Code of Remote Attestation (Client)

App

app.cpp

```
#include <stdio.h>
#include <limits.h>
#include <unistd.h>
// Needed for definition of remote attestation messages.
#include "remote_attestation_result.h"

#include "isv_enclave_u.h"

// Needed to call untrusted key exchange library APIs, i.e. sgx_ra_proc_msg2.
#include "sgx_ukey_exchange.h"

// Needed to get service provider's information, in your real project, you will
// need to talk to real server.
#include "network_ra.h"

// Needed to create enclave and do ecall.
#include "sgx_urts.h"

// Needed to query extended epid group id.
#include "sgx_uae_service.h"

#include "service_provider.h"

#ifndef SAFE_FREE
#define SAFE_FREE(ptr)     \
    {                      \
        if (NULL != (ptr)) \
        {                  \
            free(ptr);     \
            (ptr) = NULL;  \
        }                  \
    }
#endif

// In addition to generating and sending messages, this application
// can use pre-generated messages to verify the generation of
// messages and the information flow.
#include "sample_messages.h"

#define ENCLAVE_PATH "isv_enclave.signed.so"

uint8_t *msg1_samples[] = {msg1_sample1, msg1_sample2};
uint8_t *msg2_samples[] = {msg2_sample1, msg2_sample2};
uint8_t *msg3_samples[] = {msg3_sample1, msg3_sample2};
uint8_t *attestation_msg_samples[] =
```

```
    {attestation_msg_sample1, attestation_msg_sample2};

extern char sendbuf[BUFSIZ]; //数据传送的缓冲区
extern char recvbuf[BUFSIZ];

// Some utility functions to output some of the data structures passed between
// the ISV app and the remote attestation service provider.
void PRINT_BYTE_ARRAY(
    FILE *file, void *mem, uint32_t len)
{
    if (!mem || !len)
    {
        fprintf(file, "\n( null )\n");
        return;
    }
    uint8_t *array = (uint8_t *)mem;
    fprintf(file, "%u bytes:\n{\n", len);
    uint32_t i = 0;
    for (i = 0; i < len - 1; i++)
    {
        fprintf(file, "0x%x, ", array[i]);
        if (i % 8 == 7)
            fprintf(file, "\n");
    }
    fprintf(file, "0x%x ", array[i]);
    fprintf(file, "\n}\n");
}

void PRINT_ATTESTATION_SERVICE_RESPONSE(
    FILE *file,
    ra_samp_response_header_t *response)
{
    if (!response)
    {
        fprintf(file, "\t\n( null )\n");
        return;
    }

    fprintf(file, "RESPONSE TYPE:   0x%x\n", response->type);
    fprintf(file, "RESPONSE STATUS: 0x%x 0x%x\n", response->status[0],
            response->status[1]);
    fprintf(file, "RESPONSE BODY SIZE: %u\n", response->size);

    if (response->type == TYPE_RA_MSG2)
    {
        sgx_ra_msg2_t *p_msg2_body = (sgx_ra_msg2_t *)(response->body);

        fprintf(file, "MSG2 gb - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->g_b), sizeof(p_msg2_body->g_b));

        fprintf(file, "MSG2 spid - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->spid), sizeof(p_msg2_body->spid));

        fprintf(file, "MSG2 quote_type : %hx\n", p_msg2_body->quote_type);

        fprintf(file, "MSG2 kdf_id : %hx\n", p_msg2_body->kdf_id);

        fprintf(file, "MSG2 sign_gb_ga - ");
```

```
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sign_gb_ga),
                         sizeof(p_msg2_body->sign_gb_ga));

        fprintf(file, "MSG2 mac - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->mac), sizeof(p_msg2_body->mac));

        fprintf(file, "MSG2 sig_rl - ");
        PRINT_BYTE_ARRAY(file, &(p_msg2_body->sig_rl),
                         p_msg2_body->sig_rl_size);
    }
    else if (response->type == TYPE_RA_ATT_RESULT)
    {
        sample_ra_att_result_msg_t *p_att_result =
            (sample_ra_att_result_msg_t *)(response->body);
        fprintf(file, "ATTESTATION RESULT MSG platform_info_blob - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->platform_info_blob),
                         sizeof(p_att_result->platform_info_blob));

        fprintf(file, "ATTESTATION RESULT MSG mac - ");
        PRINT_BYTE_ARRAY(file, &(p_att_result->mac), sizeof(p_att_result->mac));

        fprintf(file, "ATTESTATION RESULT MSG secret.payload_tag - %u bytes\n",
                p_att_result->secret.payload_size);

        fprintf(file, "ATTESTATION RESULT MSG secret.payload - ");
        PRINT_BYTE_ARRAY(file, p_att_result->secret.payload,
                         p_att_result->secret.payload_size);
    }
    else
    {
        fprintf(file, "\nERROR in printing out the response. "
                      "Response of type not supported %d\n",
                response->type);
    }
}

// This sample code doesn't have any recovery/retry mechanisms for the remote
// attestation. Since the enclave can be lost due S3 transitions, apps
// susceptible to S3 transitions should have logic to restart attestation in
// these scenarios.
#define _T(x) x
int main(int argc, char *argv[])
{
    int ret = 0;
    ra_samp_request_header_t *p_msg0_full = NULL;
    ra_samp_response_header_t *p_msg0_resp_full = NULL;
    ra_samp_request_header_t *p_msg1_full = NULL;
    ra_samp_response_header_t *p_msg2_full = NULL;
    sgx_ra_msg3_t *p_msg3 = NULL;
    ra_samp_response_header_t *p_att_result_msg_full = NULL;
    sgx_enclave_id_t enclave_id = 0;
    int enclave_lost_retry_time = 1;
    int busy_retry_time = 4;
    sgx_ra_context_t context = INT_MAX;
    sgx_status_t status = SGX_SUCCESS;
    ra_samp_request_header_t *p_msg3_full = NULL;

    int32_t verify_index = -1;
```

```
    int32_t verification_samples = sizeof(msg1_samples) /
sizeof(msg1_samples[0]);

    FILE *OUTPUT = stdout;

#define VERIFICATION_INDEX_IS_VALID() (verify_index > 0 && \
                                       verify_index <= verification_samples)
#define GET_VERIFICATION_ARRAY_INDEX() (verify_index - 1)

    if (argc > 1)
    {

        verify_index = atoi(argv[1]);

        if (VERIFICATION_INDEX_IS_VALID())
        {
            fprintf(OUTPUT, "\nVerifying precomputed attestation messages "
                            "using precomputed values# %d\n",
                    verify_index);
        }
        else
        {
            fprintf(OUTPUT, "\nValid invocations are:\n");
            fprintf(OUTPUT, "\n\tisv_app\n");
            fprintf(OUTPUT, "\n\tisv_app <verification index>\n");
            fprintf(OUTPUT, "\nValid indices are [1 - %d]\n",
                    verification_samples);
            fprintf(OUTPUT, "\nUsing a verification index uses precomputed "
                            "messages to assist debugging the remote attestation
"
                            "service provider.\n");
