

Internship Project Report 2015

Implementation of Client-Server architecture
between an RCP Text Editor and LLVM
Based Clang static analyzer

**SAMSUNG R&D INSTITUTE INDIA,
BANGALORE**

July 13, 2015
Authored by: Abhijeet Gaur

ABSTRACT

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them. Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver fast compiles, useful error and warning messages and to provide a platform for building great source level tools [1]. The Clang Static Analyzer is a tool that automatically detects bugs in the written code, and can be conceptualized using the Clang frontend as a library to parse C/C++ code [2]. Clang offers a genuinely better alternative than GCC while compiling and is about three times faster than GCC when compiling Objective-C code in a debug configuration. The project attempts to modify the existing Tizen SDK text-editing framework, provided with the Integrated Development Environment (IDE) of Tizen Software Development Kit (SDK) so as to utilize the enhanced functionality provided by the Open Source LLVM-based toolchain. The current IDE of Tizen SDK utilizes the pre-existing CDT library for the analysis of C/C++ languages for the development of RCP applications. This project aimed to remove the dependency of our Text-Editor on the Java based library and suggest a light-weight model for application development using the Tizen SDK.

ACKNOWLEDGEMENTS

I would like to take the opportunity to thank and express my deep sense of gratitude to my manager Mr. Senthil Kumar Thangavelu. I am greatly indebted to him for providing his valuable insights and constructive criticism at all stages of project and serving as a source of constant inspiration and guidance.

I would also like to thank my mentor Mr. Alok Mishra, for laying down the foundation and providing me the direction for the project. His constant encouragement and supportive attitude during the project were vital for its successful completion.

I would also like to display my gratitude towards Mr. Aditya Aswani, Mr. Karthik Venkatesh Bhat and Mr. Dinesh Dwivedi, who were kind enough to spare their precious time in resolving my apprehensions and doubts which were encountered during the course of the project.

I am also thankful to Mr. Harith C and Ms. Sanjana Sipani and other members of the campus recruitment team, who found me worthy of this internship and ensured that I had a great experience at SRI-B.

Last but not the least, I owe my wholehearted thanks and appreciation to my friends and fellow interns and the entire staff of SRI-B for their cooperation and assistance during the eight weeks of internship.

I hope I can build upon the experience and knowledge that I have gained and make a valuable contribution towards the industry in the coming future.

CONTENTS

S.No.	Topic	Pg. No.
1.	Introduction	4
2.	Background and Project Overview	5
3.	Project Architecture	6
4.	Server Side <ul style="list-style-type: none">• Clang Tokenizer• Server	7
5.	Google Protocol Buffer	8
6.	Client Side <ul style="list-style-type: none">• RCP Text-editor• Client	9
7.	Testing	10
8.	Improvements and Enhancements	
9.	Conclusion	12
10.	Bibliography	13

INTRODUCTION

While the Tizen development platform is designed to serve as an open tools platform, it is architected so that its components could be used to build just about any client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform. Applications other than IDEs can be built using a subset of the platform. These rich applications are still based on a dynamic plug-in model, and the UI is built using the same toolkits and extension points. The layout and function of the workbench is under fine-grained control of the plug-in developer in this case. When we say that the Rich Client Platform is the minimal set of plug-ins needed to build a platform application with a UI, we mean that your application need only require two plug-ins, *org.eclipse.ui* and *org.eclipse.core.runtime*, and their prerequisites [3].

A text-editor is vital tool of the Integrated Development Environment in a Software Development Kit. It has been observed and cited by various seasoned developers that a good text-editor can greatly enhance the programming experience. The various color codes while typing according to the Semantic and Syntactic context of the program, ensure a better navigation and analysis. The main aim of this project was to increase my understanding of the architecture of a typical Rich Client Application in Tizen SDK and utilize the functionality of a C++ library in a text-editor plug-in in Syntax Highlighting.

Clang, a front-end tool of the LLVM toolchain, is a very powerful entity when it comes to the creation of Abstract Syntax Tree (AST), which is crucial for the syntax and semantic highlighting of the program. The process of Syntax Highlighting requires static analysis of the code which results in an intermediate code-generation [4].

