Data Serialization
&
Remote
Procedure
Calls

Road Map

1 Data Serialization Concepts

A technique to reform the data (Flattening) before sending it over the network

2 Data Structure STREAMS

A nice Data structure used to facilitate Serialization of Data

3 Data Serialization in Action

Code to Serialize and De-serialize the data

4 Remote Procedure Calls

How to implement RPCs

5 Application State Synchronization

Clone the application Memory state On Remote Machine

6 Check pointing

Save the and restore the Application Memory State to/from the Disk

Warning : lot of Coding and be Good in C !!

Agenda

- What is Serialization/(De)Serialization ?
- Why Data need to be (De)Serialized ?
- Data Structure : Streams
- (De)Serializing C Nested Structures
- Mini Exercise
- Summary

Pre-requisite : C and pointers, and be good at it !

Data (De)Serialization Concept

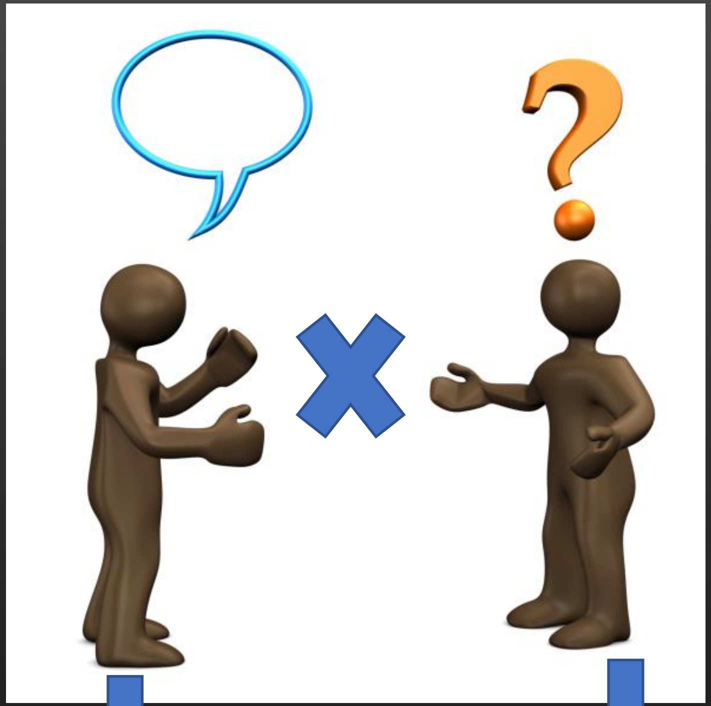
Data Serialization and De-Serialization

Introduction

- Data Serialization is a mechanism to transform the program's internal data structures into a form that could be sent to remote machine over the network
- Data Serialization is performed by sending machine
- Data DeSerialization is a mechanism to transform the serialized data and convert or reconstruct the internal data structure
- Data DeSerialization is performed by receiving process
- Data (De)Serialization helps in making data exchange between processes running on heterogeneous machines independent of underlying OS, Compiler, programming language, Hardware etc

Data Serialization and De-Serialization

Analogy



understand French only

understand Chinese only

understand French only

translator, mediator Understands French and Chinese Both

understand Chinese only

Here, Mediator is a (De)Serializer who performs Serialization (French->chinese) and deserialization (Chinese->French)

Data Serialization and De-Serialization

Why we need Data (De)Serialization ?

- Two machines wanting to exchange data over the network may not be identical and vary from each other wrt various parameters such as
  - Hardware Architecture
    - 32bit address space, 64 bit bit address space
  - Software version Or different compiler
    - sizeof int is 32 bit on one machine, could be 64 bit on another machine
  - Endianness
    - The way machine stores the data in memory
- Thus, there is a need that two machines need to communicate with each other in a agreed format
  - Sending machine needs to encode the data in the standard pre-defined format
  - Receiving machine should decide the data as per the same standard format
  - For ex : If the standard says, integers should be encoded as 32 bit values, then even 64 bit machines on which *sizeof(int)* is 8 bytes, must encode the integer data in 32 bits

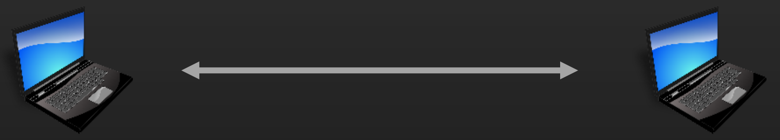
Data Serialization and De-Serialization

Why we need Data (De)Serialization ?

- Let us Try to understand what are the problems and difficulties we shall face when processes exchanges data over the network and communicating processes are running on machines of different flavors altogether

For example : size of integer on sending machine is 32 bit, whereas size of integer on receiving machine is 64 bits

Sending machine (32 bit)	Receiving machine (64 bit)
struct person_t { char name[30]; int age; int weight; };	struct person_t { char name[30]; int age; int weight; };
Size = 38 bytes	Size = 46



32 bit

64 bit

Mem Layout 1 :

<name 30 B>	<age 4 B>	<weight 4 B>

Actual msg sent out on wire by sending machine

Mem Layout 2 :

<name 30 B>	<age 8 B>	<weight 8 B>

Data Serialization and De-Serialization

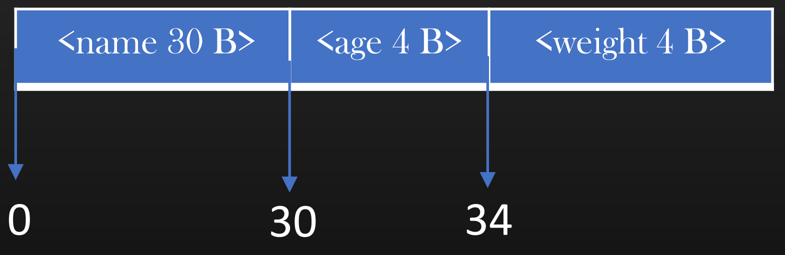
Why we need Data (De)Serialization ?

Sending machine (32 bit)	Receiving machine (64 bit)

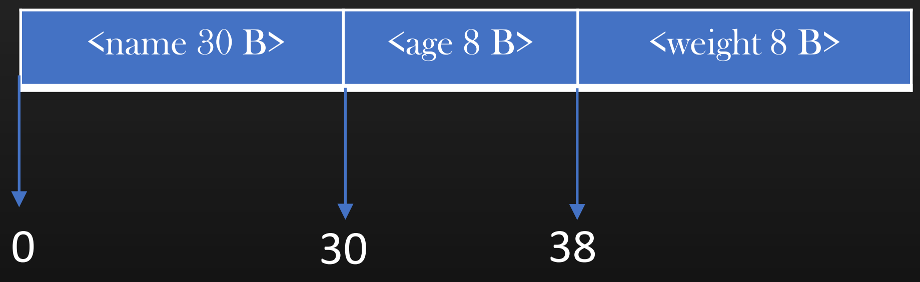
struct person_t {
char
name[30];
int age;
int weight;
};
	struct person_t {
	char
	name[30];
	int age;
	int weight;
	};

Size – 38 bytes	Size = 46

Mem Layout 1 :
(Sending machine)



Mem Layout 2 :
(Receiving machine)



ptr->age
Sending machine : reads 4 bytes starting from 30th offset
Receiving machine : reads 8 bytes starting from 30th offset

ptr->weight
Sending machine : reads 4 bytes starting from 34th offset
Receiving machine : reads 8 bytes starting from 38th offset

Data Serialization and De-Serialization

Why we need Data (De)Serialization ?

Sending machine (32 bit)	Receiving machine (64 bit)

struct person_t {
char name[30];
int age;
int weight;
};
	struct person_t {
	char name[30];
	int age;
	int weight;
	};

Size – 38 bytes	Size = 46

msg1

Mem Layout 1 :
(Sending machine)

<name 30 B>	<age 4 B>	<weight 4 B>

Mem Layout 2 :
(Receiving machine)

<name 30 B>	<age 8 B>	<weight 8 B>

msg2

Receiving machine would try to read this message as per its definition of Structure (memory layout 2), it end up in reading garbage

Problem on Receiving machine :

```
struct person_t *per = (struct person_t *)buffer;
per->name;
per->age;      /*Corruption : Reading age as 8Bs, instead of 4Bs*/
per->weight;  /*Corruption : Reading 38th byte instead of 34th byte*/
```

Data Serialization and De-Serialization

Why we need Data (De)Serialization ?

- Thus we discussed how different sizes of a primitive data type could cause problems in data exchange between two machines

- There are other barriers in data exchange between communication machines, but their detailed discussion is not worth for the sake of briefness of this course

- Pls Google – Endianness of machines and padding bytes to understand how these factors also affect the data exchange between heterogeneous machines

- As long as two machines interprets the same C structure definition different (different memory layout) , they simply cannot exchange data without (De)Serialization

Why we need Data (De)Serialization ?

2. Compiler Optimization Settings

   For example : sending machine's compiler is 4 byte aligned, whereas alignment optimization is disabled on recieving machine

Sending machine (32 bit)	**Receiving machine (32 bit)**

struct person_t {
char name[30];
< 2 padding byte s>
int age;
int weight;
};
	struct person_t {
	char name[30];
	int age;
	int weight;
	};

Size = 40 bytes	Size = 38

Mem Layout 1 :
(Sending machine)

<name 30 B>	2B	<age 4 B>	<weight 4 B>

Mem Layout 2 :
(Receiving machine)

<name 30 B>	<age 4 B>	<weight 4 B>

* **If you do not understand structure padding, pls google and comeback*

Data Serialization and De-Serialization

Why we need Data (De)Serialization ?

2. Compiler Optimization Settings

For example : Receiving machine's compiler is 4 byte aligned, whereas alignment optimization is disabled on receiving machine

Msg1 :
(Sending machine)

<name 30 B>	2B	<age 4 B>	<weight 4 B>

Msg2:
(Receiving machine)