            return -1;
        }
    }

    // SOCKET: connect to server
    if (client("127.0.0.1", 12333) != 0)
    {
        fprintf(OUTPUT, "Connect Server Error, Exit!\n");
        return 0;
    }
    // Preparation for remote attestation by configuring extended epid group id.
    {
        uint32_t extended_epid_group_id = 0;
        ret = sgx_get_extended_epid_group_id(&extended_epid_group_id);
        if (SGX_SUCCESS != ret)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call sgx_get_extended_epid_group_id fail
[%s].",
                    __FUNCTION__);
            return ret;
        }
        fprintf(OUTPUT, "\nCall sgx_get_extended_epid_group_id success.");

        p_msg0_full = (ra_samp_request_header_t *)
            malloc(sizeof(ra_samp_request_header_t) + sizeof(uint32_t));
        if (NULL == p_msg0_full)
```

```
        {
            ret = -1;
            goto CLEANUP;
        }
        p_msg0_full->type = TYPE_RA_MSG0;
        p_msg0_full->size = sizeof(uint32_t);

        *(uint32_t *)((uint8_t *)p_msg0_full + sizeof(ra_samp_request_header_t))
= extended_epid_group_id;
        {

            fprintf(OUTPUT, "\nMSG0 body generated -\n");

            PRINT_BYTE_ARRAY(OUTPUT, p_msg0_full->body, p_msg0_full->size);
        }
        // The ISV application sends msg0 to the SP.
        // The ISV decides whether to support this extended epid group id.
        fprintf(OUTPUT, "\nSending msg0 to remote attestation service
provider.\n");

        // SOCKET: send & recv
        ret = ra_network_send_receive("http://SampleServiceProvider.intel.com/",
                                      p_msg0_full,
                                      &p_msg0_resp_full);
        if (ret != 0)
        {
            fprintf(OUTPUT, "\nError, ra_network_send_receive for msg0 failed "
                            "[%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nSent MSG0 to remote attestation service.\n");
    }
    // Remote attestation will be initiated the ISV server challenges the ISV
    // app or if the ISV app detects it doesn't have the credentials
    // (shared secret) from a previous attestation required for secure
    // communication with the server.
    {
        // ISV application creates the ISV enclave.
        int launch_token_update = 0;
        sgx_launch_token_t launch_token = {0};
        memset(&launch_token, 0, sizeof(sgx_launch_token_t));
        do
        {
            ret = sgx_create_enclave(_T(ENCLAVE_PATH),
                                     SGX_DEBUG_FLAG,
                                     &launch_token,
                                     &launch_token_update,
                                     &enclave_id, NULL);
            if (SGX_SUCCESS != ret)
            {
                ret = -1;
                fprintf(OUTPUT, "\nError, call sgx_create_enclave fail [%s].",
                        __FUNCTION__);
                goto CLEANUP;
            }
            fprintf(OUTPUT, "\nCall sgx_create_enclave success.");
```

```
            ret = enclave_init_ra(enclave_id,
                                  &status,
                                  false,
                                  &context);
            //Ideally, this check would be around the full attestation flow.
        } while (SGX_ERROR_ENCLAVE_LOST == ret && enclave_lost_retry_time--);

        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_init_ra fail [%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nCall enclave_init_ra success.");

        // isv application call uke sgx_ra_get_msg1
        p_msg1_full = (ra_samp_request_header_t *)
            malloc(sizeof(ra_samp_request_header_t) + sizeof(sgx_ra_msg1_t));
        if (NULL == p_msg1_full)
        {
            ret = -1;
            goto CLEANUP;
        }
        p_msg1_full->type = TYPE_RA_MSG1;
        p_msg1_full->size = sizeof(sgx_ra_msg1_t);
        do
        {
            ret = sgx_ra_get_msg1(context, enclave_id, sgx_ra_get_ga,
                                  (sgx_ra_msg1_t *)((uint8_t *)p_msg1_full +
sizeof(ra_samp_request_header_t)));
            sleep(3); // Wait 3s between retries
        } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
        if (SGX_SUCCESS != ret)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call sgx_ra_get_msg1 fail [%s].",
                    __FUNCTION__);
            goto CLEANUP;
        }
        else
        {
            fprintf(OUTPUT, "\nCall sgx_ra_get_msg1 success.\n");

            fprintf(OUTPUT, "\nMSG1 body generated -\n");

            PRINT_BYTE_ARRAY(OUTPUT, p_msg1_full->body, p_msg1_full->size);
        }

        if (VERIFICATION_INDEX_IS_VALID())
        {

            memcpy_s(p_msg1_full->body, p_msg1_full->size,
                     msg1_samples[GET_VERIFICATION_ARRAY_INDEX()],
                     p_msg1_full->size);

            fprintf(OUTPUT, "\nInstead of using the recently generated MSG1, "
                            "we will use the following precomputed MSG1 -\n");
```

```
            PRINT_BYTE_ARRAY(OUTPUT, p_msg1_full->body, p_msg1_full->size);
        }

        // The ISV application sends msg1 to the SP to get msg2,
        // msg2 needs to be freed when no longer needed.
        // The ISV decides whether to use linkable or unlinkable signatures.
        p_msg2_full = (ra_samp_response_header_t *)malloc(180);
        memset(p_msg2_full, 0, 180);
        if (NULL == p_msg2_full)
        {
            ret = -1;
            goto CLEANUP;
        }
        ret = ra_network_send_receive("http://SampleServiceProvider.intel.com/",
                                      p_msg1_full,
                                      &p_msg2_full);

        if ((ret == 0) || (p_msg2_full == NULL))
        {
            fprintf(OUTPUT, "\nError, ra_network_send_receive for msg1 failed "
                            "[%s].",
                    __FUNCTION__);
            if (VERIFICATION_INDEX_IS_VALID())
            {
                fprintf(OUTPUT, "\nBecause we are in verification mode we will "
                                "ignore this error.\n");
                fprintf(OUTPUT, "\nInstead, we will pretend we received the "
                                "following MSG2 - \n");

                SAFE_FREE(p_msg2_full);
                ra_samp_response_header_t *precomputed_msg2 =
                    (ra_samp_response_header_t
*)msg2_samples[GET_VERIFICATION_ARRAY_INDEX()];
