



CPI-C for RUST on Linux Implementation Guide

Communications Server for Data Center Deployment

January, 2023

Authors:

Jeff L. Smith

jefsmith@us.ibm.com

Version 1.11

Table of Contents

1	Introduction	3
2	Example RUSTAPING program that initiates a conversation	5
2.1	RUST Cargo files.....	6
3	Example RUST_TP transaction program that receives a conversation	8
3.1	RUST Cargo files.....	10
4	CPI-C declaration for RUST in cpic.rs	11
5	Conclusion.....	12

History of changes:

Date	Subject	Owner
2022/07/21	Original Version	Jeff Smith
2022/08/09	Minimum level of Communication Server for Data Center Deployment V7.1 required	Jeff Smith
2023/01/17	Correct "rust_tp" key word requirement on Windows	Jeff Smith

1 Introduction

The Common Programming Interface for Communications (CPI-C) is a portable standard application programming interface for applications implementing APPC across different platforms. The CPI-C API is widely utilized for transaction sessions mainframe applications like CICS and IMS that track thousands of transactions per second. For Linux and Windows platforms, the IBM Communications Server for Data Center Deployment (CSDCD) provides CPI-C libraries in support of applications performing transactions via sessions of LU6.2 peer to peer communications. In recent years, the RUST programming language has been developed by an opensource community targeting improvements to security, reduced vulnerabilities in memory management, and efficiency.

RUST is closely compatible with C. The CPI-C for RUST implementation uses `std::os::raw::{c_int, c_char}` references in the RUST code to declare some CPI-C common variables and declare function entry points to the CPI-C libraries provided by the CSDCD. On Linux and Windows, the `cpic.rs` include file declares what is needed to make calls into CPI-C. Not all CPI-C interface calls are supported in the CPI-C for RUST. There are some extensions to the CPI-C 2.0 standard interface implemented for the CSDCD that are not available in RUST. For the most part, these extensions are not required, and the information can be extracted through the common functions supported.

The implementation of the CPI-C for Rust described in this guide closely mirrors the CPI-C for Java implementation documented in the CPI-C Programmers manual that accompanies the CSDCD. The current implementation examples provided in this guide and descriptions are for Linux and Windows systems. The CSDCD also supports CPI-C on AIX. When RUST is available on AIX, this guide will be updated to include details for that platform.

The minimum level of CSDCD required to run CPI-C for Rust is V7.1.0.0. There are some specific C interface calls that need mapping into the CPI-C for Rust and to accommodate this, the 7.1 level must be the minimum level.

Transaction sessions require a pair of nodes, one is the sender and the other the receiver. This guide provides the examples of CPI-C for Rust between a sending node (RUSTAPING) and a receiving node (RUST_TP). The first example application, RUSTAPING, sends “pings” over APPC, LU6.2, sessions to a receiving transaction program on a SNA node. The Communications Server can be configured to allocate a session to the APINGD transaction program which most SNA nodes support by default. It can also be configured to allocate a session to RUST_TP transaction program, the second example program provided. The RUST_TP example program, is a transaction program that receives “pings” from RUSTAPING (or from the APING utility provide with the Communications Server). The RUST_TP example logs the connection and the amount of data pinged to a log file. The programs provided are examples of how an application can be written to connect to a partner LU and send data back and forth and

then deallocate the session. This guide will also describe how to configure a SNA node to dynamically invoke the RUST_TP transaction program.

2 Example RUSTAPING program that initiates a conversation

The example program, RUSTAPING, demonstrates how to initiate an APPC session to a partner LU, send and receive data from the partner LU and deallocate the session. The program requires the symbolic name of a CPIC_Side_Info record that defines a partner LU, TP name of the partner application and the mode characteristics for the connection used. To define a CPIC_Side_Info record that allocates a session to the RUST_TP example program, use the snaadmin command line tool:

```
snaadmin define_cplic_side_info,sym_dest_name=RUSTTEST,  
        partner_lu_name=USIBMNM.LTLWGN9, mode_name=#BATCH, tp_name=RUST_TP
```