<name 30 B>	<age 4 B>	<weight 4 B>

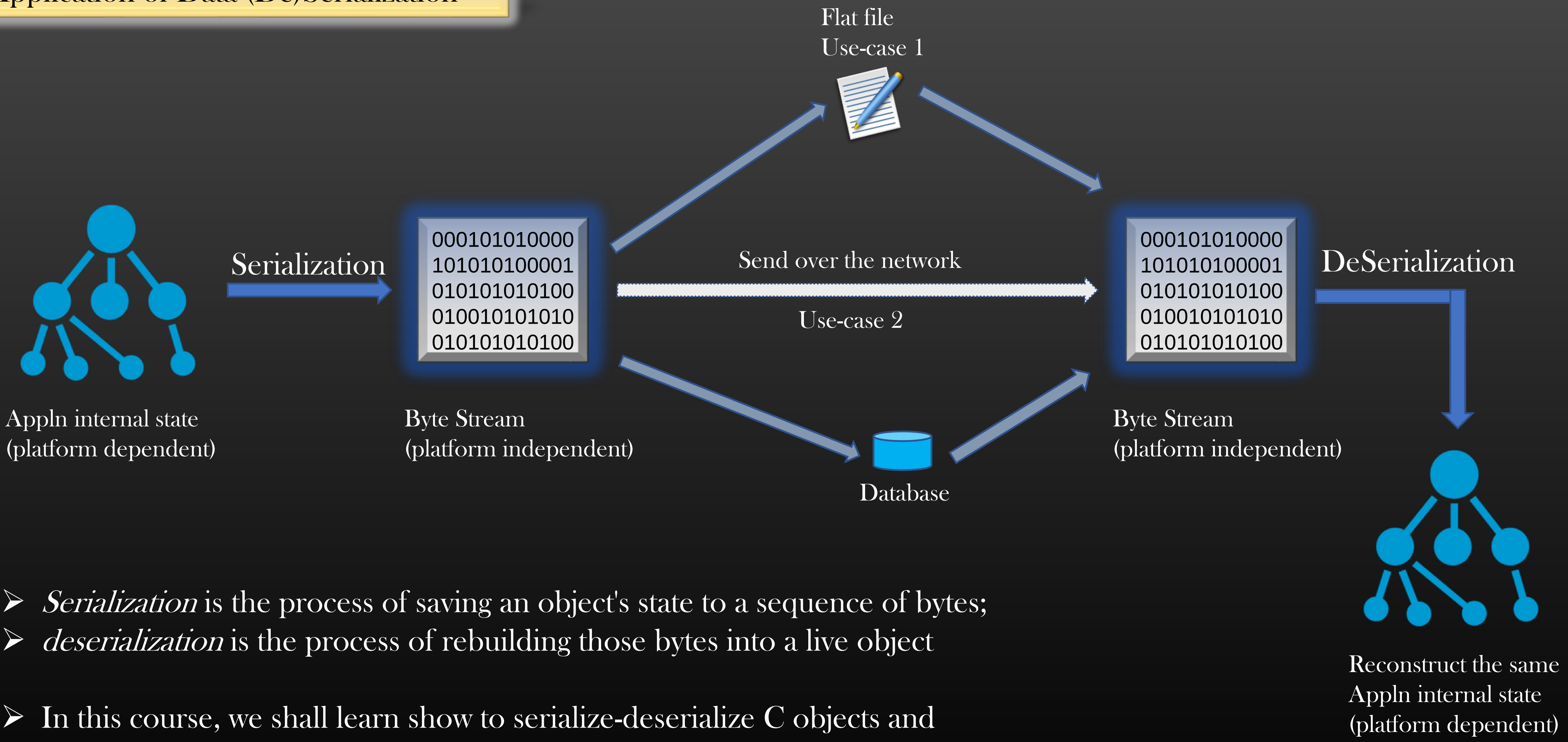
Problem on Receiving machine :

```
struct person_t *per = (struct person_t *)buffer;
per->name;
per->age;       /*Corruption : Reading age from 30th byte instead of
                     32th byte */
```

* *If you do not understand structure padding, pls google and comeback*

Data Serialization and De-Serialization

Application of Data (De)Serialization



- *Serialization* is the process of saving an object's state to a sequence of bytes;
- *deserialization* is the process of rebuilding those bytes into a live object

- In this course, we shall learn show to serialize-deserialize C objects and
  - Use-case 1 and 2

What are big-Endian and Little-Endian machines

3. Endianness

For example : sending machine is little endian machine and receiving machine is big endian machine

Back ground :

Machines are categorized into two categories depending on their Hardware architecture :

1. Big Endian Machines
2. Little Endian Machines

Let US take a short tour to understand what Big endian and Little endian machines are !

What are big-Endian and Little-Endian machines

- Endianness refers to the sequential order in which bytes of data are stored in computer memory
- Best understood with the help of example
- Let's us you have a number : 3066773545

Binary representation of this number is : (10110110 11001011 01000000 00101001)

Address ->	2000	2001	2002	2003
	10110110	11001011	01000000	00101001
Byte no ->	3	2	1	0

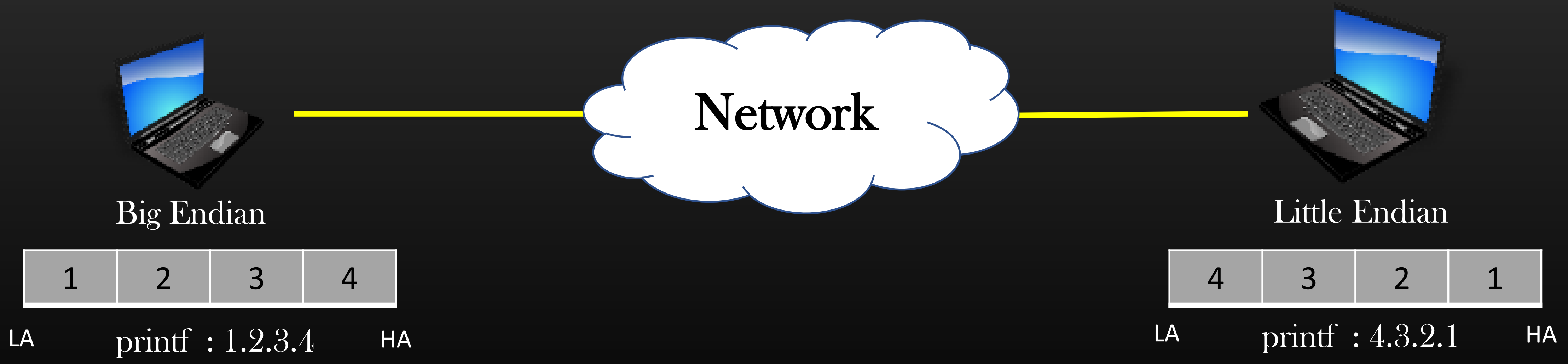
This is big endian representation
(Least significant byte at higher address)
Network Byte Order

Address ->	2000	2001	2002	2003
	00101001	01000000	11001011	10110110
Byte no ->	3	2	1	0

This is little endian representation
(Least significant byte at lower address)
Host Byte Order

What are big-Endian and Little-Endian machines

- Machines of different endianness may need to communicate over the network
- IETF (Internet Engineering Task Force) has standardized the Data flowing over the network MUST be in Network Byte order



Endian-ness of Machines

What are big-Endian and Little-Endian machines

- Interview Question :

Write a C program which determines whether your machine is big endian or little endian ?

0	1

Big-Endian

1	0

Little-Endian

```
/* return 0 – Big endian, 1 for Little endian*/

int
machine_endianness_type() {

        unsigned short int a = 1;
        char ist_byte = *((char *)&a);
        if ( ist_byte == 0 )
                return 0;
        else if ( ist_byte == 1 )
                return 1;
}
```

Summary

- Thus, we discussed the challenges one can come across when machines over the network wants to communicate
- Data serialization and DeSerialization is a technique to facilitate data exchange between heterogeneous machines using standard pre-defined format
- Data serialization and DeSerialization (DDe) removes the "language" barrier between communicating machines
- In this course, we shall learn how to serialize the C structures on sending machine and De-Serialize the same on receiving machine
- In this course, we shall assume, that two communicating machines are almost identical – that makes our Serialization/Deserialization routines simple to write and understand.
- Lets begin understanding the Serialization and Deserialization of C structures

Data Serialization and Data Deserialization

Data Serialization

Data DeSerialization

Building block of Various System programming Concepts such as :

- RPC
- State Synchronization
- Check pointing the application state

Data Serialization

Let us Learn the concepts of Serialization . . .

1. Simple C structures
2. C structures with nested C sub-structures
3. C structures with pointer members

Data Serialization

Let us Learn the concepts of Serialization . . .

1. Simple C structures

2. C structures with nested C sub-structures

3. C structures with pointer members

Data Serialization and Data Deserialization

Data Serialization

Simple Structures :

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

2 B of extra padding

Size = 40 B

<name 30 B>	2B	<age 4 B>	<weight 4 B>

Memory foot print

Serialize(struct person_t *per)

<name 30 B>	<age 4 B>	<weight 4 B>

Serialized Version
(get rid of padding bytes)

Data Serialization and Data Deserialization

Data Serialization

Nested Structures :

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation occ;
        int weight;
};

struct occupation{
        char dept_name[30];
        int emp_code;
};
```

<name 30 B>	2B	<age 4 B>	<dept_name 30B>	2B	<emp_code 4B>	<weight 4B>

Memory foot print

Serialized Version

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

Data Serialization and Data Deserialization

Data Serialization

Structures with pointer members

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation *occ;
        int weight;
};

struct occupation{
        char dept_name[30];
        int emp_code;
};
```

<dept name 30 B>	2B	<emp_code 4 B>

<name 30 B>	2B	<age 4 B>	0xdf00ff 4B	<weight 4B>

Memory foot print

Serialized Version

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

Flattened structure

Data Serialization and Data Deserialization

Data Serialization – Encode NuLL as Sentinel

Structures with pointer members = NULL

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation *occ;  /*= NULL*/
        int weight;
};
```

```
struct occupation{
        char dept_name[30];
        int emp_code;
};
```

<name 30 B>	2B	<age 4 B>	NULL	<weight 4B>

Actual object Memory foot print

Serialized Version

<name 30 B>	<age 4 B>	0xFFFFFFFF	<weight 4B>

Serialized object

- We have chosen 0xFFFFFFFF as a sentinel to represent NULL
- During Deserialization of Serialized object, we need to give special treatment to this sentinel

During Deserialization, when we encounter the sentinel value in the serialized object, we would know that occ field was set to NULL in the original object

Data DeSerialization

Let us Learn the concepts of Deserialization . . .

1. Simple C structures
2. C structures with nested C sub-structures
3. C structures with pointer members

Data Serialization and Data Deserialization

Data DeSerialization

Simple Structures :

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

2 B of extra padding

<name 30 B>	<age 4 B>	<weight 4 B>

Serialized Version

DeSerialize(ser_data *)

```
struct person_t {
        char name[30];  // "Abhishek"
        int age;        //   31
        int weight;     //   72
};
```

<name 30 B>	2B	<age 4 B>	<weight 4 B>

Memory foot print depending on Receiving machine Hw and compiler

De-Serialization is a process of reconstructing the application internal data structure from flat Serialized data

Data Serialization and Data Deserialization

Data DeSerialization

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

Serialized Version

DeSerialize(ser_data *)

Nested Structures :

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation occ;
        int weight;
};

struct occupation{
    char dept_name[30];
    int employee_code;
}occ;
```

```
struct person_t {
        char name[30];              // "Abhishek"
        int age;                    // 31
        occ.dept_name;              // "CS
        <padding bytes>
        occ.employee_code;          // 52437
        int weight;                 // 72
};
```

<name 30 B>	2B	<age 4 B>	<dept_name 30B>	2B	<emp_code 4B>	<weight 4B>

Memory foot print depending on Receiving machine Hw and compiler

Note : You don't explicitly insert the padding bytes, compiler inserts the padding bytes when you malloc the object or declare local variable on stack

Data Serialization and Data Deserialization

Data DeSerialization

Structures with pointer members

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation *occ;
        int weight;
};

struct occupation{
        char dept_name[30];
        int emp_code;
};
```

<name 30 B>	2B	<age 4 B>	0x4300dff 4B	<weight 4B>

<dept name 30 B>	2B	<emp_code 4 B>

Hierarchy is restored !!

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

Serialized Version

DeSerialize(ser_data *)

```
struct person_t {
        char name[30];           // "Abhishek"
        int age;                 // 31
        struct occupation *occ;          // 0x4300dff
        int weight;                              // 71
};
```

located at memory : 0x4300dff

```
struct occupation{
        char dept_name[30];              // "CSE"
        int emp_code;                            // 52437
};
```

Data DeSerialization – Handling Sentinel Value

Structures with pointer members

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation *occ;
        int weight;
};

struct occupation{
        char dept_name[30];
        int emp_code;
};
```

<name 30 B>	<age 4 B>	0xFFFFFFFF	<weight 4B>

Serialized Version

DeSerialize(ser_data *)