                const size_t msg2_full_size = sizeof(ra_samp_response_header_t) +
precomputed_msg2->size;
                p_msg2_full =
                    (ra_samp_response_header_t *)malloc(msg2_full_size);
                if (NULL == p_msg2_full)
                {
                    ret = -1;
                    goto CLEANUP;
                }
                memcpy_s(p_msg2_full, msg2_full_size, precomputed_msg2,
                         msg2_full_size);

                PRINT_BYTE_ARRAY(OUTPUT, p_msg2_full,
                                 sizeof(ra_samp_response_header_t) + p_msg2_full-
>size);
            }
            else
            {
                goto CLEANUP;
            }
        }
        else
        {
            // Successfully sent msg1 and received a msg2 back.
```

```
            // Time now to check msg2.
            if (TYPE_RA_MSG2 != p_msg2_full->type)
            {

                fprintf(OUTPUT, "\nError, didn't get MSG2 in response to MSG1. "
                                "[%s]. receive type is %d\n",
                        __FUNCTION__, p_msg2_full->type);

                PRINT_BYTE_ARRAY(OUTPUT, p_msg2_full,
                                 176);
                if (VERIFICATION_INDEX_IS_VALID())
                {
                    fprintf(OUTPUT, "\nBecause we are in verification mode we "
                                    "will ignore this error.");
                }
                else
                {
                    goto CLEANUP;
                }
            }

            fprintf(OUTPUT, "\nSent MSG1 to remote attestation service "
                            "provider. Received the following MSG2:\n");
            PRINT_BYTE_ARRAY(OUTPUT, p_msg2_full,
                             sizeof(ra_samp_response_header_t) + p_msg2_full-
>size);

            fprintf(OUTPUT, "\nA more descriptive representation of MSG2:\n");
            PRINT_ATTESTATION_SERVICE_RESPONSE(OUTPUT, p_msg2_full);

            if (VERIFICATION_INDEX_IS_VALID())
            {
                // The response should match the precomputed MSG2:
                ra_samp_response_header_t *precomputed_msg2 =
                    (ra_samp_response_header_t *)
                        msg2_samples[GET_VERIFICATION_ARRAY_INDEX()];
                if (MSG2_BODY_SIZE !=
                        sizeof(ra_samp_response_header_t) + p_msg2_full->size ||
                    memcmp(precomputed_msg2, p_msg2_full,
                           sizeof(ra_samp_response_header_t) + p_msg2_full-
>size))
                {
                    fprintf(OUTPUT, "\nVerification ERROR. Our precomputed "
                                    "value for MSG2 does NOT match.\n");
                    fprintf(OUTPUT, "\nPrecomputed value for MSG2:\n");
                    PRINT_BYTE_ARRAY(OUTPUT, precomputed_msg2,
                                     sizeof(ra_samp_response_header_t) +
precomputed_msg2->size);
                    fprintf(OUTPUT, "\nA more descriptive representation "
                                    "of precomputed value for MSG2:\n");
                    PRINT_ATTESTATION_SERVICE_RESPONSE(OUTPUT,
                                                       precomputed_msg2);
                }
                else
                {
                    fprintf(OUTPUT, "\nVerification COMPLETE. Remote "
                                    "attestation service provider generated a "
                                    "matching MSG2.\n");
```

```
                }
            }
        }

        sgx_ra_msg2_t *p_msg2_body = (sgx_ra_msg2_t *)((uint8_t *)p_msg2_full +
sizeof(ra_samp_response_header_t));

        uint32_t msg3_size = 0;
        if (VERIFICATION_INDEX_IS_VALID())
        {
            // We cannot generate a valid MSG3 using the precomputed messages
            // we have been using. We will use the precomputed msg3 instead.
            msg3_size = MSG3_BODY_SIZE;
            p_msg3 = (sgx_ra_msg3_t *)malloc(msg3_size);
            if (NULL == p_msg3)
            {
                ret = -1;
                goto CLEANUP;
            }
            memcpy_s(p_msg3, msg3_size,
                     msg3_samples[GET_VERIFICATION_ARRAY_INDEX()], msg3_size);
            fprintf(OUTPUT, "\nBecause MSG1 was a precomputed value, the MSG3 "
                            "we use will also be. PRECOMPUTED MSG3 - \n");
        }
        else
        {
            busy_retry_time = 2;
            // The ISV app now calls uKE sgx_ra_proc_msg2,
            // The ISV app is responsible for freeing the returned p_msg3!!
            do
            {
                ret = sgx_ra_proc_msg2(context,
                                       enclave_id,
                                       sgx_ra_proc_msg2_trusted,
                                       sgx_ra_get_msg3_trusted,
                                       p_msg2_body,
                                       p_msg2_full->size,
                                       &p_msg3,
                                       &msg3_size);
            } while (SGX_ERROR_BUSY == ret && busy_retry_time--);
            if (!p_msg3)
            {
                fprintf(OUTPUT, "\nError, call sgx_ra_proc_msg2 fail. "
                                "p_msg3 = 0x%p [%s].",
                        p_msg3, __FUNCTION__);
                ret = -1;
                goto CLEANUP;
            }
            if (SGX_SUCCESS != (sgx_status_t)ret)
            {
                fprintf(OUTPUT, "\nError, call sgx_ra_proc_msg2 fail. "
                                "ret = 0x%08x [%s].",
                        ret, __FUNCTION__);
                ret = -1;
                goto CLEANUP;
            }
            else
            {
```

```
                fprintf(OUTPUT, "\nCall sgx_ra_proc_msg2 success.\n");
                fprintf(OUTPUT, "\nMSG3 - \n");
            }
        }

        PRINT_BYTE_ARRAY(OUTPUT, p_msg3, msg3_size);

        p_msg3_full = (ra_samp_request_header_t *)malloc(
            sizeof(ra_samp_request_header_t) + msg3_size);
        if (NULL == p_msg3_full)
        {
            ret = -1;
            goto CLEANUP;
        }
        p_msg3_full->type = TYPE_RA_MSG3;
        p_msg3_full->size = msg3_size;
        if (memcpy_s(p_msg3_full->body, msg3_size, p_msg3, msg3_size))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        // The ISV application sends msg3 to the SP to get the attestation
        // result message, attestation result message needs to be freed when
