective-C

Supported Platforms

Thrift supports the following platforms as well as other platforms.

- Mac OS X 10.5 or later
- iOS 4 or later

cocoa represents the above platforms. The runtime library and the generated code adapt themselves to the platforms automatically. Thus, please use *cocoa* for them.

```
% thrift --gen cocoa idl.thrift
```

Installation

See this page.

Support of ARC

Since iOS 5 and Mac OS 10.7, Objective-C supports Automatic Reference Counting (ARC) in addition to the conventional object life cycle management by retain/release (non-ARC).

Thrift 0.8.0 does not support ARC. If you want use it within an app in ARC mode, you need to compile the thrift runtime library in non-ARC mode by specifying -fno-objc-arc flag.

Use 0.9.0-dev or later, if you need to use thrift in ARC mode. It supports both ARC and non-ARC. Both the runtime library and generated Objective-C code can be compiled both in ARC mode and in non-ARC mode. They automatically adapt themselves to the compilation modes without any switch.

When you modify Thrift

Use *retain_stub* instead of *retain* and use *release_stub* instead of *release* inside the runtime library. The thrift compiler should generate *retain_stub/release_stub* instead of *retain/release*. See this for detail.

How to Link the Runtime Library

You need to link the runtime library of Thrift with your project. The location of the runtime library: *thrift-x.x.x/lib/cocoa/src*

There are two options.

- If your project is small, you can add the source files of the runtime library into your project.

- Otherwise, if you use Thrift within several sub projects, you must make a framework from the source files of the runtime library. Since ways of making a framework is a bit complicated and exceeds this guide, please consult other sites. We expect somebody will contribute a script that makes a framework automatically.

The rest of this guide assumes we directly add the runtime library into our sample project and we don't make the framework.

Getting started

This tutorial will walk you through creating a sample project which is a bulletin board application using Thrift. It consists of an iOS client app written in Objective-C and a Java server. It is assumed that you will have a basic knowledge of Objective-C and Xcode to complete this tutorial. Note that the client runs on Mac OS with small modification.

`http://staff.aist.go.jp/hirano-s/thrift/result.tiff|align=right`

Acknowledgment: A part of this tutorial was borrowed from *newacct's* tutorial.

Sample Download

You can download the sample project from here. It includes:

- The whole myThriftApp project
- the runtime library in thrift/ directory from 0.8.0-dev (You should replace this with the latest one.)
- idl.thrift
- Server.java and BulletinBoard.java in gen-java/ directory

Requirements

Make sure that your system meets the requirements as noted in ThriftRequirements.

- Thrift 0.9.0+ (0.8.0-dev+)
- iOS 4+ or Mac OS X 10.5+
- Xcode 4.2+

The following are required for the Java server; but, not the Objective-C client.

- Java 1.4+ (already installed in Mac OS X 10.5+)
- SLF4J (available at http://www.slf4j.org/download.html)

Creating the Thrift file

We will use a simple thrift IDL file, *myThriftApp/idl.thrift*. This defines a bulletin board service. You can upload your message and date using *add()* method. *get()* method returns a list of the struct so that we demonstrate how to send/receive an array of structs in Objective-C.

```
// idl.thrift
struct Message {
     1: string text,
     2: string date
}

service BulletinBoard {
    void add(1: Message msg),
    list<Message> get()
```

}

- Run the thrift compiler to generate the stub files (e.g. gen-cocoa/idl.h and gen-cocoa/idl.m).

```
thrift --gen java idl.thrift
thrift --gen cocoa idl.thrift
```

Create the Objective-C client app

The objective-c client is a simple app that allows the user to fill out a text field add/get them from the server.