<name 30 B>	2B	<age 4 B>	NULL	<weight 4B>

Reconstructed object

During Deserialization , Sentinel value in the serialized object is transformed to NULL in the new Constructed object

Data Serialization and Data Deserialization

Example

```
struct person{
        char name[30];
        int age;
        struct occupation occ;
        struct person *mentor;
        struct person *boss;
};

struct occupation {
        char designation[30];
        int salary;
};
```

```
struct person *ptr = calloc(1, sizeof(struct person));
strcpy(ptr->name, “Abhishek”);
ptr->age = 31;
strcpy(ptr->occ.designation, “SE”);
ptr->occ.salary = 100000;
ptr->mentor = NULL;
ptr->boss = calloc(1, sizeof(struct person));
strcpy(ptr->boss->name, “Ram”);
ptr->boss->age = 45;
strcpy(ptr->boss->occ.designation, “MGR”);
ptr->boss->salary = 500000
ptr->boss->mentor = NULL;
ptr->boss->boss = NULL;
```

Actual object		Serialized object
NULL	⟺	0xFFFFFFFF

Malloc’d Object in Memory

<Abhishek 30B>	2B	<31 4B>	<SE 30B>	<100000 4B>	NULL	0xff00ddee

<Ram 30B>	2B	<45 4 B>	<MGR 30B>	<500000 4B>	NULL	NULL

serialization

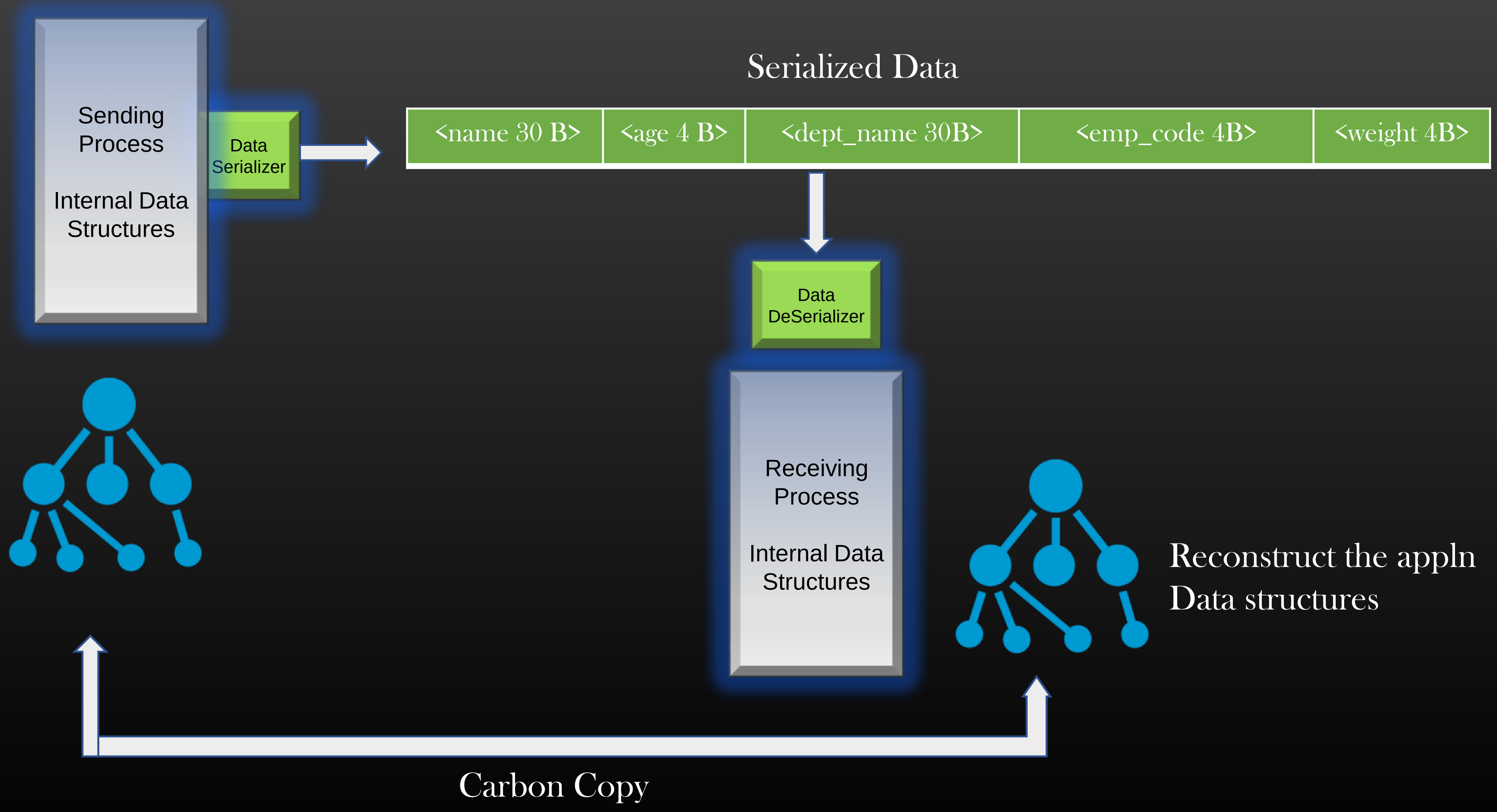
Deserialization

Serialized Object

Abhishek 30B	31 4B	SE 30B	100000 4B	0xFFFFFFFF 4B	Ram 30B	45 4B	MGR 30B	500000 4B	0xFFFFFFFF 4B	0xFFFFFFFF 4B

Data Serialization and Data Deserialization

In a Nut Shell



Data Serialization and Data Deserialization

APIs

Given Structure S

Serializing routine

void serialize_S
(S *obj, ser_buff_t *b)

Input — output

Serialization routine for S

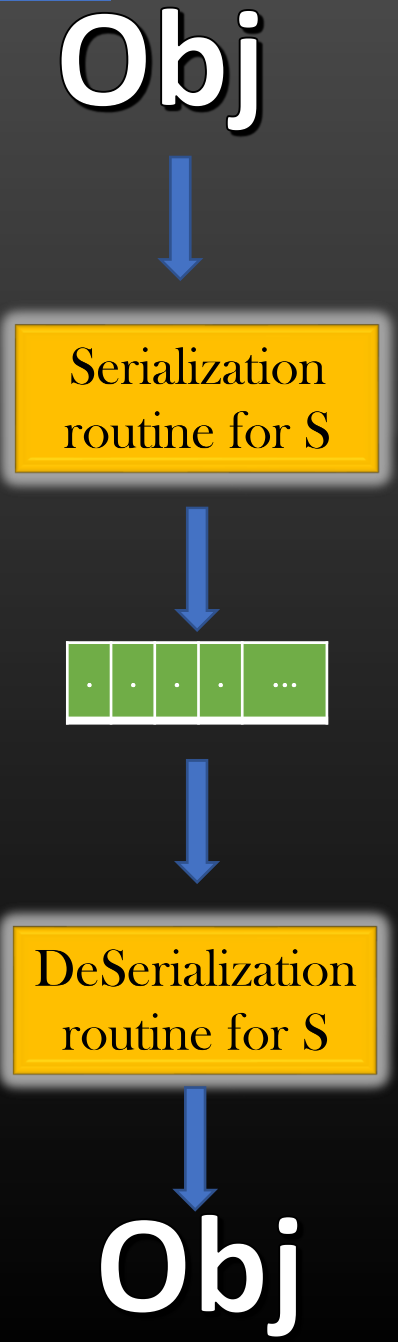
DeSerializing routine

S* de_serialize_S
(ser_buff_t *b)

Output

Input

DeSerialization routine for S



APIs - Example

Structure Name	Serialization routine	Deserialization routine
person_t	void serialize_person_t (person_t *obj, ser_buff_t *b);	person_t * de_serialize_person_t (ser_buff_t *b);
company_t	void serialize_company_t (company_t *obj, ser_buff_t *b);	company_t * de_serialize_company_t (ser_buff_t *b);

Data Serialization and Data Deserialization

Flowchart

Internal Data → Serializer → Serialized Data

↓

Send over the Network

↓

DeSerializer

↓

Internal Data

Data Serialization

Data DeSerialization

RPC

Summary

- We understood why we need to do Data serialization and Data deserialization
- For every structure, we need to do perform its Serialization on sending process and deserialization on receiving process
- Serialized data is a flat structure , contains no pointers, padding bytes or any hierarchical meta data
- Next, we shall learn how to write routines to serialize and de-serialze C structures. These concepts are not limited to C, applicable to any programming language
- So, you are learning concepts which are programming language agonistic
- To make the Serialization and Deserialization easy, we use simple data structure called STREAM , also called Serialized buffer
- Before jumping into actual process of (De)Serializing the C structures, lets discuss about STREAMs

Data (De)Serialization Concept

Thank you ☺☺

Data (De)Serialization Exercises

Data Serialization and De-Serialization

Q1. Serialize the linked list, send the serialized Linked list to remote machine using UDP sockets.
Receiving machine on receiving machine, de-serialize the linked list and send back sum of integers Of a Linked-List.

```
typedef struct list_node_{
        int data;
        struct list_node_ *next;
} list_node_t ;

typedef struct list_{
        list_node_t *head;
} list_t;
```

Q2. Construct the Binary search tree and print the Inorder, Pre-order and post-order traversal of a tree on local node. Now, Serialize the tree and send the Serialized data to remote machine. Remote machine should deserialize the tree and print the Inorder, Pre-order and post-order traversal of a reconstructed tree.
Hint : You will see that traversal output of remote machine may not be same as those which was printed on sending machine

Q3. Consider the structure settings as below :

```
typedef struct computer_ {

        int n_cores;
        int ram_size;
        int price;
        char *glue_ptr;
} computer_t;
```

```
typedef struct dealer_ {

        char dealer_name[32];
        char dealer_address[32];
        char glue;
} dealer_t;
```

```
/*Object creation*/

computer_t *comp = calloc(1, sizeof(computer_t));
comp->n_cores = 3;
comp->ram_size = 2;
comp->price = 50000;

dealer_t *dealer = calloc(1, sizeof(dealer_t));
strcpy(dealer->dealer_name, “DELL”);
strcpy(dealer->dealer_address, “Bangalore”);

/*Binding*/
Comp->glue_ptr = &dealer->glue;
```

→ *Print the object comp along with the dealer details*

→ *Serialize the object comp. See where is the problem.*
Could you do something about this problem ?
What is your conclusion ?