        // no longer needed. The ISV service provider decides whether to use
        // linkable or unlinkable signatures. The format of the attestation
        // result is up to the service provider. This format is used for
        // demonstration.  Note that the attestation result message makes use
        // of both the MK for the MAC and the SK for the secret. These keys are
        // established from the SIGMA secure channel binding.
        p_att_result_msg_full = (ra_samp_response_header_t *)malloc(180);
        memset(p_msg2_full, 0, 180);
        ret = ra_network_send_receive("http://SampleServiceProvider.intel.com/",
                                      p_msg3_full,
                                      &p_att_result_msg_full);
        if (ret == 0 || p_att_result_msg_full == NULL)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, sending msg3 failed [%s].", __FUNCTION__);
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nReceive attestation data is\n");
        PRINT_BYTE_ARRAY(OUTPUT, p_att_result_msg_full, 180);
        sample_ra_att_result_msg_t *p_att_result_msg_body =
            (sample_ra_att_result_msg_t *)((uint8_t *)p_att_result_msg_full +
sizeof(ra_samp_response_header_t));
        if (TYPE_RA_ATT_RESULT != p_att_result_msg_full->type)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError. Sent MSG3 successfully, but the message "
                            "received was NOT of type att_msg_result. Type = "
                            "%d. [%s].",
                    p_att_result_msg_full->type,
                    __FUNCTION__);
            goto CLEANUP;
        }
```

```
        else
        {
            fprintf(OUTPUT, "\nSent MSG3 successfully. Received an attestation "
                            "result message back\n.");
            if (VERIFICATION_INDEX_IS_VALID())
            {
                if (ATTESTATION_MSG_BODY_SIZE != p_att_result_msg_full->size ||
                    memcmp(p_att_result_msg_full->body,

attestation_msg_samples[GET_VERIFICATION_ARRAY_INDEX()],
                           p_att_result_msg_full->size))
                {
                    fprintf(OUTPUT, "\nSent MSG3 successfully. Received an "
                                    "attestation result message back that did "
                                    "NOT match the expected value.\n");
                    fprintf(OUTPUT, "\nEXPECTED ATTESTATION RESULT -");
                    PRINT_BYTE_ARRAY(OUTPUT,

attestation_msg_samples[GET_VERIFICATION_ARRAY_INDEX()],
                                     ATTESTATION_MSG_BODY_SIZE);
                }
            }
        }

        fprintf(OUTPUT, "\nATTESTATION RESULT RECEIVED - ");
        PRINT_BYTE_ARRAY(OUTPUT, p_att_result_msg_full->body,
                         p_att_result_msg_full->size);
        fprintf(OUTPUT, "\natt data Body - ");
        PRINT_BYTE_ARRAY(OUTPUT, p_att_result_msg_body,
                         p_att_result_msg_full->size);

        if (VERIFICATION_INDEX_IS_VALID())
        {
            fprintf(OUTPUT, "\nBecause we used precomputed values for the "
                            "messages, the attestation result message will "
                            "not pass further verification tests, so we will "
                            "skip them.\n");
            goto CLEANUP;
        }

        // Check the MAC using MK on the attestation result message.
        // The format of the attestation result message is ISV specific.
        // This is a simple form for demonstration. In a real product,
        // the ISV may want to communicate more information.
        ret = verify_att_result_mac(enclave_id,
                                    &status,
                                    context,
                                    (uint8_t *)&p_att_result_msg_body-
>platform_info_blob,
                                    sizeof(ias_platform_info_blob_t),
                                    (uint8_t *)&p_att_result_msg_body->mac,
                                    sizeof(sgx_mac_t));
        if ((SGX_SUCCESS != ret) ||
            (SGX_SUCCESS != status))
        {
            ret = -1;
            fprintf(OUTPUT, "\nError: INTEGRITY FAILED - attestation result "
                            "message MK based cmac failed in [%s].",
```

```
                    __FUNCTION__);
            goto CLEANUP;
        }

        bool attestation_passed = true;
        // Check the attestation result for pass or fail.
        // Whether attestation passes or fails is a decision made by the ISV
Server.
        // When the ISV server decides to trust the enclave, then it will return
success.
        // When the ISV server decided to not trust the enclave, then it will
return failure.
        if (0 != p_att_result_msg_full->status[0] || 0 != p_att_result_msg_full-
>status[1])
        {
            fprintf(OUTPUT, "\nError, attestation result message MK based cmac "
                            "failed in [%s]. %d %d ",
                    __FUNCTION__,
                    p_att_result_msg_full->status[0],
                    p_att_result_msg_full->status[1]);
            attestation_passed = false;
            goto CLEANUP;
        }
        // The attestation result message should contain a field for the Platform
        // Info Blob (PIB).  The PIB is returned by attestation server in the
attestation report.
        // It is not returned in all cases, but when it is, the ISV app
        // should pass it to the blob analysis API called
sgx_report_attestation_status()
        // along with the trust decision from the ISV server.
        // The ISV application will take action based on the update_info.
        // returned in update_info by the API.
        // This call is stubbed out for the sample.
        //
        // sgx_update_info_bit_t update_info;
        // ret = sgx_report_attestation_status(
        //     &p_att_result_msg_body->platform_info_blob,
        //     attestation_passed ? 0 : 1, &update_info);

        // Get the shared secret sent by the server using SK (if attestation
        // passed)
        if (attestation_passed)
        {
            fprintf(OUTPUT,
                    "\nthe size of secret is %d, the secret is\n",
                    p_att_result_msg_body->secret.payload_size);
            PRINT_BYTE_ARRAY(OUTPUT, &p_att_result_msg_body->secret, 40);
            fprintf(OUTPUT, "\nthe context is:\n");
            PRINT_BYTE_ARRAY(OUTPUT, &context, sizeof(context));