- Create a new iOS single view project.
  - Product name is *myThriftApp*.
  - Check to *Use Automatic Reference Counting*

- Add generated files.
  - Right click on myThriftApp and select *Add files to "myThirtApp"*. Choose "gen-cocoa".`http://staff.aist.go.jp/hirano-s/thrift/add.tiff|align=right`
- Add the runtime library.
  - Right click on myThriftApp and select *Add files to "myThirtApp"*. Choose "thrift-x.x.x/lib/cocoa/src".
  - rename group name from *src* to *thrift*.`http://staff.aist.go.jp/hirano-s/thrift/group.tiff|align=right`
- Setup header search path in build settings.
  - Always Search User Path: YES `http://staff.aist.go.jp/hirano-s/thrift/path2.tiff|align=right`
  - Framework Search Paths: add *$(SRCROOT)* and *$(inherited)* `http://staff.aist.go.jp/hirano-s/thrift/path.tiff|align=right`

- Copy the following text to ViewController.h

```
#import <UIKit/UIKit.h>

@class BulletinBoardClient;

@interface ViewController : UIViewController <UITextFieldDelegate> {
    BulletinBoardClient *server;
}
@property (strong, nonatomic) IBOutlet UITextField *textField;
@property (strong, nonatomic) IBOutlet UITextView *textView;
- (IBAction)addPressed:(id)sender;
@end
```

- Open the ViewController_iPhone.xib
  - Place a Text Field, a Round Rect Button, and a Text View. `http://staff.aist.go.jp/hirano-s/thrift/xib.png|align=right`
  - Connect the delegate of the Text Field to File's Owner.
  - Connect the Text Field to *textField* variable in ViewController.h.
  - Connect the Button to *addPressed:* method in ViewController.h. Give *add* title to the button.
  - Connect the Text View to *textView* variable in ViewController.h. Clear the content of the Text View.
- Copy the following code into ViewController.m

```
#import <TSocketClient.h>
#import <TBinaryProtocol.h>
#import "ViewController.h"
#import "idl.h"

@implementation ViewController
@synthesize textField;
@synthesize textView;

- (void)viewDidLoad {
    [super viewDidLoad];

    // Talk to a server via socket, using a binary protocol
    TSocketClient *transport = [[TSocketClient alloc] initWithHostname:@"localhost" port:7911];
    TBinaryProtocol *protocol = [[TBinaryProtocol alloc] initWithTransport:transport strictRead:YES strictWrite:
YES];
    server = [[BulletinBoardClient alloc] initWithProtocol:protocol];
}

- (void)viewDidUnload {
    [self setTextField:nil];
    [self setTextView:nil];
    [super viewDidUnload];
}

- (IBAction)addPressed:(id)sender {
    Message *msg = [[Message alloc] init];
    msg.text = textField.text;
    msg.date = [[NSDate date] description];

    [server add:msg];       // send data

    NSArray *array = [server get];    // receive data
    NSMutableString *s = [NSMutableString stringWithCapacity:1000];
    for (Message *m in array)
        [s appendFormat:@"%@ %@\n", m.date, m.text];
    textView.text = s;
}

- (BOOL)textFieldShouldReturn:(UITextField*)aTextField {
    [aTextField resignFirstResponder];
    return YES;
}
@end
```

- This code creates a new transport object that connects to localhost:7911, it then creates a protocol using the strict read and strict write settings. These are important as the java server has strictRead off and strictWrite On by default. In the iOS app they are both off by default. If you omit these parameters the two objects will not be able to communicate.

Creating the Java Server

- cd into the gen-java directory
- make sure that your classpath is properly setup. You will need to ensure that ".", libthrift.jar, slf4j-api, and slf4j-simple are in your classpath.
- create the file BulletinBoardImpl.java

```
import org.apache.thrift.TException;
import java.util.List;
import java.util.ArrayList;

class BulletinBoardImpl implements BulletinBoard.Iface {
    private List<Message> msgs;

    public BulletinBoardImpl() {
        msgs = new ArrayList<Message>();
    }

    @Override
    public void add(Message msg) throws TException {
        System.out.println("date: " + msg.date);
        System.out.println("text: " + msg.text);
        msgs.add(msg);
    }

    @Override
    public List<Message> get() throws TException {
        return msgs;
    }
}
```