BACKGROUND AND PROJECT OVERVIEW

Text-editor, as a Tizen Text Editor application can be fundamentally divided into four parts:

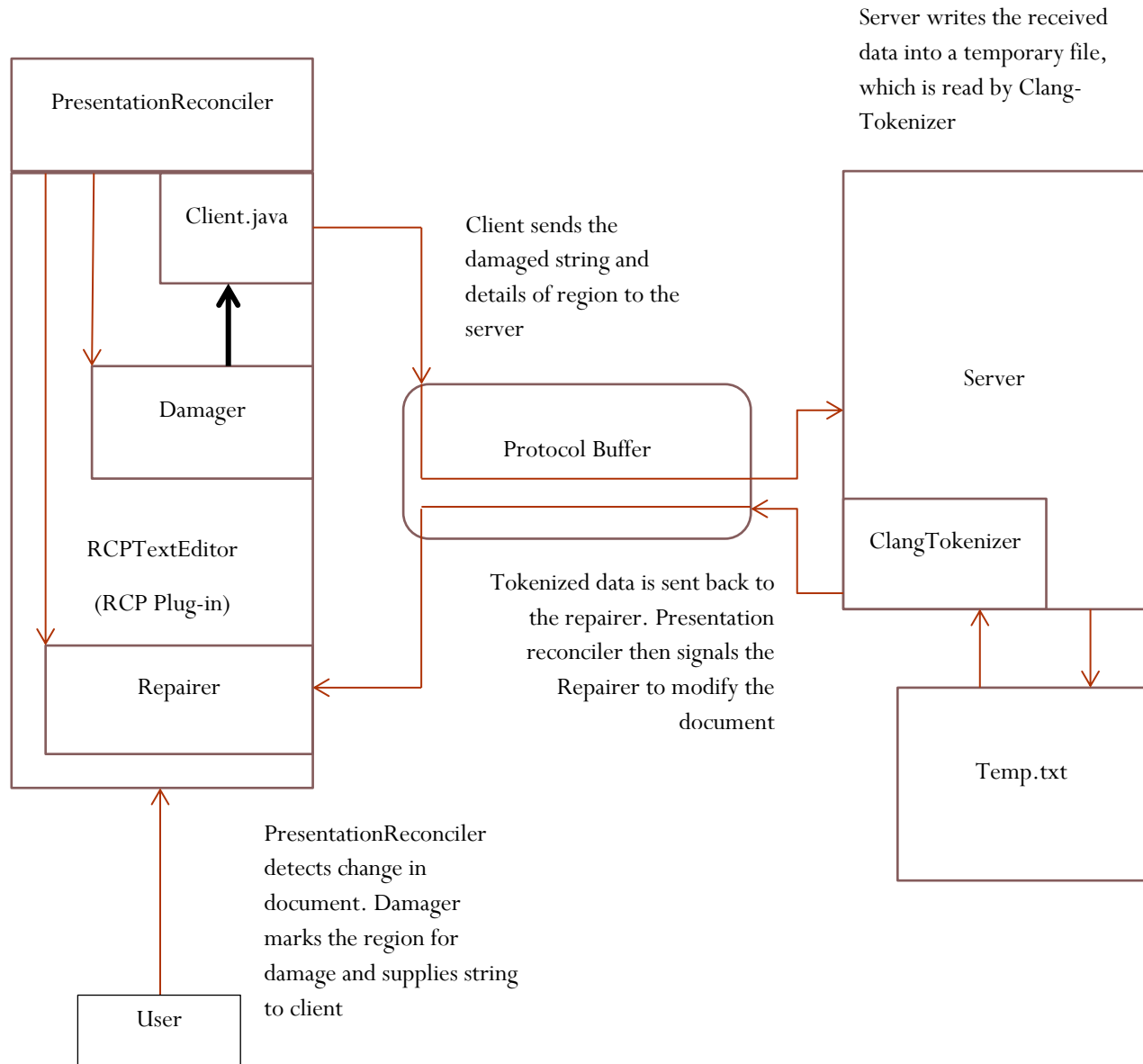
- Core
- Views
- Source
- Commands

The libraries of SWT and jFace, which are inbuilt in the current Tizen SDK Framework, provide the commands and the tools for building the aforementioned 4 parts and implementing a functional text highlighter. 'libClang' is the standard library of the Clang Tool that consists of tokenizing and AST creation tools of C, C++, Objective-C and other C-family languages. Since, Clang utilizes LLVM for compilation and ultimately GCC for execution as a C++ library, using Clang directly in Tizen SDK is not possible. This is due to the dependence of Eclipse-based editor on JVM machine for the compilation and execution of java-code. The aim of the project was to establish a link between the Tizen SDK Text-editor application and the C++ library using a novel approach of Client-Server architecture.