            ret = put_secret_data(enclave_id,
                                  &status,
                                  context,
                                  p_att_result_msg_body->secret.payload,
                                  p_att_result_msg_body->secret.payload_size,
                                  p_att_result_msg_body->secret.payload_tag);
            if ((SGX_SUCCESS != ret) || (SGX_SUCCESS != status))
            {
                fprintf(OUTPUT, "\nError, attestation result message secret "
```

```
                                "using SK based AESGCM failed in [%s]. ret = "
                                "0x%0x. status = 0x%0x",
                        __FUNCTION__, ret,
                        status);
                goto CLEANUP;
            }
        }
        else
        {
            fprintf(OUTPUT, "\nRemote attestation fail in [%s]", __FUNCTION__);
            goto CLEANUP;
        }

        //fprintf(OUTPUT, "\nSecret successfully received from server.");
        fprintf(OUTPUT, "\nRemote attestation success!");

        //三个过程共用，每次用完都释放
        ra_samp_request_header_t *p_setkeyreq = NULL;
        ra_samp_response_header_t *p_response = NULL;
        int recvlen = 0;
        //这时候开始与远程enclave建立加密信道

        //set key:g_sp_db就是协商出来的加密秘钥
        uint8_t token_with_key[32] = {0};
        fprintf(OUTPUT, "\nStart generate server key!");
        ret = generate_server_key(enclave_id, &status, token_with_key, 32);
        if ((SGX_SUCCESS != ret) || (SGX_SUCCESS != status))
        {
            fprintf(OUTPUT, "\nError ");
            goto CLEANUP;
        }
        //传key到server enclave中

        uint8_t out_data[16] = {'K', 'E', 'Y', 'S', 'E', 'T', 'S', 'U', 'C', 'C',
'E', 'S'};
        p_setkeyreq = (ra_samp_request_header_t
*)malloc(sizeof(ra_samp_request_header_t) + 32);
        memset(p_setkeyreq, 0 ,sizeof(ra_samp_request_header_t) + 32);
        p_setkeyreq->type = TYPE_RA_MSGSETKEY;
        p_setkeyreq->size = 32;
        if (memcpy_s(p_setkeyreq->body, 32, token_with_key, 32))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        fprintf(OUTPUT, "\nGenerate server aes key Success.");
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_setkeyreq, sizeof(ra_samp_request_header_t) +
32);
        SendToServer(sizeof(ra_samp_request_header_t) + 32);
        SAFE_FREE(p_setkeyreq);
        recvlen = RecvfromServer();
        //TODO:检查返回信息
        p_response = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t) + 32);
```

```
        if (memcpy_s(p_response, recvlen, recvbuf, recvlen))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        if ((p_response->type != TYPE_RA_MSGSETKEY) ||
            (memcmp(p_response->body, out_data, 16) != 0))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        else
        {
            fprintf(OUTPUT, "\nSuccess Set Server KEY ");
        }
        SAFE_FREE(p_response);

        //test remote aes service 加密
        uint8_t test_data[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15};
        uint8_t encryptdata[16] = {0};
        uint8_t decryptdata[16] = {0};

        p_setkeyreq = (ra_samp_request_header_t
*)malloc(sizeof(ra_samp_request_header_t) + 16);
        p_setkeyreq->type = TYPE_RA_MSGENC;
        p_setkeyreq->size = 16;

        if (memcpy_s(p_setkeyreq->body, 16, test_data, 16))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_setkeyreq, sizeof(ra_samp_request_header_t) +
16);
        SendToServer(sizeof(ra_samp_request_header_t) + 16);
        recvlen = RecvfromServer();
        //TODO:检查返回信息
        p_response = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t) + 32);

        if (memcpy_s(p_response, recvlen, recvbuf, recvlen))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        if ((p_response->type != TYPE_RA_MSGENC))
```

```
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        else
        {
            memcpy_s(encryptdata, 16, p_response->body, 16);
            fprintf(OUTPUT, "\nSuccess Encrypt");
            PRINT_BYTE_ARRAY(OUTPUT, test_data, 16);
            PRINT_BYTE_ARRAY(OUTPUT, encryptdata, 16);
        }
        SAFE_FREE(p_response);

        //解密

        p_setkeyreq = (ra_samp_request_header_t
*)malloc(sizeof(ra_samp_request_header_t) + 16);
        p_setkeyreq->type = TYPE_RA_MSGDEC;
        p_setkeyreq->size = 16;

        if (memcpy_s(p_setkeyreq->body, 16, encryptdata, 16))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_setkeyreq, sizeof(ra_samp_request_header_t) +
16);
        SendToServer(sizeof(ra_samp_request_header_t) + 16);
        recvlen = RecvfromServer();
        //检查返回信息
        p_response = (ra_samp_response_header_t
*)malloc(sizeof(ra_samp_response_header_t) + 32);

        if (memcpy_s(p_response, recvlen, recvbuf, recvlen))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }

        if ((p_response->type != TYPE_RA_MSGDEC))
        {
            fprintf(OUTPUT, "\nError: INTERNAL ERROR - memcpy failed in [%s].",
                    __FUNCTION__);
            ret = -1;
            goto CLEANUP;
        }
        else
        {
            memcpy_s(decryptdata, 16, p_response->body, 16);
            fprintf(OUTPUT, "\nSuccess Decrypt ");
            PRINT_BYTE_ARRAY(OUTPUT, encryptdata, 16);
```

```
            PRINT_BYTE_ARRAY(OUTPUT, decryptdata, 16);
        }
        SAFE_FREE(p_response);
    }

CLEANUP:
    // Clean-up
    // Need to close the RA key state.
    if (INT_MAX != context)
    {
        int ret_save = ret;
        ret = enclave_ra_close(enclave_id, &status, context);
        if (SGX_SUCCESS != ret || status)
        {
            ret = -1;
            fprintf(OUTPUT, "\nError, call enclave_ra_close fail [%s].",
                    __FUNCTION__);
        }
        else
        {
            // enclave_ra_close was successful, let's restore the value that
            // led us to this point in the code.
            ret = ret_save;
        }
        fprintf(OUTPUT, "\nCall enclave_ra_close success.");
    }

    sgx_destroy_enclave(enclave_id);

    ra_free_network_response_buffer(p_msg0_resp_full);
    ra_free_network_response_buffer(p_msg2_full);
    ra_free_network_response_buffer(p_att_result_msg_full);

    // p_msg3 is malloc'd by the untrusted KE library. App needs to free.
    SAFE_FREE(p_msg3);
    SAFE_FREE(p_msg3_full);
    SAFE_FREE(p_msg1_full);
    SAFE_FREE(p_msg0_full);
    printf("\nExit ...\n");
    return ret;
}
```