A similar CPIC_Side_Info record can be created to define a connection using APINGD on any SNA node that will respond to RUSTAPING:

```
snaadmin define_cplic_side_info,sym_dest_name=RUSTPING,  
        partner_lu_name=APPN.X64LU62, mode_name=#BATCH, tp_name=APINGD
```

To view the CPI-C Side Info on a Communications Server for Linux, or AIX, you can issue the following command:

```
snaadmin query_cplic_side_info
```

Example output could look like the following:

```
list_options = FIRST_IN_LIST  
  
sym_dest_name = RUSTPING  
description = ""  
partner_lu_name = APPN.X64LU62  
tp_name_type = APPLICATION_TP  
tp_name = APINGD  
mode_name = #BATCH  
conversation_security_type = NONE  
security_user_id = ""  
security_password = ""  
lu_alias = ""  
  
sym_dest_name = RUSTTEST  
description = ""  
partner_lu_name = USIBMNM.LTLWGN9  
tp_name_type = APPLICATION_TP  
tp_name = RUST_TP  
mode_name = #BATCH  
conversation_security_type = NONE  
security_user_id = ""  
security_password = ""  
lu_alias = ""
```

There are optional parameters for RUSTAPING that specify the iteration (-i) count for the number of “pings” to send, the count (-c) for the number of messages per ping to send, the size (-s) of each message (maximum 32,000 bytes), and a flag to indicate a line number (-#) when displaying output for each ping sent:

Windows:

```
rustaping_win64 -i 10 -c 3 -s 250 -# RUSTTEST
```

Linux:

```
./rustaping_x86_64_bin -i 10 -c 3 -s 250 -# RUSTTEST
```

IBM aping for RUST version 1.0: APPC echo test with timings.

Allocate duration: 1.0 ms

Local LU Alias: sls15sp2 connected to Partner LU: USIBMNM.LTLWGN9

Program startup and Confirm duration: 1.0 ms

```
Connected to partner running on: RUST_TP for Linux 1.0
#1 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1563.8 (Mb/s) 12.510
#2 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1658.3 (Mb/s) 13.266
#3 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1647.3 (Mb/s) 13.179
#4 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1627.1 (Mb/s) 13.017
#5 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1727.3 (Mb/s) 13.818
#6 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1636.6 (Mb/s) 13.093
#7 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1605.8 (Mb/s) 12.846
#8 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1588.5 (Mb/s) 12.708
#9 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1495.0 (Mb/s) 11.960
#10 Duration(ms): 0.3 Data Sent(bytes): 1500 Data Rate(KB/s) 1453.6 (Mb/s) 11.629
-- Totals:
Duration(ms): 3.1 Total Data (bytes): 15000 Data Rate(KB/s) 1596.6 (Mb/s) 12.773
```

The output provides the duration in milliseconds for the allocate, the Local LU alias and partner LU names, the exchange identity flow time in milliseconds, and ping transaction rates are displayed. The default parameters, if not specified, are to send 3 pings of 1000 bytes each using the CPIC_Side_Info record APINGD.

2.1 RUST Cargo files

The packages provided with this guide include RUST Cargo directory contents for Windows and Linux platforms. The Linux on Intel (x86_64) package is different from the Linux on Power (ppc64le) and Linux on IBM Z (s390x) package due to how characters are defined in the RUST C libraries. Other than this, the packages all have similar build scripts and source code.

RUST uses a build utility, cargo, to compile and link the application to Rust and C libraries. The example packages provided contain the cargo directory for “rustaping”. There is a build script that shows how to define the require environment variables and parameters to build the rustaping program. There is a “src” directory with 3 source files. All RUST programs require a main.rs file for an executable program. The cpic.rs file is the CPI-C include file that defines the API calls. The lib.rs has generic routines for reading in parameters passed in on the command line and printing usage information:

Package files:

rustaping/

 build_rustaping_cmd (build_rustaping.bat on Windows)

 src/

 cpic.rs

 lib.rs

 main.rs

To build the rustaping program, run the build script provided. For rustaping, the default build script will create a debug version under a “target/debug” directory. To build a non-debug, or release version, change the “cargo build --verbose” command in the build script to “cargo build --release”. The release build will create a “target/release” directory where the rustaping program (rustaping.exe on Windows) will be found.