- create a file called 'Server.java'

```
import java.io.IOException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TBinaryProtocol.Factory;
import org.apache.thrift.server.TServer;
import org.apache.thrift.server.TServer.Args;
import org.apache.thrift.server.TSimpleServer;
import org.apache.thrift.transport.TServerTransport;
import org.apache.thrift.transport.TServerSocket;
import org.apache.thrift.transport.TTransportException;

public class Server {

    private void start() {
        try {
            BulletinBoard.Processor processor = new BulletinBoard.Processor(new BulletinBoardImpl());
            TServerTransport serverTransport = new TServerSocket(7911);
            TServer server = new TSimpleServer(new Args(serverTransport).processor(processor));

            System.out.println("Starting server on port 7911 ...");
            server.serve();
        } catch (TTransportException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        Server srv = new Server();
        srv.start();
    }
}
```

- compile the classes

```
javac *.java
```

- This will run a server that implements the service BulletinBoard on port 7911. The service simply stores Message structures in a List and returns the list when requested.

Running

- run the server

```
java Server
```

- Build and run the myThriftApp **in the iPhone simulator**. Since it tries to connect to localhost, you need to run it in the iPhone simulator rather than an iPhone.
    - Put a text in the text field and push add button.

`http://staff.aist.go.jp/hirano-s/thrift/result.tiff|align=right`

Objective-C specific Notes

Method signature

When a service with multiple arguments is compiled, argument names are omitted.

```
// IDL
service TestService {
   void method(1: i32 value1, 2: i32 value2),
}
```

```
void method:(int)value1 :(int)value2;
//   [server method:24  :33];
```

Attributes

Struct members can be accessed via attributes of an instance.

```
// IDL
struct Person {
  1: string name,
  2: string city,
}
```

```
   Person *p = [[Person alloc] init];
   p.name = @"HIRANO";
   p.city = @"Tsukuba";
```

Initialization

Since 0.9.0 (0.9.0-dev), initialization of struct and const is supported.

```
// IDL
struct Person {
  1: string name = "your name",
  2: string city = "your city",
}

const list<Person> MyFamily = [{name: "Satoshi", city: "Tsukuba"}, {name: "Akiko", city: "Tokyo"}]
```

String type

string is converted into UTF-8 encoding before sending. The UTF-8 encoding is converted to NSString* after receiving.

Note all language versions convert encoding automatically. For example Python does not do encoding conversion. (Thus, you need to do conversion by yourself).

Binary type

Thrift for Objective-C supports *binary* type as NSData*.

```
// IDL
struct Person {
  1: string name,
  2: binary photo,

service TestService {
   void register(1: Person person),
}
```

```
   Person *p = [[Person alloc] init];
   p.name = @"HIRANO Satoshi";
   NSImage *image = [NSImage imageNamed:@"hirano.png"];   // load an image
   NSData *tiffData = [image TIFFRepresentation];   // obtain data
   p.photo = tiffData;
   [server register:p];
```

Collection types

The objective-C version supports collection types, list, set and map as NSArray, NSSet, and NSDictionary.

```
// IDL
struct TestData {
   list<string> listData,
   set<string> setData,
   map<string, string> mapData,
}
```

```
  TestData *t = [[TestData alloc] init];
  t.listData = [NSArray arrayWithObjects:@"a", @"b", nil];
  t.setData = [NSSet setWithObjects::@"e", @"f", nil];
  t.mapData = [NSDictionary dictionaryWithObjectsAndKeys:@"name", @"HIRANO", @"city", @"Tsukuba", nil];
```

Client Side Transports

You can mainly use the socket transport and the HTTP transport.