Enclave

enclave.edl

```
enclave {
    from "sgx_tkey_exchange.edl" import *;

    include "sgx_key_exchange.h"
    include "sgx_trts.h"

    trusted {
```

```
        public sgx_status_t enclave_init_ra(int b_pse,
                                            [out] sgx_ra_context_t *p_context);
        public sgx_status_t enclave_ra_close(sgx_ra_context_t context);
        public sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                                  [in,size=message_size] uint8_t*
message,
                                                  size_t message_size,
                                                  [in,size=mac_size] uint8_t*
mac,
                                                  size_t mac_size);
        public sgx_status_t put_secret_data(sgx_ra_context_t context,
                                            [in,size=secret_size] uint8_t*
p_secret,
                                            uint32_t secret_size,
                                            [in,count=16] uint8_t* gcm_mac);
        public sgx_status_t generate_server_key([in,size=32] uint8_t* out_data,
                                            uint32_t secret_size);
    };
};
```

enclave.cpp

```
#include <assert.h>
#include "isv_enclave_t.h"
#include "sgx_tkey_exchange.h"
#include "sgx_tcrypto.h"
#include "string.h"

// This is the public EC key of the SP. The corresponding private EC key is
// used by the SP to sign data used in the remote attestation SIGMA protocol
// to sign channel binding data in MSG2. A successful verification of the
// signature confirms the identity of the SP to the ISV app in remote
// attestation secure channel binding. The public EC key should be hardcoded in
// the enclave or delivered in a trustworthy manner. The use of a spoofed public
// EC key in the remote attestation with secure channel binding session may lead
// to a security compromise. Every different SP the enlcave communicates to
// must have a unique SP public key. Delivery of the SP public key is
// determined by the ISV. The TKE SIGMA protocl expects an Elliptical Curve key
// based on NIST P-256
static const sgx_ec256_public_t g_sp_pub_key = {
    {
        0x72, 0x12, 0x8a, 0x7a, 0x17, 0x52, 0x6e, 0xbf,
        0x85, 0xd0, 0x3a, 0x62, 0x37, 0x30, 0xae, 0xad,
        0x3e, 0x3d, 0xaa, 0xee, 0x9c, 0x60, 0x73, 0x1d,
        0xb0, 0x5b, 0xe8, 0x62, 0x1c, 0x4b, 0xeb, 0x38
    },
    {
        0xd4, 0x81, 0x40, 0xd9, 0x50, 0xe2, 0x57, 0x7b,
        0x26, 0xee, 0xb7, 0x41, 0xe7, 0xc6, 0x14, 0xe2,
        0x24, 0xb7, 0xbd, 0xc9, 0x03, 0xf2, 0x9a, 0x28,
        0xa8, 0x3c, 0xc8, 0x10, 0x11, 0x14, 0x5e, 0x06
    }

};
```

```
// Used to store the secret passed by the SP in the sample code. The
// size is forced to be 8 bytes. Expected value is
// 0x01,0x02,0x03,0x04,0x0x5,0x0x6,0x0x7
uint8_t g_secret[8] = {0};
sgx_ec_key_128bit_t sk_key;
#ifdef SUPPLIED_KEY_DERIVATION

#pragma message ("Supplied key derivation function is used.")

typedef struct _hash_buffer_t
{
    uint8_t counter[4];
    sgx_ec256_dh_shared_t shared_secret;
    uint8_t algorithm_id[4];
} hash_buffer_t;

const char ID_U[] = "SGXRAENCLAVE";
const char ID_V[] = "SGXRASERVER";

// Derive two keys from shared key and key id.
bool derive_key(
    const sgx_ec256_dh_shared_t *p_shared_key,
    uint8_t key_id,
    sgx_ec_key_128bit_t *first_derived_key,
    sgx_ec_key_128bit_t *second_derived_key)
{
    sgx_status_t sgx_ret = SGX_SUCCESS;
    hash_buffer_t hash_buffer;
    sgx_sha_state_handle_t sha_context;
    sgx_sha256_hash_t key_material;

    memset(&hash_buffer, 0, sizeof(hash_buffer_t));
    /* counter in big endian  */
    hash_buffer.counter[3] = key_id;

    /*convert from little endian to big endian */
    for (size_t i = 0; i < sizeof(sgx_ec256_dh_shared_t); i++)
    {
        hash_buffer.shared_secret.s[i] = p_shared_key->s[sizeof(p_shared_key-
>s)-1 - i];
    }

    sgx_ret = sgx_sha256_init(&sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&hash_buffer, sizeof(hash_buffer_t),
sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_U, sizeof(ID_U), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
```

```
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_update((uint8_t*)&ID_V, sizeof(ID_V), sha_context);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_get_hash(sha_context, &key_material);
    if (sgx_ret != SGX_SUCCESS)
    {
        sgx_sha256_close(sha_context);
        return false;
    }
    sgx_ret = sgx_sha256_close(sha_context);

    assert(sizeof(sgx_ec_key_128bit_t)* 2 == sizeof(sgx_sha256_hash_t));
    memcpy(first_derived_key, &key_material, sizeof(sgx_ec_key_128bit_t));
    memcpy(second_derived_key, (uint8_t*)&key_material +
sizeof(sgx_ec_key_128bit_t), sizeof(sgx_ec_key_128bit_t));

    // memset here can be optimized away by compiler, so please use memset_s on
    // windows for production code and similar functions on other OSes.
    memset(&key_material, 0, sizeof(sgx_sha256_hash_t));

    return true;
}

//isv defined key derivation function id
#define ISV_KDF_ID 2

typedef enum _derive_key_type_t
{
    DERIVE_KEY_SMK_SK = 0,
    DERIVE_KEY_MK_VK,
} derive_key_type_t;

sgx_status_t key_derivation(const sgx_ec256_dh_shared_t* shared_key,
    uint16_t kdf_id,
    sgx_ec_key_128bit_t* smk_key,
    sgx_ec_key_128bit_t* sk_key,
    sgx_ec_key_128bit_t* mk_key,
    sgx_ec_key_128bit_t* vk_key)
{
    bool derive_ret = false;

    if (NULL == shared_key)
    {
        return SGX_ERROR_INVALID_PARAMETER;
    }

    if (ISV_KDF_ID != kdf_id)
    {
        //fprintf(stderr, "\nError, key derivation id mismatch in [%s].",
__FUNCTION__);
        return SGX_ERROR_KDF_MISMATCH;
    }
```

```
    derive_ret = derive_key(shared_key, DERIVE_KEY_SMK_SK,
        smk_key, sk_key);
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }

    derive_ret = derive_key(shared_key, DERIVE_KEY_MK_VK,
        mk_key, vk_key);
    if (derive_ret != true)
    {
        //fprintf(stderr, "\nError, derive key fail in [%s].", __FUNCTION__);
        return SGX_ERROR_UNEXPECTED;
    }
    return SGX_SUCCESS;
}
#else
#pragma message ("Default key derivation function is used.")
#endif

// This ecall is a wrapper of sgx_ra_init to create the trusted
// KE exchange key context needed for the remote attestation