3 Example RUST_TP transaction program that receives a conversation

The example program RUST_TP is a transaction program that can be configured to be invoked dynamically by Communications Server when an allocate message is received from a partner SNA node. The partner application would be running the RUSTAPING program or APING. The RUST_TP first makes a call to CPI-C to register itself as the transaction program "RUST_TP". This acts like a specific queue for allocation request to this TP program. The program then issues an "accept conversation" call to receive any incoming request to start a conversation. The configuration definition for the TP can specify how long the queue can remain open with no allocate requests before timing out.

Once the conversation is accepted, the transaction program extracts the local LU and partner LU information to log to the file /tmp/rust_tp.txt. This file logs the times that allocate requests are processed between the LU-LU pair and the size of the data sent back and forth, in number of bytes. Example of the log file:

```
cat /tmp/rust_tp.txt
IBM aping for RUST version 1.0: APPC echo test with timings.

Local time is: 2021-10-27 16:05:00.963584436 -04:00
  Local LU ALias: ltlwgn9  connected to Parnter LU: USIBMNM.LTLWJLS
  Program startup and Confirm complete
  \x01\x02,\x02RUSTAPING for Linux 1.0
  Total bytes received from partner: 5000

Local time is: 2021-10-27 16:05:43.312382087 -04:00
  Local LU ALias: ltlwgn9  connected to Parnter LU: USIBMNM.LTLWJLS
  Program startup and Confirm complete
  \x01\x02,\x02RUSTAPING for Linux 1.0
  Total bytes received from partner: 5000

error at calling point CMA CCP call: : 25
```

The last entry above is the timeout response, error code 25, when there are no more allocates to process after the configured number of seconds for the TP.

Linux:

On Linux, the TP configuration can be defined using a utility command, `snatpinstall`. The command syntax is as follows:

```
snatpinstall -a rust_tp.tps
```

You can query the TP definitions for the Linux system by issuing:

```
snatpinstall -q
```

The rust_tp.tps file on Linux will have these parameters:

```
[RUST_TP]
PATH=/usr/local/bin/rust_tp
ENV=APPCTPN=RUST_TP
USERID=bin
GROUP=sna
TIMEOUT=60
TYPE=QUEUED
```

Windows:

On Windows, the TP configuration can be defined using the utility tpinst32.exe. The command would be

```
tpinst32 -a rust_tp.tps
```

To see the TP definitions for the Windows platform, issue:

```
tpinst32 -q
```

The rust_tp.tps file on Windows will should these parameters:

```
[RUST_TP]
PATH=C:\temp\rust_tp.exe
ENV=APPCTPN=RUST_TP
TIMEOUT=60
TYPE=QUEUED
SHOW=NORMAL
SECURITY_TYPE=APPLICATION
USERID=ADMINISTRATOR
```

Where the USERID must be set to the User who would have permissions to send/receive SNA CPI-C messages. The TIMEOUT can be 1 to 65535 seconds, or -1 to mean no timeout. A value of 60 means that after the rust_tp issues a CMACCP() call to accept an Allocate message from a partner LU, the Remote API client will wait 60 seconds before timing out if no Allocate message is received for the TP. The PATH variable must point to the path where the RUST_TP executable resides. The SECURITY_TYPE must be APPLICATION for the rust_tp program.

For transaction programs on Windows receiving Allocate messages, the Remote API client requires that variable APPCTPN be defined in a TP name record under the SNA client section in the Windows Registry. To do this, use the following script to define the TP name in the Windows Registry entry for the configuration parameter for the SNA client:

```
REM
REM   Set the TP executable name (without the .exe) as a key parameter and
APPCTPN to RUST_TP as a sub-key
REM
reg add "HKLM\SOFTWARE\SNA Client\SxCClient\Parameters\rust_tp"
REM
reg add "HKLM\SOFTWARE\SNA Client\SxCClient\Parameters\rust_tp" /v APPCTPN
/t REG_SZ /f /d RUST_TP
```

With the “rust_tp” key word defined as a TP program name to the Remote API client and in the Windows Registry, the program will now be invoked when incoming Allocate messages are received for that TP.