STREAMS

Serialized buffer

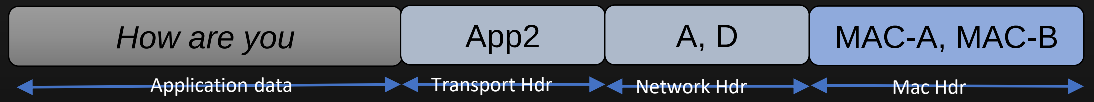
Serialized Buffer/Streams

- Data structure : Stream
- It helps in copying the bytes into a buffer, as well as reading the bytes from the buffer very easy and handy
- It resembles to type writer – start with the next line when there is no space left in the current line
- It is a flexible buffer which grows automatically when run out of space
- This Data Structure is useful in situation when we need to collect/append data one after the other incrementally. For example, appending OSI layer headers in the packet one after the other (L1,L2,L3,L4) headers

Serialized Buffer/Streams

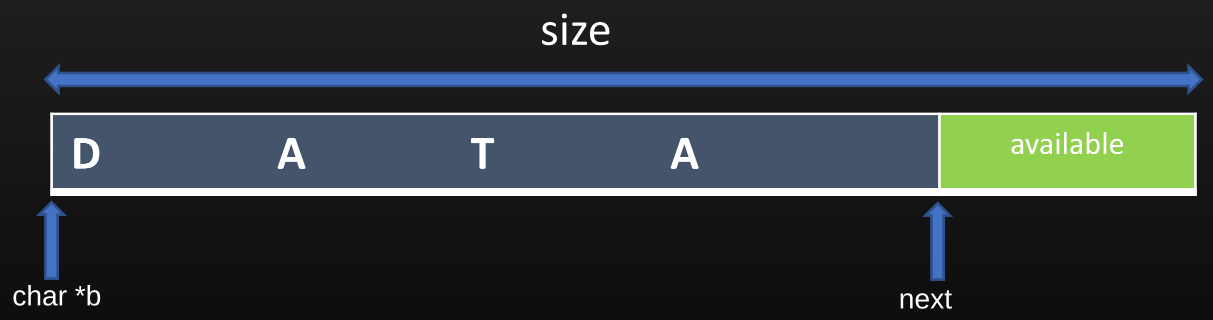
- Data structure : Stream
- It helps in copying the bytes into a buffer, as well as reading the bytes from the buffer very easy and handy
- It resembles to type writer – start with the next line when there is no space left in the current line
- It is a flexible buffer which grows automatically when run out of space
- This Data Structure is useful in situation when we need to collect/append data one after the other incrementally. For example, appending OSI layer headers in the packet one after the other (L1,L2,L3,L4) headers

Network packet ->



Structure definition

```
typedef struct serialized_buffer{

    char *b;    /*Always point to the start of the memory buffer which actually stores the data*/

    int size;    /*size of Serialized buffer*/

    int next;    /*byte position in serialized buffer where next data item will be written into Or read from*/

} ser_buff_t;
```



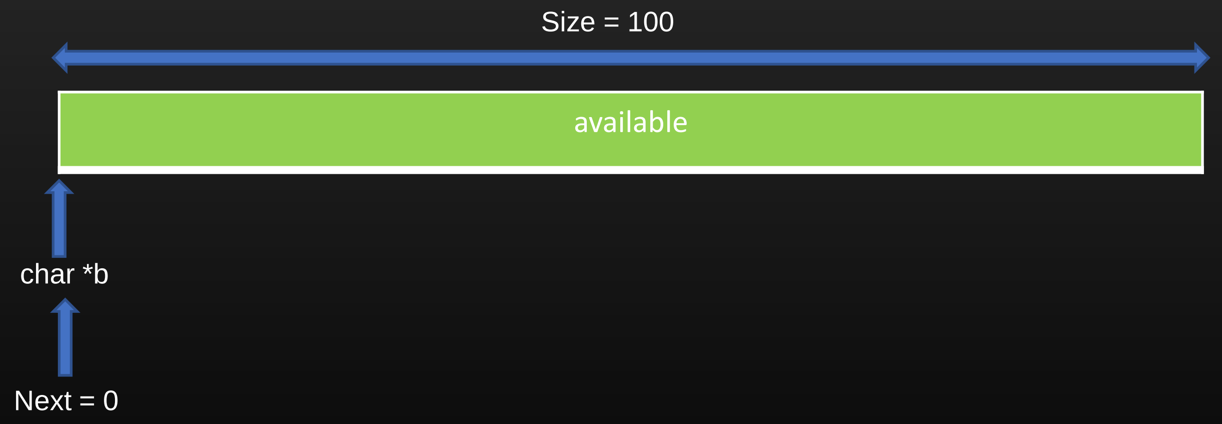
Initialization

```
void
init_serialized_buffer (ser_buff_t **b){

    (*b) = (ser_buff_t *)calloc(1, sizeof(ser_buff_t));

    (*b)->b = calloc(1, SERIALIZE_BUFFER_DEFAULT_SIZE); /*const , say 100*/

    (*b)->size = SERIALIZE_BUFFER_DEFAULT_SIZE;

    (*b)->next = 0;
}
```

Usage :

```
ser_buff_t *stream;
init_serialized_buffer(&stream);
```

```
typedef struct serialized_buffer{

    char *b;
    int size;
    int next;
} ser_buff_t;
```

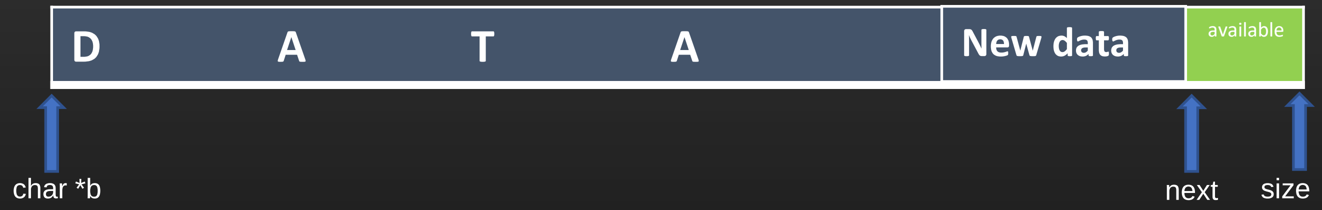


Copying data into Serialized buffer

```
void serialize_data (ser_buff_t *buff, char *data, int nbytes){

    int available_size = buff->size - buff->next;
    char isResize = 0;

    while(available_size < nbytes){
        buff->size = buff->size * 2;
        available_size = buff->size - buff->next;
        isResize = 1;
    }

    if(isResize == 0){
        memcpy((char *)buff->b + buff->next, data, nbytes);
        buff->next += nbytes;
        return;
    }

    // resize of the buffer
    buff->b = realloc(buff->b, buff->size);
    memcpy((char *)buff->b + buff->next, data, nbytes);
    buff->next += nbytes;
    return;
}
```

Copying data into Serialized buffer

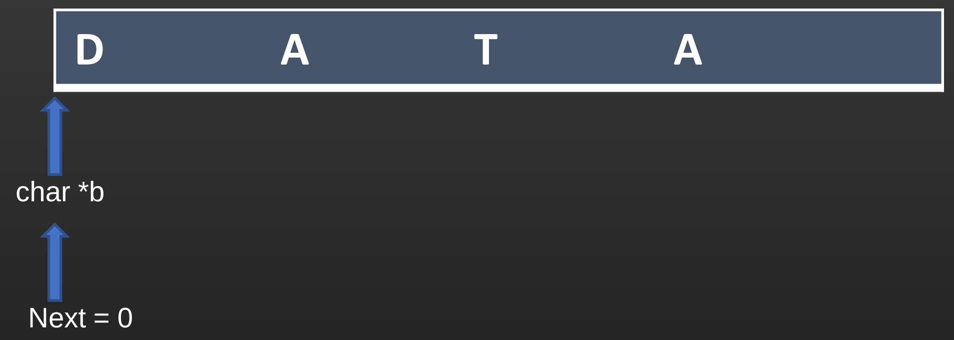
```
void serialize_data (ser_buff_t *buff, char *data, int nbytes){

    int available_size = buff->size - buff->next;
    char isResize = 0;

    while(available_size < nbytes){
        buff->size = buff->size * 2;
        available_size = buff->size - buff->next;
        isResize = 1;
    }

    if(isResize == 0){
        memcpy((char *)buff->b + buff->next, data, nbytes);
        buff->next += nbytes;
        return;
    }

    // resize of the buffer
    buff->b = realloc(buff->b, buff->size);
    memcpy((char *)buff->b + buff->next, data, nbytes);
    buff->next += nbytes;
    return;
}
```



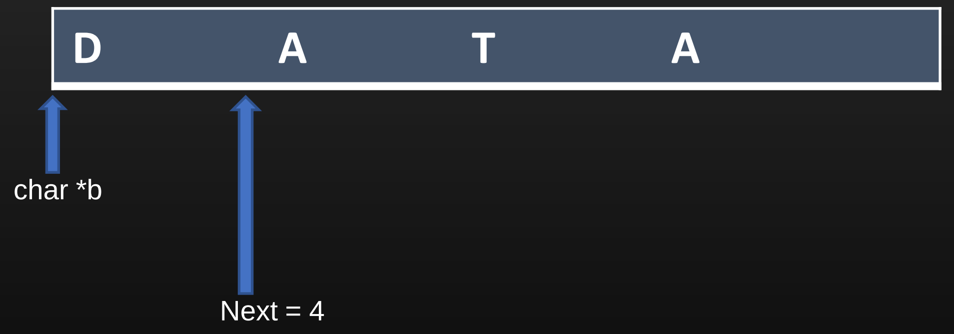
If there is no space to accommodate new data,
Double the size of entire buffer while preserving the
Exiting content

Reading the data from Serialized buffer

```
void
de_serialize_data (char *dest, ser_buff_t *b, int size){

    memcpy(dest, b->b + b->next, size);

    b->next += size;
}
```




```
unsigned int dest;
de_serialize_data ((char *)&dest, b, 4);
```



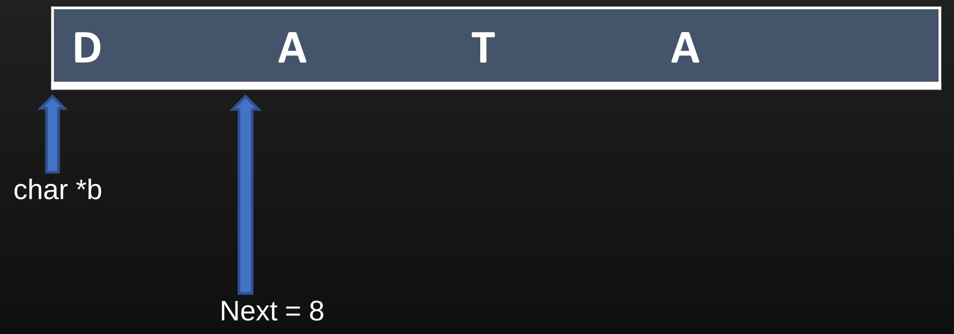

```
de_serialize_data ((char *)&dest, b, 4);
```

Note that : The 'next' is always updated while writing the data into serialized buffer Or reading the data From it

Skipping the Serialized buffer

```
void
serialize_buffer_skip (ser_buff_t *b, int skip_size){

    if(b->next + skip_size > 0 &&
            b->next + skip_size < b->size)
    b->next += skip_size;
}
```


```
serialize_buffer_skip (b, 4);
```




```
serialize_buffer_skip (b, -4);  /*Rewind back wards*/
```