// SIGMA API's. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
//
// @param b_pse Indicates whether the ISV app is using the
//              platform services.
// @param p_context Pointer to the location where the returned
//                  key context is to be copied.
//
// @return Any error return from the create PSE session if b_pse
//         is true.
// @return Any error returned from the trusted key exchange API
//         for creating a key context.

sgx_status_t enclave_init_ra(
        int b_pse,
        sgx_ra_context_t *p_context) {
    // isv enclave call to trusted key exchange library.
    sgx_status_t ret;
#ifdef SUPPLIED_KEY_DERIVATION
    ret = sgx_ra_init_ex(&g_sp_pub_key, b_pse, key_derivation, p_context);
#else
    ret = sgx_ra_init(&g_sp_pub_key, b_pse, p_context);
#endif
    return ret;
}


// Closes the tKE key context used during the SIGMA key
// exchange.
//
// @param context The trusted KE library key context.
//
// @return Return value from the key context close API
```

```
sgx_status_t SGXAPI enclave_ra_close(
        sgx_ra_context_t context) {
sgx_status_t ret;
ret = sgx_ra_close(context);
return ret;
}


// Verify the mac sent in att_result_msg from the SP using the
// MK key. Input pointers aren't checked since the trusted stubs
// copy them into EPC memory.
//
//
// @param context The trusted KE library key context.
// @param p_message Pointer to the message used to produce MAC
// @param message_size Size in bytes of the message.
// @param p_mac Pointer to the MAC to compare to.
// @param mac_size Size in bytes of the MAC
//
// @return SGX_ERROR_INVALID_PARAMETER - MAC size is incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESCMAC function.
// @return SGX_ERROR_MAC_MISMATCH - MAC compare fails.

sgx_status_t verify_att_result_mac(sgx_ra_context_t context,
                                   uint8_t* p_message,
                                   size_t message_size,
                                   uint8_t* p_mac,
                                   size_t mac_size)
{
    sgx_status_t ret;
    sgx_ec_key_128bit_t mk_key;

    if(mac_size != sizeof(sgx_mac_t))
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }
    if(message_size > UINT32_MAX)
    {
        ret = SGX_ERROR_INVALID_PARAMETER;
        return ret;
    }

    do {
        uint8_t mac[SGX_CMAC_MAC_SIZE] = {0};

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_MK, &mk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }
        ret = sgx_rijndael128_cmac_msg(&mk_key,
                                       p_message,
                                       (uint32_t)message_size,
                                       &mac);
        if(SGX_SUCCESS != ret)
        {
```

```
            break;
        }
        if(0 == consttime_memequal(p_mac, mac, sizeof(mac)))
        {
            ret = SGX_ERROR_MAC_MISMATCH;
            break;
        }

    }
    while(0);

    return ret;
}


// Generate a secret information for the SP encrypted with SK.
// Input pointers aren't checked since the trusted stubs copy
// them into EPC memory.
//
// @param context The trusted KE library key context.
// @param p_secret Message containing the secret.
// @param secret_size Size in bytes of the secret message.
// @param p_gcm_mac The pointer the the AESGCM MAC for the
//                 message.
//
// @return SGX_ERROR_INVALID_PARAMETER - secret size if
//         incorrect.
// @return Any error produced by tKE  API to get SK key.
// @return Any error produced by the AESGCM function.
// @return SGX_ERROR_UNEXPECTED - the secret doesn't match the
//         expected value.

sgx_status_t put_secret_data(
    sgx_ra_context_t context,
    uint8_t *p_secret,
    uint32_t secret_size,
    uint8_t *p_gcm_mac)
{
    sgx_status_t ret = SGX_SUCCESS;
    sgx_ec_key_128bit_t sk_key;

    do {
        if(secret_size != 8)
        {
            ret = SGX_ERROR_INVALID_PARAMETER;
            break;
        }

        ret = sgx_ra_get_keys(context, SGX_RA_KEY_SK, &sk_key);
        if(SGX_SUCCESS != ret)
        {
            break;
        }

        uint8_t aes_gcm_iv[12] = {0};
        ret = sgx_rijndael128GCM_decrypt(&sk_key,
                                         p_secret,
                                         secret_size,
```

```
                                        &g_secret[0],
                                        &aes_gcm_iv[0],
                                        12,
                                        NULL,
                                        0,
                                        (const sgx_aes_gcm_128bit_tag_t *)
                                           (p_gcm_mac));

        uint32_t i;
        bool secret_match = true;
        for(i=0;i<secret_size;i++)
        {
            if(g_secret[i] != i)
            {
                secret_match = false;
            }
        }

        if(!secret_match)
        {
            ret = SGX_ERROR_UNEXPECTED;
        }

        // Once the server has the shared secret, it should be sealed to
        // persistent storage for future use. This will prevents having to
        // perform remote attestation until the secret goes stale. Once the
        // enclave is created again, the secret can be unsealed.
    } while(0);
    return ret;
}

sgx_status_t generate_server_key(
    uint8_t *token_with_key,
    uint32_t secret_size)
{
    sgx_status_t ret = SGX_SUCCESS;
    if(secret_size != 32)//加密数据
    {
        ret = SGX_ERROR_UNEXPECTED;
        return ret;
    }
    uint8_t token[32] = {0};
    ret = sgx_read_rand(token, 32);//随机生成秘钥
    uint8_t aes_gcm_iv[12] = {0};
    sgx_aes_gcm_128bit_tag_t c_gcm_mac;
    //token用于server识别这个enclave已经有了sk_key,原理就是用client key加密以后server解密还是5到12

    do {

        ret = sgx_rijndael128GCM_encrypt(&sk_key,
                                        &token[0],
                                        secret_size,
                                        token_with_key,
                                        &aes_gcm_iv[0],
                                        12,
                                        NULL,
                                        0,
```

```
                                        &c_gcm_mac);
    } while(0);

    for(int i=0;i<16;i++)
    {
        token[i] = i+5;
    }
    return ret;
}
```