3.1 RUST Cargo files

The Cargo directory for the rust_tp build package is provided as an example of writing a Transaction Program. RUST uses a build utility, cargo, to compile and link the application to Rust and C libraries. The example packages provided contain the cargo directory for “rust_tp”. There is a build script that shows how to define the require environment variables and parameters to build the rustaping program. There is a “src” directory with 2 source files.

All RUST programs require a main.rs file for an executable program. The cpic.rs file is the CPI-C include file that defines the API calls.

Package files:

```
rust_tp/
    build_rust_tp_cmd (build_rust_tp.bat on Windows)
    src/
        cpic.rs
        main.rs
```

To build the rust_tp program, run the build script provided. For rust_tp, the default build script will create a release version under a “target/release” directory. To build a debug, version, change the “cargo build --release” command in the build script to “cargo build --verbose”. The release build will create a “target/debug” directory where the rust_tp program (rust_tp.exe on Windows) will be found. You must copy the rust_tp.exe to the file path specified in the RUST_TP definition set using the tpinst32.exe utility.

4 CPI-C declaration for RUST in cpic.rs

RUST includes modules similar to how C includes header files. The statement in RUST to include CPI-C declarations is:

```
mod cpic;
```

This statement should be one of the first statements in the main.rs file used for RUST. The cpic.rs file would reside in the src directory under the name of the program, rustaping, or rust_tp.

The cpic.rs file contains the definitions for the CPI-C interface return codes, session states, and C function routines found in the libcpic.so file. The APPC, NOF and sna_r (multi-threaded SNA routines) library files that are required to complete CPI-C calls. The CPI-C interface is a higher level interface that calls APPC verbs underneath. The libraries are listed in the cpic.rs file in this manner, where the name will be prefixed by “lib” and appended with “.so” to list the dynamically loaded library file (libcpic.so for instance):

```
#[link(name = "cpic")]
#[link(name = "appc")]
#[link(name = "nof")]
#[link(name = "sna_r")]
```

The RUST application links the C libraries at build time using the RUSTFLAGS keyword to pass the link option to the compiler. It is important to build the product with the following exported environment variables that define to the executable where the dynamically loadable libraries reside:

Linux:

```
export LD_LIBRARY_PATH=/opt/ibm/sna/lib64:/usr/lib64
export LD_RUN_PATH=/opt/ibm/sna/lib64:/usr/lib64
RUSTFLAGS='-L /opt/ibm/sna/lib64' cargo build --release
```

Windows:

```
set LIB=C:\ibmcs\W64CLI\sdk64
```

When invoking the “cargo build” command to build the Rust executable, the “--release” option is needed to build a non-debug version of the application. If this parameter is omitted, then a debug version of the executable is built in the target/debug directory. The release version will be located in the directory target/release.

5 Conclusion

CPI-C is a standard API for issuing transactions over SNA for large database applications. Critical business applications like IBM CICS and IMS use CPI-C APIs to guarantee transactions at a high rates. There are still many legacy COBAL applications running on the mainframe that use CPI-C APIs as a means for performing transactions.

Rust is a powerful abstract language with security and performance built into the design of the code. Strong type definitions, strict rules for memory management , and increasing deployment of platform integration make Rust very attractive to developers today.

The examples provided in this package are to help a Rust developer use the legacy API for SNA transactions that come with the distributed Communications Server products. CPI-C is a standard API for making development across platforms simpler. Rust already has cross platform implementations with many operating system compatible Cargo packages.

References:

Library for the Communications Server for Data Center Deployment on Linux:

<https://www.ibm.com/support/pages/communications-server-data-center-deployment-linux-library>

Good resources on Rust:

Chapter 23: Foreign Functions – **Programming Rust, 2nd Edition**

<https://www.oreilly.com/library/view/programming-rust-2nd/9781492052586/>