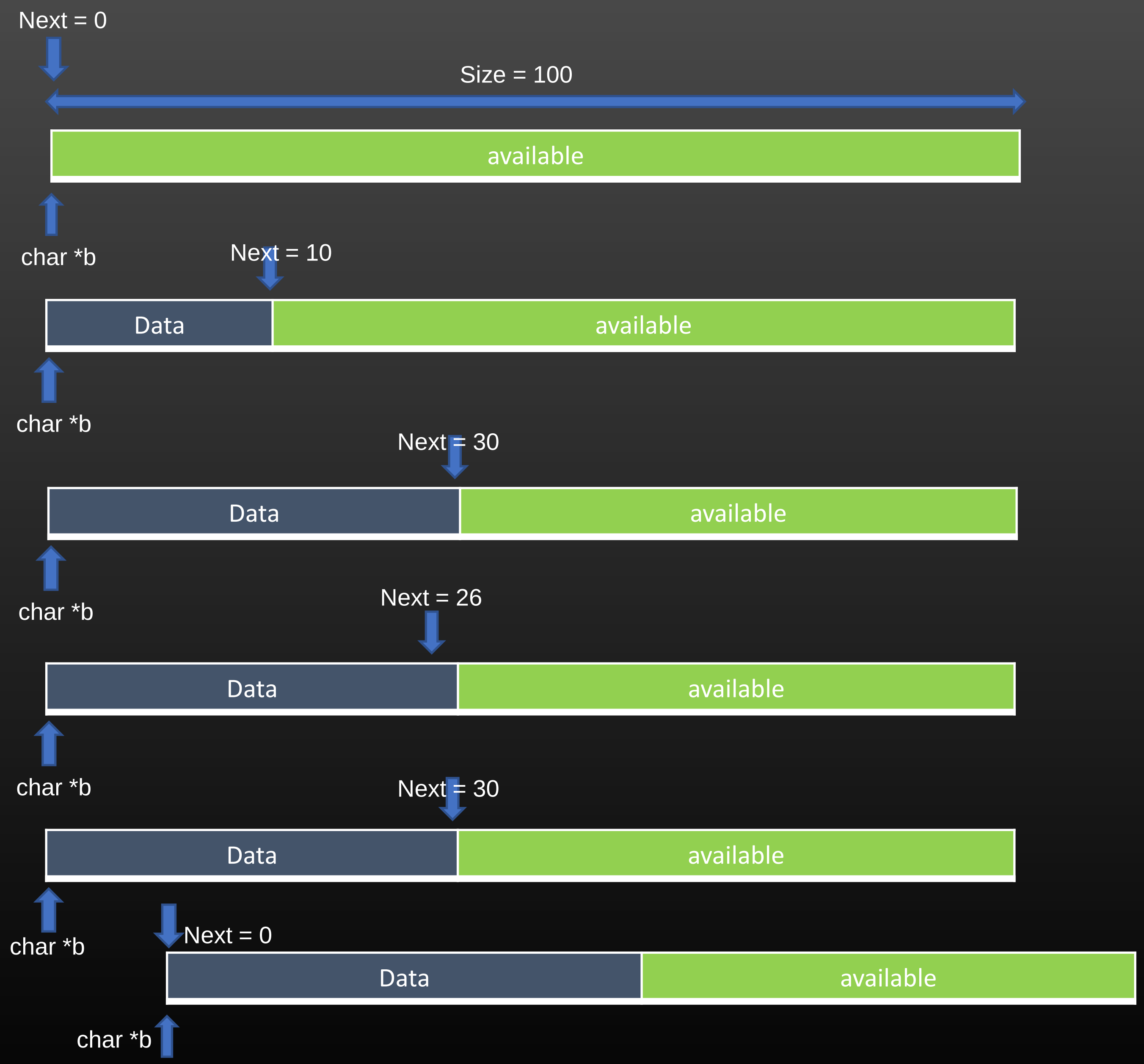
Free the Serialized buffer

Free the serialized buffer once you are done using it.

```
void
free_serialize_buffer (ser_buff_t *b)
```

Streams – Serialized buffer in Action

```
ser_buff_t *b;
init_serialized_buffer(&b);

serialize_data (b, char *data, 10);

serialize_data (b, char *data, 20);

serialize_buffer_skip (b, -4);

serialize_buffer_skip (b, 4);

reset_serialize_buffer(b);
```



Streams

Src code :

git clone http://github.com/casepracticals/RPC

Dir : RPC/De_Serialization
Files : serialize.c/.h

STREAMS

Serialized buffer

Thank you

Data
(De)Serialization
Implementation

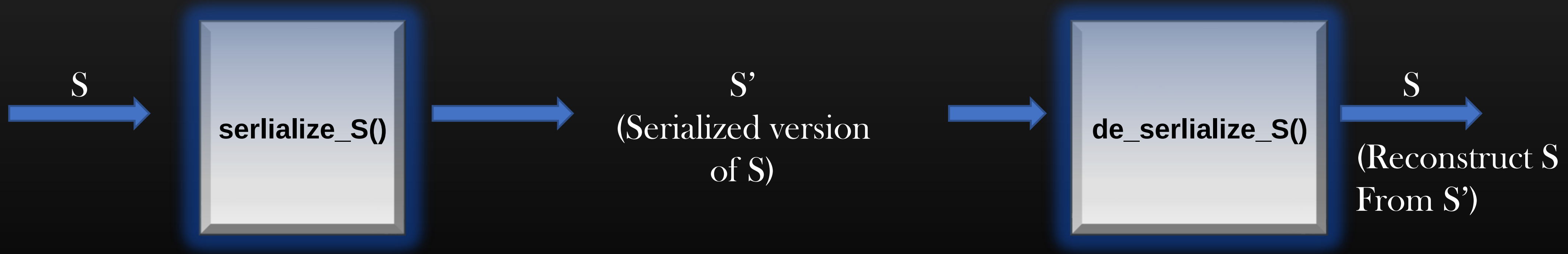
Data Serialization and De-Serialization Implementation

- In this Module, we shall now learn how to implement Serialization and Deserialization of a a given C structure

- Pre-requisite
  - You have understood STREAM data structure
  - Have written at least serialize_data(), de_serialize_data, init_serialized_buffer() APIs
  - Good at pointers

- For simplicity, we shall assume that sending and receiving machines are identical in terms of Hardware architecture, compiler being used and other factors, otherwise serialize/deserialize will become over complicated

- Our Goal will be , for a given C structure S, we shall write serialize_S() and de_serialize_S() APIs as below :



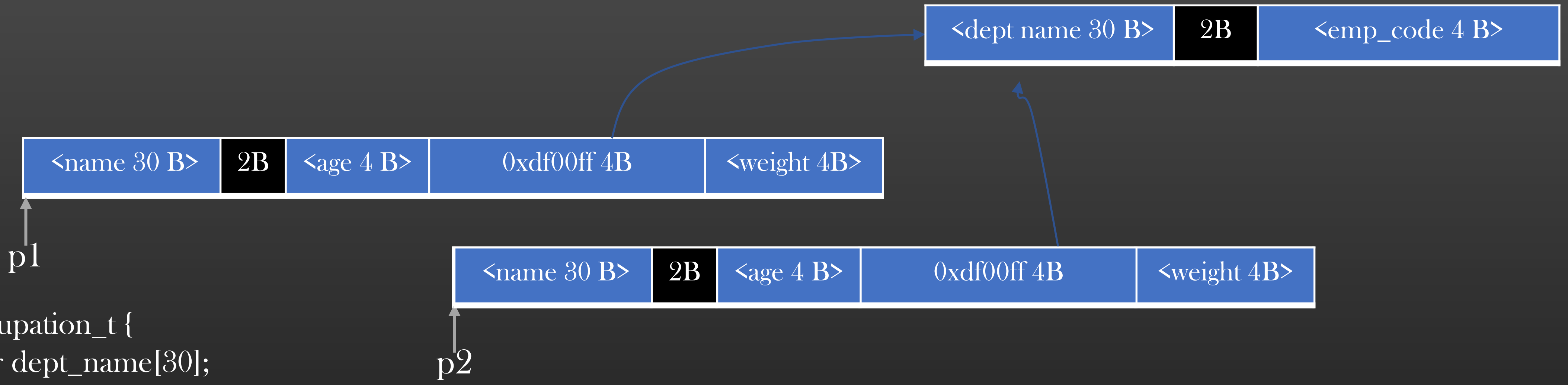
Data Serialization and De-Serialization Implementation

- The process of Data Serialization and De-Serialization are highly recursive in nature
- We shall see, how , recursively we will serialize and de-serialize a given C objects
- Be good at C and pointers, lets do some build up in C

Warning !

- We Discussed the process of Serialization and De-Serialization in detail
- If you copy the data into serialized buffer or read from it wrongly, at any step, be ready for segmentation fault
- It shall be difficult to diagnose where you copied/read wrongly !
- So, be ready to insert your favorite printfs (unless you know GDB) to give you suitable information what is happening , else, debugging will be a nightmare ! Believe me !

C Revision




```
struct occupation_t {
        char dept_name[30];
        int employee_code;
};

struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```

```
struct person_t *p1 /*pointing to object in memory*/;

struct person_t *p2 = calloc(1, sizeof(struct person_t));

*p2 = *p1; /*Called shallow copy*/

free(p1);    /*shallow free*/
```

Just a break ! Let me test your C skill

C Revision

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t occ;
        int weight;
};


struct occupation_t {
        char dept_name[30];
        int employee_code;
};
```

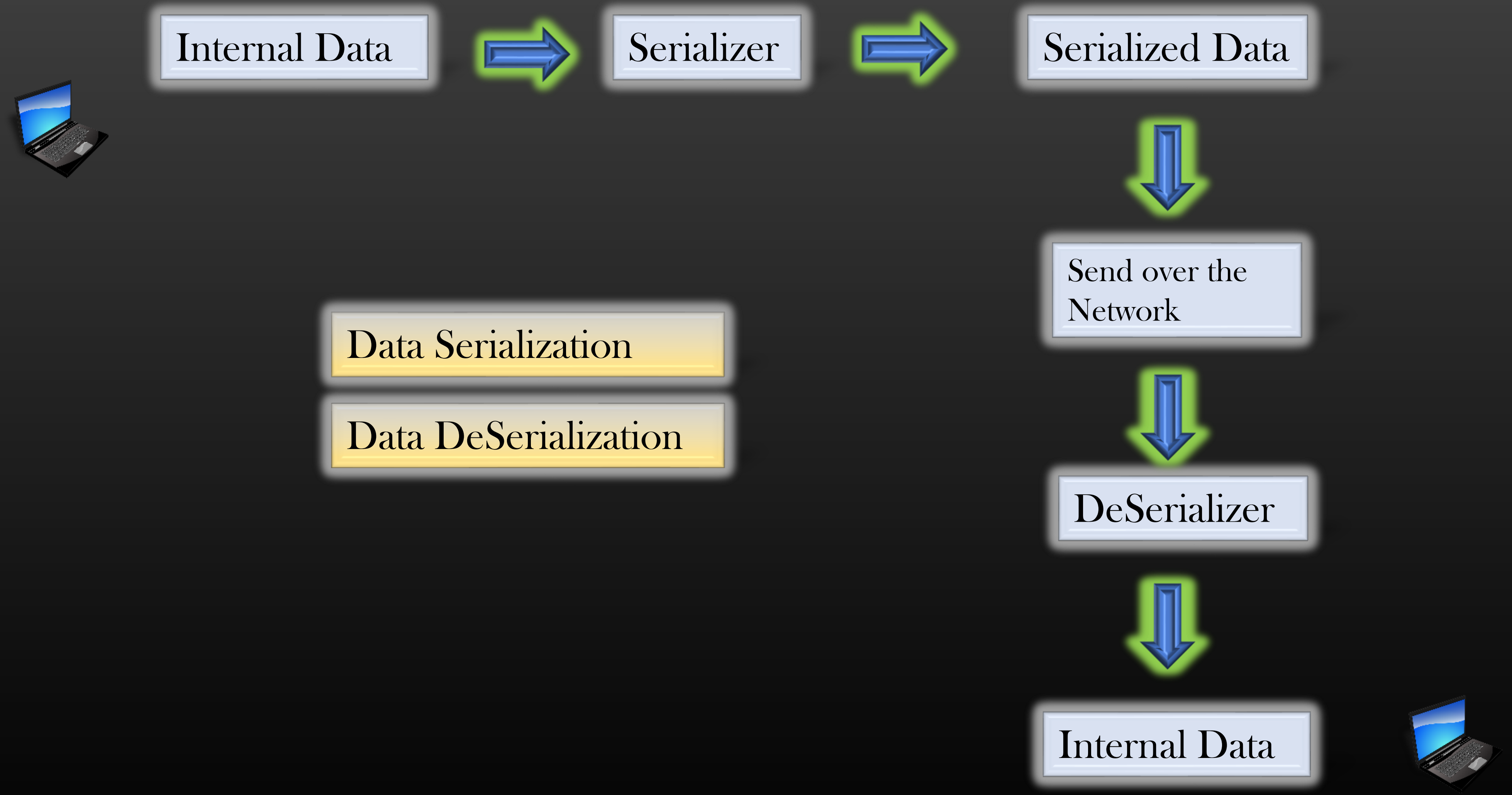
```
struct person_t p1;
memset(&p1, 0, sizeof(p1));
p1 = *p2;    /*shallow copy, p2 is ptr to another person object*/


p1.occ = p2->occ; /*shallow copy internal structure*/
p1 = *foo(); /*foo returns pointer to person_t*/

* is actually equivalent to memcpy


free(&p1.occ);  /*Invalid */
free(p2->occ);  /*Invalid */
```