Service Provider

network_ra_client.cpp

提供socket通信服务

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include "network_ra.h"
#include "service_provider.h"
//add
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

char sendbuf[BUFSIZ];  //数据传送的缓冲区
char recvbuf[BUFSIZ];
int  client_sockfd;//客户端套接字



void PRINT_BYTE_ARRAY(
    FILE *file, void *mem, uint32_t len)
{
    if(!mem || !len)
    {
        fprintf(file, "\n( null )\n");
        return;
    }
    uint8_t *array = (uint8_t *)mem;
    fprintf(file, "%u bytes:\n{\n", len);
    uint32_t i = 0;
    for(i = 0; i < len - 1; i++)
    {
        fprintf(file, "0x%x, ", array[i]);
        if(i % 8 == 7) fprintf(file, "\n");
    }
    fprintf(file, "0x%x ", array[i]);
    fprintf(file, "\n}\n");
}
```

```
// Used to send requests to the service provider sample.  It
// simulates network communication between the ISV app and the
// ISV service provider.  This would be modified in a real
// product to use the proper IP communication.
//
// @param server_url String name of the server URL
// @param p_req Pointer to the message to be sent.
// @param p_resp Pointer to a pointer of the response message.

// @return int
// 修改成真正的网络通讯
int ra_network_send_receive(const char *server_url,
    const ra_samp_request_header_t *p_req,
    ra_samp_response_header_t **p_resp)
{
    FILE* OUTPUT = stdout;
    int ret = 0;
    int len = 0;
    int msg2len = 0;

    if((NULL == server_url) ||
        (NULL == p_req) ||
        (NULL == p_resp))
    {
        return -1;
    }
    switch(p_req->type)
    {

    case TYPE_RA_MSG0:
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_req, sizeof(ra_samp_request_header_t)+p_req-
>size);
        len = SendToServer(sizeof(ra_samp_request_header_t)+p_req->size);
        sleep(1);//等待起作用
        if (0 == len)
        {
            fprintf(stderr, "\nError,Send MSG0 fail [%s].",
                __FUNCTION__);
        }
        break;

    case TYPE_RA_MSG1:
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_req, sizeof(ra_samp_request_header_t)+p_req-
>size);
        ret = SendToServer(sizeof(ra_samp_request_header_t)+p_req->size);
        fprintf(stdout, "\nSend MSG1 To Server [%s].",__FUNCTION__);
        ret = RecvfromServer();
        msg2len = sizeof(ra_samp_response_header_t)+sizeof(sample_ra_msg2_t);

        memcpy_s(*p_resp, msg2len, recvbuf, ret);
        break;

    case TYPE_RA_MSG3:
        memset(sendbuf, 0, BUFSIZ);
        memcpy_s(sendbuf, BUFSIZ, p_req, sizeof(ra_samp_request_header_t)+p_req-
>size);
```

```
        ret = SendToServer(sizeof(ra_samp_request_header_t)+p_req->size);
        ret = RecvfromServer();
        memcpy_s(*p_resp, ret, recvbuf, ret);
        fprintf(stderr, "\nMsg3 ret = %d [%s].", ret);
        PRINT_BYTE_ARRAY(OUTPUT, *p_resp, ret);
        break;

    default:
        ret = -1;
        fprintf(stderr, "\nError, unknown ra message type. Type = %d [%s].",
            p_req->type, __FUNCTION__);
        break;
    }

    return ret;
}
int client(const char ip[16],int port)
{
    int len;
    struct sockaddr_in remote_addr; //服务器端网络地址结构体
    memset(&remote_addr,0,sizeof(remote_addr)); //数据初始化--清零
    remote_addr.sin_family=AF_INET; //设置为IP通信
    remote_addr.sin_addr.s_addr=inet_addr(ip);//服务器IP地址
    remote_addr.sin_port=htons(port); //服务器端口号

    /*创建客户端套接字--IPv4协议，面向连接通信，TCP协议*/
    if((client_sockfd=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        perror("socket");
        return 1;
    }

    /*将套接字绑定到服务器的网络地址上*/
    if(connect(client_sockfd,(struct sockaddr *)&remote_addr,sizeof(struct
sockaddr))<0)
    {
        perror("connect");
        return 1;
    }
    printf("connected to server\n");
    return 0;
}

int SendToServer(int len)
{
    len=send(client_sockfd, sendbuf, len, 0);//发送
}

int RecvfromServer()
{
    /*接收服务端的数据*/
    int len = 0;
    len=recv(client_sockfd,recvbuf,BUFSIZ,0);
    if (len > 0 )
        recvbuf[len] = 0;
    return len;
}
```

```
int Cleanupsocket()
{
    close(client_sockfd);
    return 0;
}

// Used to free the response messages.  In the sample code, the
// response messages are allocated by the SP code.
//
//
// @param resp Pointer to the response buffer to be freed.

void ra_free_network_response_buffer(ra_samp_response_header_t *resp)
{
    if(resp!=NULL)
    {
        free(resp);
    }
}
```

###