This approach is not completely unheard of as today some very powerful tools exist in the form of Open Source projects, for instance the *YouCompleteMe* text analysis tool for Vim. The major advantage of using a Client Server approach is that it reduces the burden and a substantial amount of weight of the RCP application. Apart from that, it further improves the possibility of scaling the application in form of web-emulator or cross platform text-analysis [5]. Due the endless opportunities provided by this particular application, as a part of this project, I,

- built a small prototype of an Tizen SDK-based Text-editor
- programmed my own Text-editor with the help of Google Protocol Buffer and simultaneously built a Server that could process the requirements of the Text-editor
- in the end stage of the project, integrated both the client side application and the server using sockets and devised a protocol to channelize the flow of data

PROJECT ARCHITECTURE



SERVER SIDE

Clang-Tokenizer

The Clang Tokenizer receives data through the Server. Stored in the class *ClangTokenReader.cpp*, the source code works on the received string of characters and creates an AST. The object ClangToken has the following attributes that get initialized by the Google's Protocol Buffer as soon as the Server receives the string and Tokenizer finishes its working on it:

Object ClangToken:	String TokenName	[Name of the Token]
	String TokenType	[Type of the token: Keyword, Literal, Identifier, Comment, Punctuation]
	Int TokenOffset	[Location of Token in the String]
	Int TokenLength	[Length of the detected token]

Server

The *server.cc* is the file which implements the standard *socket()* server over a desired port number. The server receives the string, the offset of the damaged string in the text document on the client side. The string is later written into a temporary file (*temp.txt*) The server utilizes the functions provided by *token.pb.cc* to use the services of Google's Protocol buffer library which provides the desired data structure to send the data back.

Interestingly, since the offset detected by the Clang Tokenizer, as mentioned before, corresponds to the location of the token in the given string. Since, our requirement concerns the location of token in the Document, we add the value of received offset to the value defined by TokenOffset and write the added result back to the client.