Here is a client side example for the socket transport.

```
#import <TSocketClient.h>
#import <TBinaryProtocol.h>

- (void) connect {
    // Talk to a server via socket, using a binary protocol
    TSocketClient *transport = [[TSocketClient alloc] initWithHostname:@"localhost" port:7911];
    TBinaryProtocol *protocol = [[TBinaryProtocol alloc] initWithTransport:transport strictRead:YES strictWrite:
YES];
    server = [[BulletinBoardClient alloc] initWithProtocol:protocol];
```

Here is a client side example for the HTTP transport. You may use https. You can connect to Google App Engine for example.

```
#import <THTTPClient.h>
#import <TBinaryProtocol.h>

- (void) connect {
    NSURL *url = [NSURL URLWithString:@"http://localhost:8082"];
    // url = [NSURL URLWithString:@"https://myapp145454.appspot.com"];

    // Talk to a server via HTTP, using a binary protocol
    THTTPClient *transport = [[THTTPClient alloc] initWithURL:url];
    TBinaryProtocol *protocol = [[TBinaryProtocol alloc]
                                 initWithTransport:transport
                                 strictRead:YES
                                 strictWrite:YES];

    server = [[TestServiceClient alloc] initWithProtocol:protocol];
```

Asynchronous Client

The asynchronous operation means that an RPC call returns immediately without waiting for the completion of a server operation. It is different from the oneway operation. An asynchronous operation continues in background to wait for completion and it is possible to receive a return value. This feature plays very important role in GUI based apps. You don't want to block for long time when a user pushes a button.

Unlike Java version, Objective-C version does not support asynchronous operation.

However, it is possible to write asynchronous operations using *NSOperationQueue*. The basic usage of *NSOperationQueue* is like this. Your async block is executed in a background thread.

```
[asyncQueue addOperationWithBlock:^(void) {
    // your async block is here.
    int val = [remoteServer operation];
}];
```

There are some points to be considered.

- Your async blocks are done in asyncQueue, an instance of *NSOperationQueue*.
- Strong objects may not be accessed from a block. Since *self* is also a strong object, we need to avoid to access *self* within an async block. We use weakSelf, a weak reference to *self*.
- GUI operations must be in the main thread. For example, *uilabel.text = @"foo"*; must be done in the main thread and you may not write it in the above async block. A block that handles GUI is added to the *mainQueue* which represents the main thread. We use *weaksSelf2* in the nested async block.
- Exception handling is also needed.

Here are the fragments of ThriftTestViewController class.

```
// IDL
service TestService {
   i32 sum(1: i32 value1, 2: i32 value2),
}
```

```
@interface ThriftTestViewController : UIViewController {
    IBOutlet UILabel *msg;
}
```

```
@property (nonatomic, retain) UILabel *msg;
@property (nonatomic, strong) TestServiceClient *server;
@property (nonatomic, strong) NSOperationQueue *asyncQueue;
@property (nonatomic, strong) NSOperationQueue *mainQueue;
```

```
- (void )viewDidLoad {
    asyncQueue = [[NSOperationQueue alloc] init];
    [asyncQueue setMaxConcurrentOperationCount:1]; // serial
    mainQueue = [NSOperationQueue mainQueue];   // for GUI, DB

    NSURL *url = [NSURL URLWithString:@"http://localhost:8082"];

    // Talk to a server via HTTP, using a binary protocol
    THTTPClient *transport = [[THTTPClient alloc] initWithURL:url];
    TBinaryProtocol *protocol = [[TBinaryProtocol alloc]
                                 initWithTransport:transport
                                 strictRead:YES
                                 strictWrite:YES];
    // Use the service defined in profile.thrift
    server = [[TestServiceClient alloc] initWithProtocol:protocol];
    NSLog(@"Client init done %@", url);
}


-(void)doCalc {
    __unsafe_unretained ThriftTestViewController *weakSelf = self;
    [asyncQueue addOperationWithBlock:^(void) {
        __unsafe_unretained ThriftTestViewController *weakSelf2 = weakSelf;
        @try {
            weakSelf.msg.text = nil;
            int result = [weakSelf.server calc:24 :32];
            [weakSelf.mainQueue addOperationWithBlock:^(void) {
                weakSelf2.msg.text = [NSString stringWithFormat:@"%d", result];
            }];
        }
        @catch (TException *e) {
            NSString *errorMsg = e.description;
            NSLog(@"Error %@", errorMsg);
            [weakSelf.mainQueue addOperationWithBlock:^(void) {
                weakSelf2.msg.text = errorMsg;
            }];
        }
    }];
}
```