So, I hope you are comfortable with these caveats and pits in C

C Terminologies

```
struct person_t {
    char name[30];
    int age;
    struct occupation_t occ;   /*This is called embedded structure*/
    struct qualification *q;         /*This is called pointer member */
    int weight;
};
```

Data Serialization and Data Deserialization



Data Serialization and Data Deserialization

Data Serialization

Ex1 : Simple Structures :

S → serlialize_S() → S'
(Serialized version of S)

<name 30 B>	2B	<age 4 B>	<weight 4 B>

Memory foot print (S)

<name 30 B>	<age 4 B>	<weight 4 B>

Serialized Version
(get rid of padding bytes)

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

```
ser_buff_t *b = NULL;
init_serialized_buffer (&b);

void serlialize_person_t (person_t *obj,        /*Input*/
                          ser_buff_t *b)        /*Initialized Serialized buffe

{
        if(!obj){
                unsigned int sentinel = 0xFFFFFFFF;
                serialize_data(b, (char *)&sentinel, sizeof(unsigned int));
                return;
        }
        serlialize_data( b , (char *)obj->name, sizeof(char) * 30);
        serlialize_data( b , (char *)&obj->age, sizeof(int));
        serlialize_data( b , (char *)&obj->weight, sizeof(int));

}
```

These highlighted lines must always present in the beginning of all *serialize_S()* routines.
Lets call it as *Sentinel insertion code.*

Data Serialization and Data Deserialization

Data Serialization

Conventions :

1. For a given structure S, signature of serialize routing will be :

```
void serlialize_S (S *obj, ser_buff_t *b);
```

2. The serialize_data(), to be used for primitive data types, should be written as :

```
serlialize_data( b , (char *)&obj-><field name>, sizeof(field_data_type));                /*If field is not an array*/
serlialize_data( b , (char *)obj-><field name>, sizeof(field_data_type) * array_size);    /*If field is an array*/
```

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

```
ser_buff_t *b = NULL;
init_serialized_buffer (&b);

void serlialize_person_t (person_t *obj,        /*Input*/
                          ser_buff_t *b)        /*Initialized Serialized buffer, O/P*/
{
        if(!obj){
                unsigned int sentinel = 0xFFFFFFFF;
                serialize_data(b, (char *)&sentinel, sizeof(unsigned int));
                return
        }
        serlialize_data( b , (char *)obj->name, sizeof(char) * 30);
        serlialize_data( b , (char *)&obj->age, sizeof(int));
        serlialize_data( b , (char *)&obj->weight, sizeof(int));
}
```

Data Serialization and Data Deserialization

Data Serialization

Ex2 : Nested Structures :

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t occ;
        int weight;
};

struct occupation_t {
        char dept_name[30];
        int employee_code;
};
```

```
ser_buff_t *b = NULL;
init_serialized_buffer (&b);

void serlialize_person_t (person_t *obj,        /*Input*/
                          ser_buff_t *b)        /*Initialized S
{
        if(!obj){
                unsigned int sentinel = 0xFFFFFFFF;
                serialize_data(b, (char *)&sentinel, sizeof(unsigned int));
                return;
        }
        serlialize_data( b , (char *)obj->name, sizeof(char) * 30);
        serlialize_data( b , (char *)&obj->age, sizeof(int));
        serlialize_occupation_t (&obj->occ, b);
        serlialize_data( b , (char *)&obj->weight, sizeof(int));
}
```

<name 30 B>	2B	<age 4 B>	<dept_name 30B>	2B	<emp_code 4B>	<weight 4B>

Memory foot print

Serialized Version

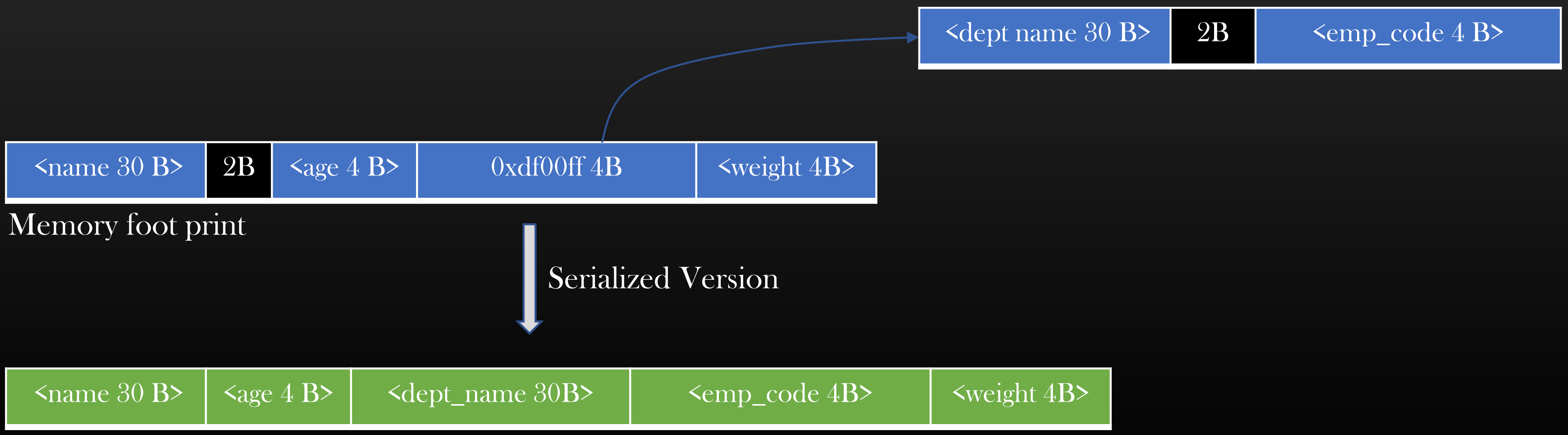
<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

Data Serialization

Structures with pointer members

Ex : 3

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};

struct occupation_t{
        char dept_name[30];
        int emp_code;
};
```



Flat structure

Data Serialization and Data Deserialization

Data Serialization

Structures with pointer members

Ex : 3

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};

struct occupation_t{
        char dept_name[30];
        int emp_code;
};
```

```
ser_buff_t *b = NULL;
init_serialized_buffer (&b);

void serlialize_person_t (person_t *obj,        /*Input*/
                          ser_buff_t *b)        /*Initialized
{
        if(!obj){
                unsigned int sentinel = 0xFFFFFFFF;
                serialize_data(b, (char *)&sentinel, sizeof(unsigned int));
                return;
        }
        serlialize_data( b , (char *)obj->name, sizeof(char) * 30);
        serlialize_data( b , (char *)&obj->age, sizeof(int));
        serlialize_occupation_t (obj->occ , b); /*Serialize internal structure*/
        serlialize_data( b , (char *)&obj->employee_code, sizeof(int));
        serlialize_data( b , (char *)&obj->weight, sizeof(int));
}
```

Serialized Version

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

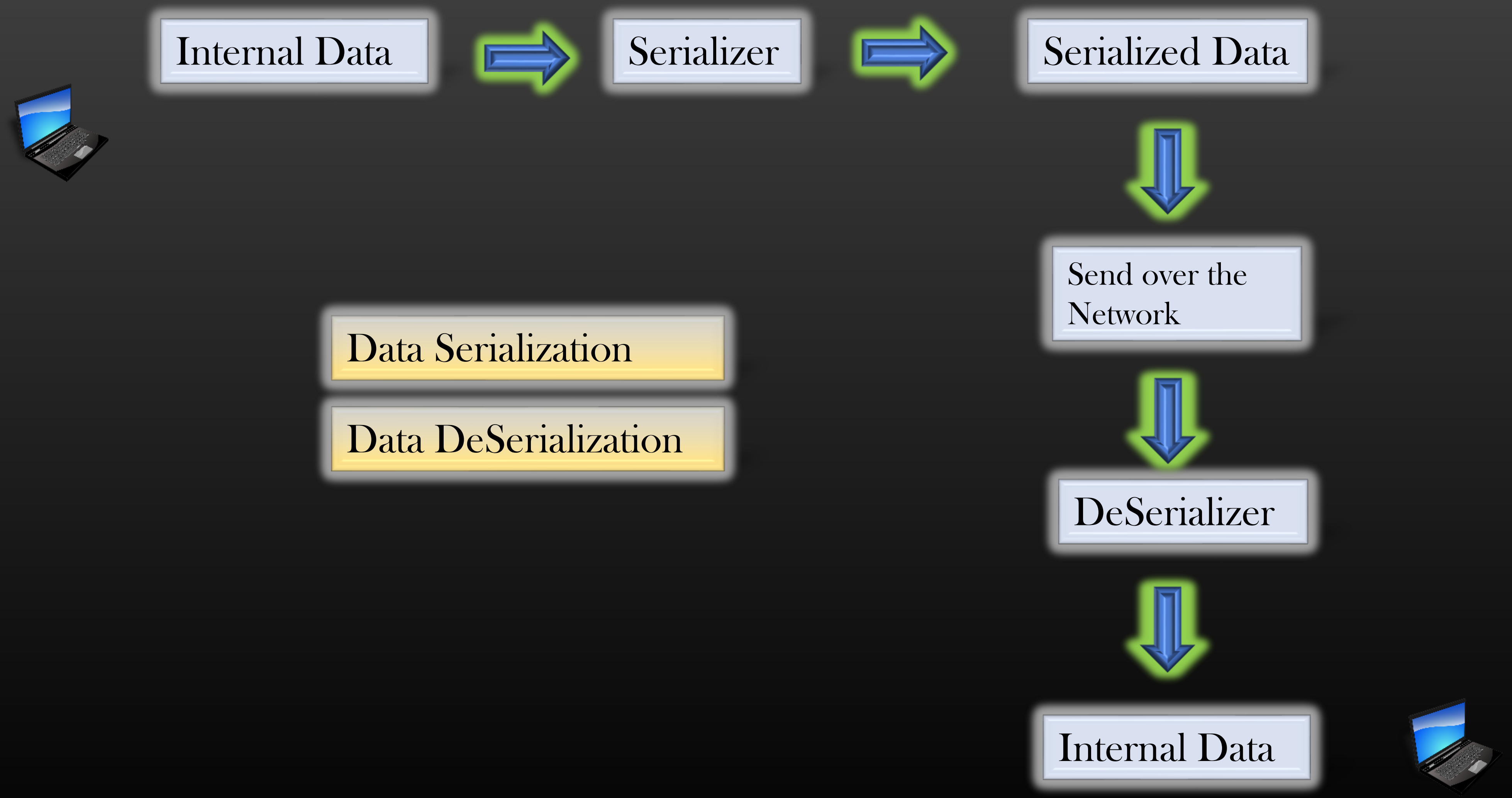
→ When internal member pointer *occ* is not NULL

Serialized Version, when internal structure pointer is NULL

<name 30 B>	<age 4 B>	0xFFFFFFFF	<weight 4B>

→ When internal member pointer *occ* is NULL

Data Serialization and Data Deserialization



Data Serialization and Data Deserialization

Data DeSerialization

Ex1 : Simple Structures :

S'
(Serialized version of S) → **de_serlialize_S()** → S
(Reconstruct S From S')

<name 30 B>	<age 4 B>	<weight 4 B>

Serialized Version (S')

<name 30 B>	2B	<age 4 B>	<weight 4 B>

Memory foot print (S)

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