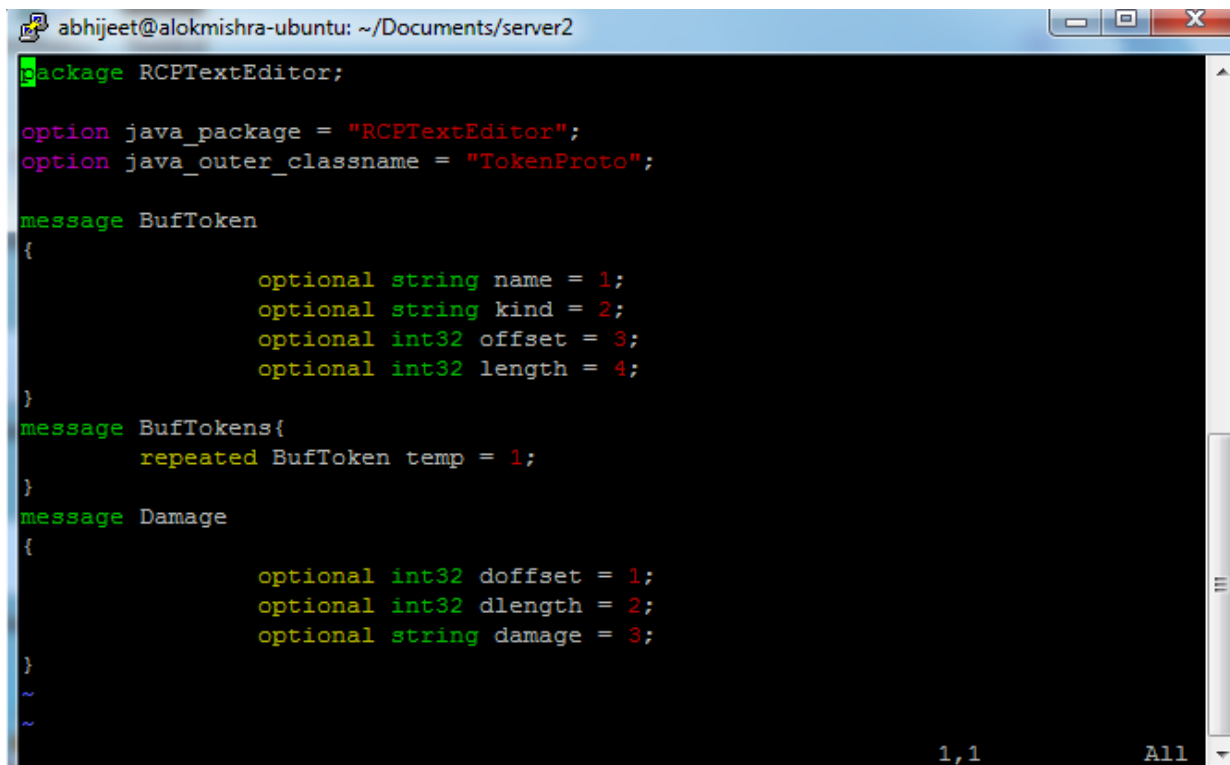
Execution in Project

In this project, the server runs on the Linux machine at the desired port number. All the constituting files: *ClangTokenReader.h*, *ClangTokenReader.cpp*, *token.pb.cc*, *token.pb.h*, *server.cc* are compiled using a simple *Makefile*. Please see the Appendix for the details of the *Makefile*. The final result is compiled into an executable *TServer*.

GOOGLE PROTOCOL BUFFER

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. One can simply define the data to be structured once. The generated source code can be used to easily write and read the structured data to and from a variety of data streams and using a variety of languages [6].

In this project, after installing the ProtocolBuffer framework and changing the path to include the protocol buffer code generator, a *token.proto* file was defined to declare the type of data structure that would be later used to transfer data to and from the server.

A screenshot of a terminal window with a dark background and light-colored text. The window title bar shows the user 'abhijeet@alokmishra-ubuntu' and the path '~/Documents/server2'. The code is a Protocol Buffer definition. It starts with a package declaration 'package RCPTextEditor;', followed by two options: 'option java_package = "RCPTextEditor";' and 'option java_outer_classname = "TokenProto";'. Then, it defines three messages: 'BufToken' with four optional fields (string name, string kind, int32 offset, int32 length), 'BufTokens' with a repeated field of 'BufToken' type, and 'Damage' with three optional fields (int32 doffset, int32 dlength, string damage). The code is color-coded: keywords in green, strings in red, and values in blue. The terminal shows line numbers '1,1' and 'All' at the bottom right.

```
package RCPTextEditor;

option java_package = "RCPTextEditor";
option java_outer_classname = "TokenProto";

message BufToken
{
    optional string name = 1;
    optional string kind = 2;
    optional int32 offset = 3;
    optional int32 length = 4;
}
message BufTokens{
    repeated BufToken temp = 1;
}
message Damage
{
    optional int32 doffset = 1;
    optional int32 dlength = 2;
    optional string damage = 3;
}
~
~
```

The *BufToken* object was used to generate a list of deliverable tokens defined by *BufTokens* on the server-side. The instances are extracted in class *Client.java* on the client side, after the generation of *TokenProto.java* file using the same command as was used to automatically create *token.pb.cc* and *token.pb.h* files.

These files defined the methods that were later used to manipulate the written and serialized data into the buffer.

CLIENT SIDE

Text-Editor

So as to gain a complete understanding of the underlying architecture of Tizen SDK Text-editor, the existing standard XMLEditor plugin, supplied as an extension, was extended to an independent C/C++ editor. As already mentioned, the working of standard editors can be divided into several basic parts. The peculiar thing about using jFace and SWT for development of a text-editor is that the resulting application provides a wide range of functionality over the applications created using Swing [7].

The current *RCPTextEditor* plug-in has the fundamental *Editor* class, which extends *TextEditor*. This sets an empty document (class *IDocument*) to the editor on initialization and declares a *SourceViewer* for the same. This class further calls upon the *Configuration* class that handles the *damager* and *repairer* of the given document.

Damager and Repairer

The notion of editing a document for highlighting is handled by making the changes to variables of *Document* class by Tizen SDK Text Editor. A two-way scan of some parts of the document helps in optimizing the entire process of text-formatting in real-time. The first scan finds out the region (class *IRegion*) that has been modified. This is called damaged region. The repairer is the program that has the information about modifying the damaged region. It gets triggered by a reconciler (class *PresentationReconciler*) which also sets itself for further scrutiny as soon as a change is made (class *DocumentEvent*) [8].

Client.java: Implementation in Project

The present working of the Damager-Repairer pair was modified in the following manner:

- 1) In this project, the damager uses the primary scan of just one type that marks the region for single and multi-line comments and default string. The secondary scan over the damaged region is carried out into the Server-side. The Client writes the damaged string into the protocol buffer, along with *IRegion* which holds the offset and length information
- 2) The Client then receives the tokenized information and supplies it to the repairer for Syntax Highlighting. A simple switch-case statement handles the attribute change.

TESTING

The final prototype worked in the following manner:

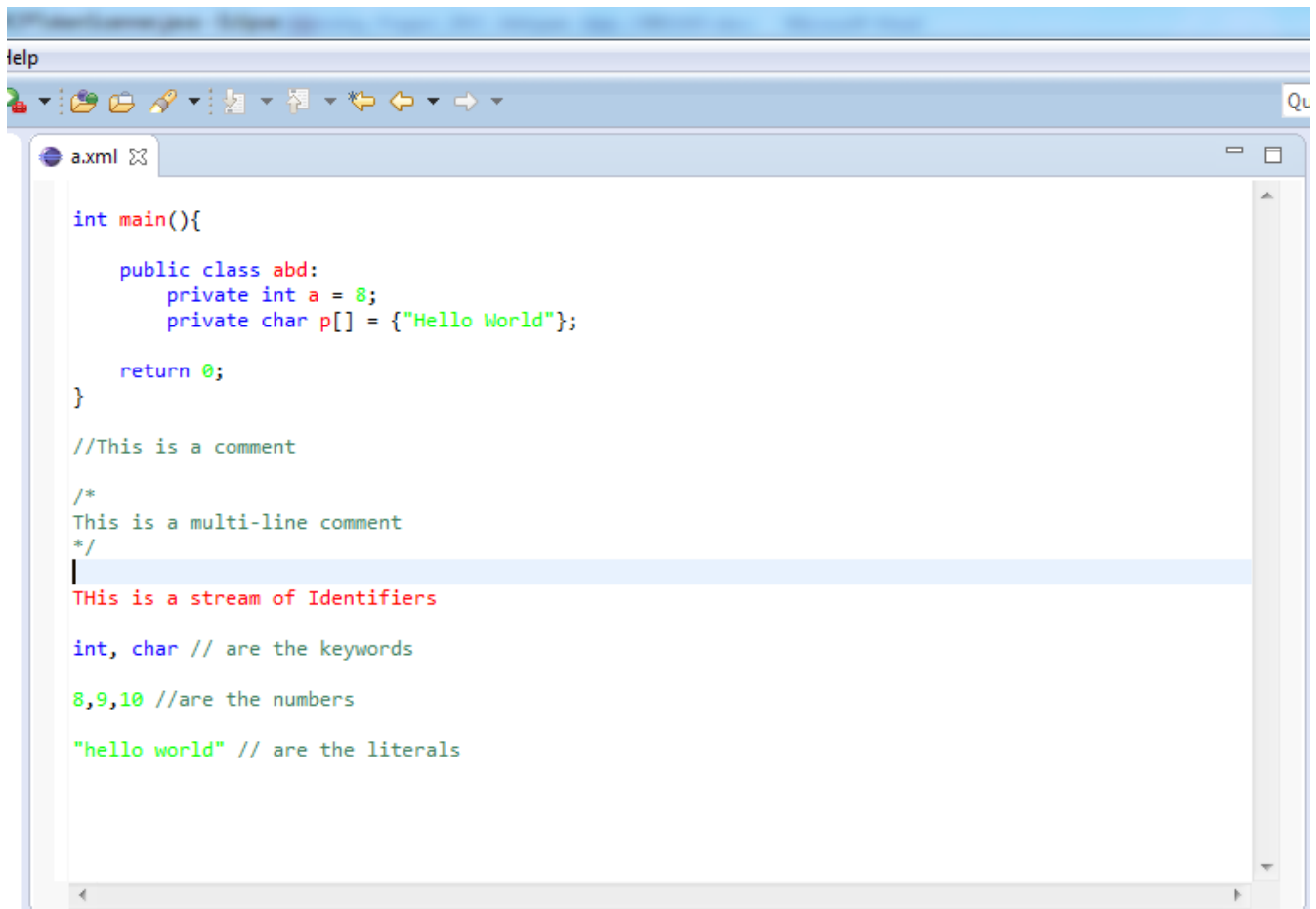
- 1) The Makefile on the server side generated the Tserver executable file. On execution, the server along with the functionality of Clang Tokenizer entered into the listening mode.

```
build clang686.zip clang+llvm-3.6.0-x86_64-linux-gnu.
clang686 clang+llvm-3.6.0-x86_64-linux-gnu llvm
abhijeet@alokmishra-ubuntu:~/Documents$ vi server2
abhijeet@alokmishra-ubuntu:~/Documents$ cd server2
abhijeet@alokmishra-ubuntu:~/Documents/server2$ ls
ClangTokenReader.cpp ClangTokenReader.h Makefile Makefile.txt server.cc tem
abhijeet@alokmishra-ubuntu:~/Documents/server2$ export LD_LIBRARY_PATH=/home/abh
abhijeet@alokmishra-ubuntu:~/Documents/server2$ vi server.cc
abhijeet@alokmishra-ubuntu:~/Documents/server2$ vi server.cc
abhijeet@alokmishra-ubuntu:~/Documents/server2$ make
cc -g -Wall -pthread -I/usr/local/include -I/home/abhijeet/Documents/clang686/i
ken.pb.cc -L/usr/local/lib -L/home/abhijeet/Documents/clang686/lib/ -lprotobuf
server.cc: In function 'int main(int, char**)':
server.cc:108:13: warning: unused variable 'd_length' [-Wunused-variable]
    int d_length = d.dlength();
        ^
Simple server named Tserver has been compiled
abhijeet@alokmishra-ubuntu:~/Documents/server2$ ./Tserver
ClangServer is up an running. Proceed ahead with your work, at port 8003
```