```
person_t * de_serlialize_person_t (ser_buff_t *b)        /*Input*/
{
        unsigned int sentinel = 0;
    de_serlialize_data((char *)&sentinel, b, sizeof(unsigned int));
        if(sentinel == 0xFFFFFFFF)
                return NULL;
        serialize_buffer_skip(b, -1 * sizeof(unsigned int));
        person_t *obj = calloc(1, sizeof(person_t));
        de_serlialize_data((char *)obj->name, b , sizeof(char) * 30);
        de_serlialize_data((char *)&obj->age, b , sizeof(int));
        de_serlialize_data((char *)&obj->weight, b , sizeof(int));
        return obj;
}
```

These highlighted lines must always be present in the beginning of all de_serialize_S() routines !
Lets call these lines of code as "Sentinel Detection Code"

Data Serialization and Data Deserialization

Data DeSerialization

Conventions :

1. For a given structure S, signature of deserialize routine will be :

```
S* de_serlialize_S (ser_buff_t *b);
```

2. The de_serialize_data should be written as :

```
de_serlialize_data((char *)&obj-><field name>, b , sizeof(field_data_type));                /*If field is not an array
de_serlialize_data((char *)obj-><field name>, b , sizeof(field_data_type) * array_size);    /*If field is an array*/
```

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

```
person_t * de_serlialize_person_t (ser_buff_t *b)        /*Input*/
{
        unsigned int sentinel = 0;
    de_serlialize_data((char *)&sentinel, b, sizeof(unsigned int));
        if(sentinel == 0xFFFFFFFF)
                return NULL;
        serialize_buffer_skip(b, -1 * sizeof(unsigned int));

        person_t *obj = calloc(1, sizeof(person_t));
        de_serlialize_data((char *)obj->name, b , sizeof(char) * 30);
        de_serlialize_data((char *)&obj->age, b , sizeof(int));
        de_serlialize_data((char *)&obj->weight, b , sizeof(int));
        return obj;
}
```

Data Serialization and Data Deserialization

Data DeSerialization

Ex2 : Nested Structures :

Eg :

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t occ;
        int weight;
};

struct occupation_t{
        char dept_name[30];
        int employee_code;
};
```

```
person_t * de_serlialize_person_t (ser_buff_t *b)        /*Input*/
{
        unsigned int sentinel = 0;
    de_serlialize_data((char *)&sentinel, b, sizeof(unsigned int));
        if(sentinel == 0xFFFFFFFF)
                return NULL;
        serialize_buffer_skip(b, -1 * sizeof(unsigned int));

        person_t *obj = calloc(1, sizeof(person_t));
        de_serlialize_data((char *)obj->name , b , sizeof(char) * 30);
        de_serlialize_data((char *)&obj->age , b , sizeof(int));
        struct occupation_t *occ = de_serlialize_occupation_t (b);
        obj->occ = *occ;  /*shallow copy, no need to copy internal objects, if any*/
        free (occ);    /*shallow free, do not free internal member pointers, if any*/
        de_serlialize_data((char *)&obj->weight, b , sizeof(int));
        return obj;
}
```

```
struct occupation_t *
de_serliaze_occupation_t (ser_buff_t *b){
        ...
        ...
}
```

Serialized Version (S')

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

Deserialization

<name 30 B>	2B	<age 4 B>	<dept_name 30B>	2B	<emp_code 4B>	<weight 4B>

Memory foot print (S)

Data DeSerialization

Structures with pointer members

Ex : 3

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};

struct occupation_t{
        char dept_name[30];
        int emp_code;
};
```

```
person_t * de_serlialize_person_t (ser_buff_t *b)        /*Input*/
{
        unsigned int sentinel = 0;
    de_serlialize_data((char *)&sentinel, b, sizeof(unsigned int));
        if(sentinel == 0xFFFFFFFF)
                return NULL;
        serialize_buffer_skip(b, -1 * sizeof(unsigned int));

        person_t *obj = calloc(1, sizeof(person_t));
        de_serlialize_data((char *)obj->name, b , sizeof(char) * 30);
        de_serlialize_data((char *)&obj->age , b , sizeof(int));
        obj->occ = de_serialize_occupation_t (b);
        de_serlialize_data((char *)&obj->weight , b , sizeof(int));
        return obj;
}
```

Serialized Version

<name 30 B>	<age 4 B>	<dept_name 30B>	<emp_code 4B>	<weight 4B>

<dept name 30 B>	2B	<emp_code 4 B>

Reconstructed object

<name 30 B>	2B	<age 4 B>	0xdf00ff4B	<weight 4B>

Reconstructed object

Data Serialization and Data Deserialization

Data DeSerialization

Structures with pointer members

Ex : 3

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ; /* = NULL*/
        int weight;
};

struct occupation_t{
        char dept_name[30];
        int emp_code;
};
```

```
person_t * de_serlialize_person_t (ser_buff_t *b)          /*Input*/
{

        unsigned int sentinel = 0;
    de_serlialize_data((char *)&sentinel, b, sizeof(unsigned int));
        if(sentinel == 0xFFFFFFFF)
                return NULL;
        serialize_buffer_skip(b, -1 * sizeof(unsigned int));

        person_t *obj = calloc(1, sizeof(person_t));
        de_serlialize_data((char *)obj->name, b , sizeof(char) * 30);
        de_serlialize_data((char *)&obj->age , b , sizeof(int));
        obj->occ = de_serialize_occupation_t (b); /*Returns NULL*/
        de_serlialize_data((char *)&obj->weight , b , sizeof(int));
        return obj;
}
```

Serialized Version

<name 30 B>	<age 4 B>	0xFFFFFFFF	<weight 4B>

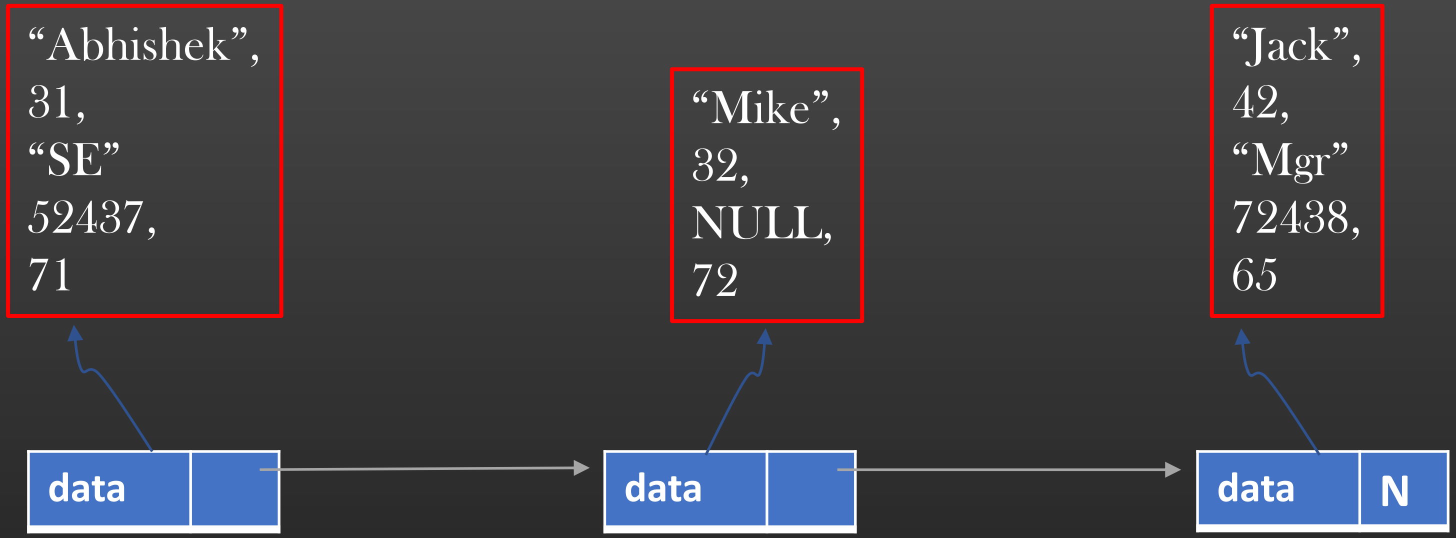
<name 30 B>	2B	<age 4 B>	NULL	<weight 4B>

Reconstructed object

Data Serialization and Data Deserialization

Serializing a Linked List

```
struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};
```




```
struct list_t {
        struct list_node_t *head;
};
```

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```

Abhishek 30B	31 4B	SE 30B	52437 4B	71 4B	Mike 30B	32 4B	0xFFFFFFFF 4B	72 4B	Jack 30B	42 4B	Mgr 30B	72438 4B	65 4B	0xFFFFFFFF 4B

Serializing a Linked List

```
struct list_t {
        struct list_node_t *head;
};
```