- 2) The client-side application was run as a plug-in to the current Tizen SDK editor and the receiving and sending of the modified tokens were mapped.

```
client created. Sending Damage
Tokens received. Printing
#
Punctuation
204
1
include
Identifier
205
7
//comment
Comment
213
9
```

3) Ultimately, the syntax highlighting worked in the sample RCP Text-Editor:



The screenshot shows a window titled 'a.xml' with a standard toolbar and a 'help' button. The code editor contains the following C++ code with syntax highlighting: keywords in blue, identifiers in black, literals in green, and comments in grey. The code is as follows:

```
int main(){  
    public class abd:  
        private int a = 8;  
        private char p[] = {"Hello World"};  
  
    return 0;  
}  
  
//This is a comment  
  
/*  
This is a multi-line comment  
*/  
|  
This is a stream of Identifiers  
  
int, char // are the keywords  
  
8,9,10 //are the numbers  
  
"hello world" // are the literals
```

IMPROVEMENTS AND ENHANCEMENTS

The existing project can be improved in the following areas:

First-line Highlighting

The first line input of the text-editor remains unaffected during syntax highlighting, irrespective of whatever text is written. However, syntax highlighting works as soon as the '#' symbol is used in front after a whitespace gap of one tab or one character.

After some debugging, it was found that this error exists due to an initialization fault in the Google Protocol Buffer. The protocol buffer fails to transmit the correctly defined damage region at offset 0. On further testing, it can be definitely concluded that the Protocol Buffer needs to be manipulated in such a way that the damaged string at offset zero is read as a normal string. Modifying the sent value of offset and de-modifying it on the server's end produces an *InvalidProtocolBufferAction* error.

Pre-processor Highlighting

A typical text-editor in Tizen SDK works on 2-scan process where the first scan is used to determine the damaged region and the second scan is done over the damaged region to tokenize the present entities in it. In this process, various sections of the code can be detected by defining their separate scanners. In our case, since our secondary scan takes place on the server side hence it becomes difficult to scan pre-processor declarations of #include, #define, #ifndef etc. as the clang-tokenizer separately recognizes the '#' and the following string.

This can be handled by declaring a separate PreProcessorScanner class that would extend RCPTokenScanner class. After the tokens for the damaged region are received, the streams of tokens need to be checked for the detection of all possible patterns constituted by the following:

```
private static final String HASH = "#";
private static final String LEFT_ANGLE = "<";
private static final String RIGHT_ANGLE = ">";
private static final String DOUBLE_QUOTE = "\"";
private static final String SINGLE_QUOTE = "'";
```

A simple default scanner can serve as the primary scanner for PreProcessorScanner but a better alternative would be to extend a RuleBasedScanner for a SingleLineRule with start sequence symbol as '#' for detection.

IMPROVEMENTS AND ENHANCEMENTS

The current project can be enhanced by adding the following features:

Semantic Highlighting

The basics of Semantic highlighting are dependent upon the recognition of functions, enumerations, classes and other API commands in the source code. Semantic Highlighting is one of the most crucial aspects of modern day text-editors. Implementing semantic analysis over the code requires the production of a file-wide Abstract Syntax Tree (AST). Currently, in this project, there is no such provision, since the protocol works in the form of a cycle triggered by the user's input in the editor. But, the protocol buffer is designed in such a way to accommodate multiple messages from the same application. Using this, we can dump the entire code of the source document, by running a parallel thread on the server on a separate port. This dumped code can then be acted upon by the ClangSemanticAnalyzer to generate the AST^[11].

Clang provides sufficient methods to traverse an automatically generated AST:

- The *IdentifierTable* and the *SourceManager* objects store the information around AST. The entry point to the AST can be called by `getTranslationUnitDecl()`, which would return a pointer to the highest Translation Unit in the AST dump.
- Whole generated AST is divided into core classes of variable declarations, compound statements, pointer types and quality types of variables.
- The internal working of Clang is interesting to understand but irrelevant to our discussion, since Clang offers a flexible API to extract the needed information^[12].
- The creation of ClangSemanticTokens can be done by traversing the AST with CXCursor object, and initializing an empty Token object with the location [offset and range], Token-type [function or compound statement or API call etc.] and name of the created token.

CONCLUSION

The approach of using the Client-Server architecture for text-attribute modification is an unorthodox. It offers a significant increase in functionality over the existing methods that are used to incorporate the benefits of LLVM into software development applications.

One of the most common methods to integrate a C++ library into a Java program features the use Java Native Interface (JNI) library. JNI is used to create a wrapper over a C++ library, by independently defining the methods for each of the instances of the C++ library. Apart from being widely used in Android development, there are several drawbacks of this method [9]:

- Usage of JNI requires writing separate unmodifiable wrappers for each library the programmer intends to add in the Java framework
- Separate wrappers need to be written for each library and there is an absence of universal approach
- Bad C/C++ code in native library can/will cause core dumps/segmentation faults that the JVM can't recover from, so the whole application can crash

At the same time, the advantages of using the Client-Server architecture approach for text attribute modification are:

- **Platform Independence**

Google's Protocol Buffer is an open source, data-serializing program and is highly compatible with C++, Java and Python applications. It's usage in handling data from a server allows the developer to design the client side application in any manner he/she desires. The same server can cater to **any**^[10] type of text-editor provided that the client uses socket() or any data stream. The Open Source Project of YouCompleteMe is a typical example of this approach where Vim editor merely acts as a wrapper to display the received data from server. There the editor uses a plain HTTP/jSon server to read-write data. The current project offers a more convenient approach as declaring token.proto is significantly simpler than using XML or jSon data structure to serialize data.

- **Departure from CDT**

Clang based tools provide a very powerful alternative to the existing CDT library of Java for static analysis. This approach can reduce the usage of CDT in the IDE thus making it more convenient for the developer to analyze the source code of the application. It also provides better semantic analysis techniques, parallel to those of some genuinely good text-editors like Sublime Text and Vim.

BIBLIOGRAPHY

- 1) <http://llvm.org/>
- 2) <http://clang.llvm.org/>
- 3) https://wiki.eclipse.org/index.php/Rich_Client_Platform
- 4) https://en.wikipedia.org/wiki/Abstract_syntax_tree
- 5) <https://github.com/Valloric/YouCompleteMe>
- 6) <https://developers.google.com/protocol-buffers/>
- 7) <http://stackoverflow.com/questions/7358775/java-gui-frameworks-what-to-choose-swing-swt-awt-swingx-jgoodies-javafx>,
http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Feditors_highlighting.htm
- 8) http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Feditors_highlighting.htm
- 9) <http://www.careerride.com/JNI-advantages-disadvantages.aspx> ,
<http://stackoverflow.com/questions/1393937/disadvantages-of-using-java-native-interface>
- 10) <https://developers.google.com/protocol-buffers/docs/overview>
- 11) <http://llvm.org/devmtg/2010-11/Gregor-libclang.pdf>
- 12) llvm.org/devmtg/2013-04/klimek-slides.pdf