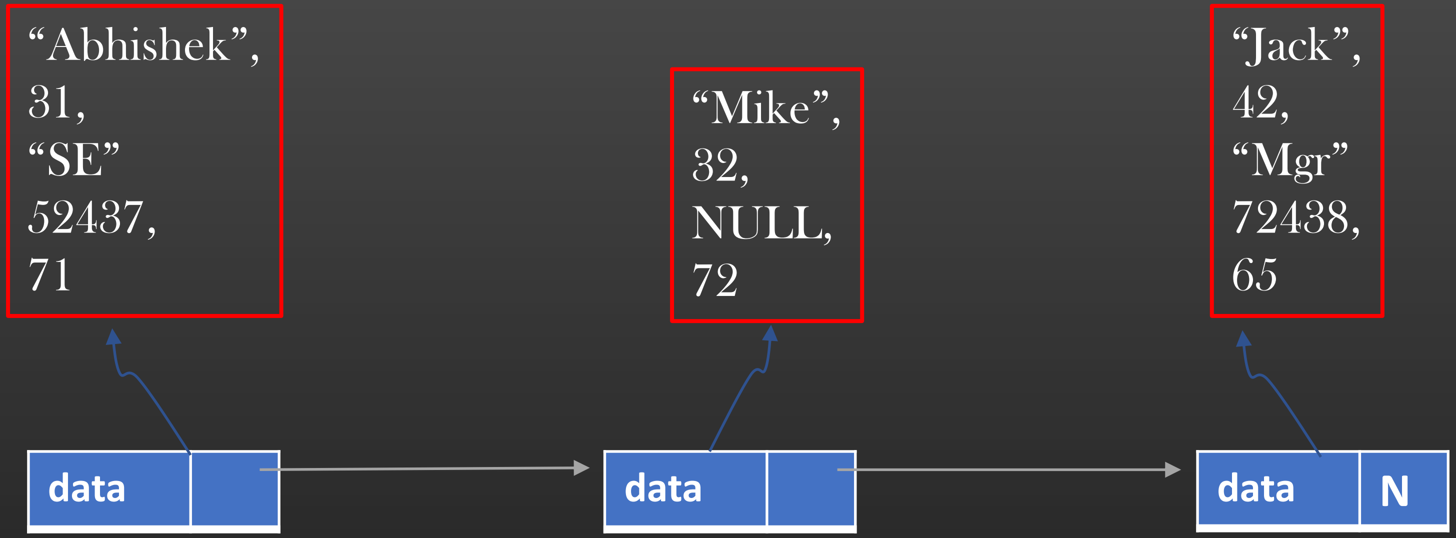
```
void serialize_list_t (struct list_t *obj, ser_buff_t *b)
{
        SENTINEL_INSERTION_CODE;
        serialize_list_node_t ( obj->head, b);
}
```

```
struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};
```

```
void  serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b)
{
        SENTINEL_INSERTION_CODE;
        serialize_person_t (obj->data, b);
        serialize_list_node_t (obj->right, b);
}
```

De-Serializing a Linked List

```
struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};

struct list_t {
        struct list_node_t *head;
};

struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```



Reconstruct the entire Linked List (De-serialize)

Serialized Linked List

Abhishek 30B	31 4B	SE 30B	52437 4B	71 4B	Mike 30B	32 4B	0xFFFFFFFF 4B	72 4B	Jack 30B	42 4B	Mgr 30B	72438 4B	65 4B	0xFFFFFFFF 4B

Data Serialization and Data Deserialization

De-Serializing a Linked List

```
struct list_t {
        struct list_node_t *head;
};

struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};

struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```

```
struct list_t  * de_serlialize_list_t (ser_buff_t *b)          /*Input*/
{
        SENTINEL_DETECTION_CODE;

        struct list_t *obj = calloc(1, sizeof(struct list_t));
        obj->head =  de_serialize_list_node_t (b);
        return obj;
}

struct list_node_t  * de_serlialize_list_node_t (ser_buff_t *b)
{
        SENTINEL_DETECTION_CODE;

        struct list_node_t *obj = calloc(1, sizeof(struct list_node_t));
        obj->data =  de_serialize_person_t (b);
        obj->right = de_serlialize_list_node_t (b);
        return obj;
}
```

Data Serialization and Data Deserialization

PIT FALL !!

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```

```
struct occupation_t {
        struct work_ex_t *wx;  /* = NULL*/
        char offc_addr[30];
};
```

S1

<Abhishek 30 B>	2B	<31 4 B>	0xffdd0c	71

NULL	Silicon valley

= obj->occ->wx = NULL

Serialized Object S1’

Abhishek 30B	31 4B	0xFFFFFFFF	Silicon Valley	71

S2

<Abhishek 30 B>	2B	<31 4 B>	NULL	71

= obj->occ = NULL

Serialized Object S2’

Abhishek 30B	31 4B	0xFFFFFFFF	71

Can you think how *de_serialize_person_t()*
Would reconstruct the actual object from
Serialized objects S1’ and S2’ respectively ?

Data Serialization and Data Deserialization

PIT FALL !!

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```

```
struct occupation_t {
        struct work_ex_t *wx;  /* = NULL*/
        char offc_addr[30];
};
```

Serialized Object S1'

Abhishek 30B	31 4B	0xFFFFFFFF	Silicon Valley	71

Serialized Object S2'

Abhishek 30B	31 4B	0xFFFFFFFF	71

Case 1

Case 2

de_serlialize_person_t()

Reconstruct the object S2

<Abhishek 30 B>	2B	<31 4 B>	NULL	71

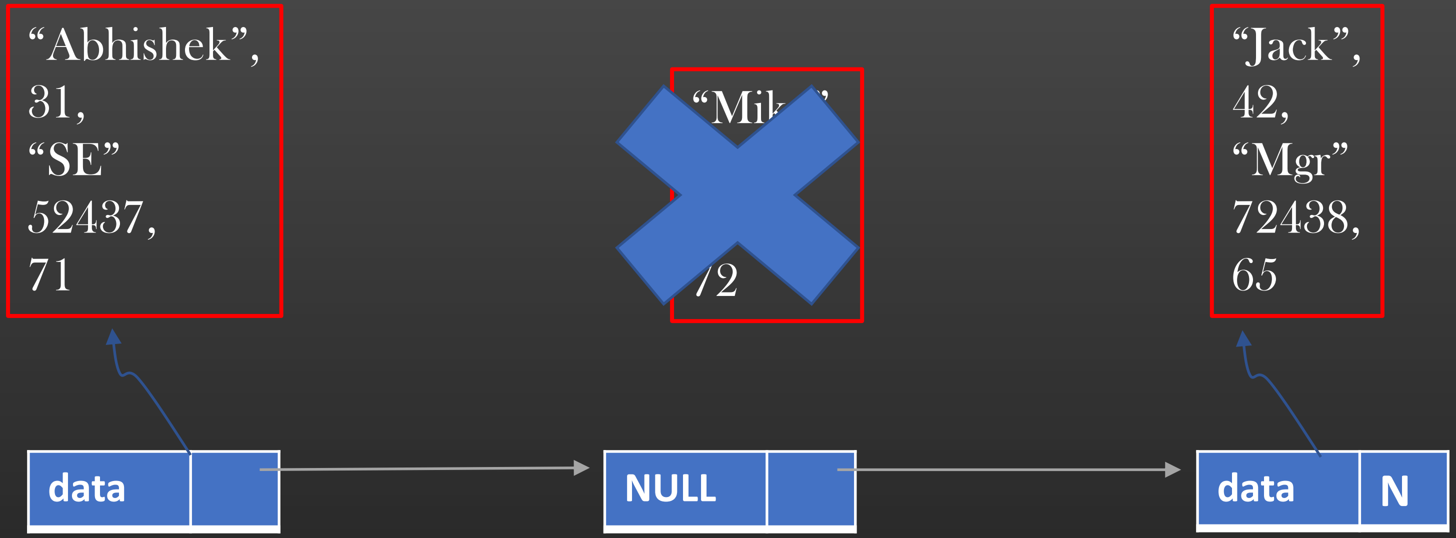
In both the cases, *de_serialize_person_t()* reconstruct the object S2 only

How to deal with this problem ?

Data Serialization and Data Deserialization

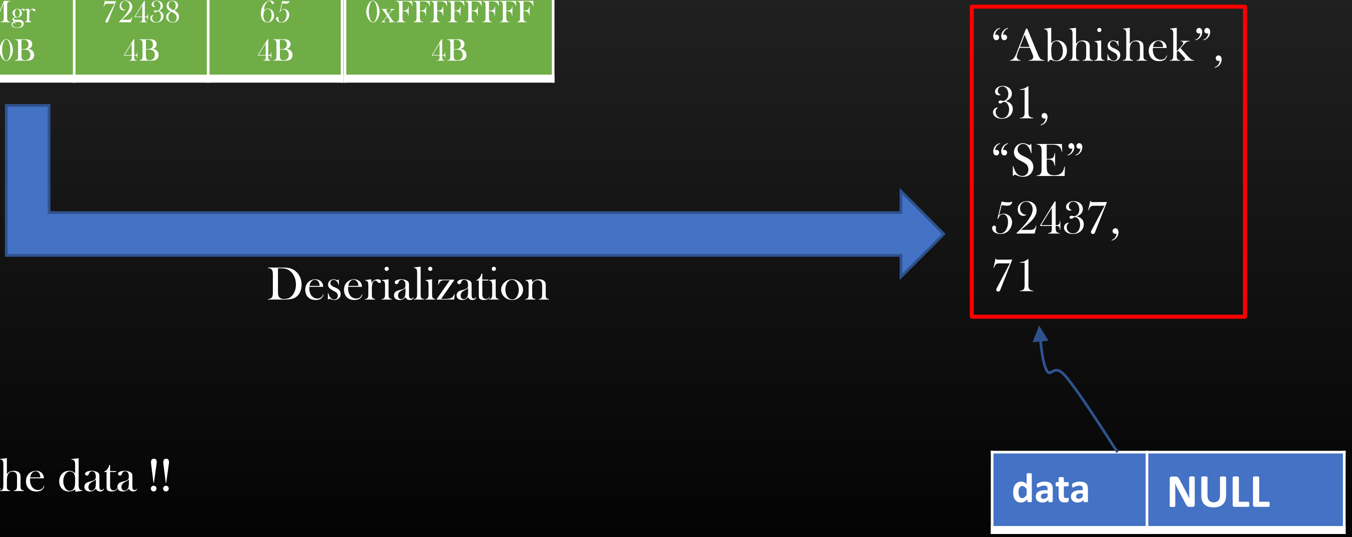
PIT FALL -> Just Another Example

```
struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};

struct list_t {
        struct list_node_t *head;
};
```



Serialized Linked List

Abhishek 30B	31 4B	SE 30B	52437 4B	71 4B	0xFFFFFFFF 4B	Jack 30B	42 4B	Mgr 30B	72438 4B	65 4B	0xFFFFFFFF 4B



☹

What's Wrong ! We have lost the data !!

PIT FALL !!

Final Verdict and Solution

Root Cause :

- De - Serializer cannot distinguish in between these two cases :
  - Case 1 : Whether the pointer member T of the parent structure S is NULL

    Or

Case 1 :

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;  /* = NULL*/
        int weight;
};
```

PIT FALL !!

Final Verdict and Solution

Root Cause :

- De - Serializer cannot distinguish in between these two cases :
  - Case 1 : Whether the pointer member T of the parent structure S is NULL

    Or

  - Case 2 : Whether the first member of the nested structure T is NULL
  - Solution :
    - The first member of the internal object (pointed by pointer member field) should not be pointer member, Avoid it

    Note : The starting address of the C objects is same as starting address of Ist member of the object

Case 2 :

```
struct person_t {
        char name[30];
        int age;
        struct occupation_t *occ;
        int weight;
};
```

```
struct occupation_t {
        struct work_ex_t *wx;  /* = NULL*/
        char offc_addr[30];
};
```

Serializing and De-Serializing arrays

```
struct _person_t_ {

        char name[32];
        company_t   dream_companies[3];          /*Array of type company_t of size 3*/
        company_t  *non_core_companies[3];       /*Array of pointers of type company_t of size 3*/
} person_t;


typedef struct _company_t_ {
    char comp_name[32];
    int emp_strength;
    person_t * CEO;
} company_t;
```

Serializing and De-Serializing arrays

```
struct _person_t_ {

            char name[32];
            company_t   dream_companies[3];
            company_t  *non_core_companies[3];
} person_t;


typedef struct _company_t_ {
    char comp_name[32];
    int emp_strength;
    person_t * CEO;
} company_t;
```

```
void serialize_person_t (person_t *obj, ser_buff_t *b){

        SENTINEL_INSERTION_CODE;

        int i = 0;
        for( i = 0 ; i < 3 ; i++){
                serialize_company_t (&obj->dream_companies[i], b);
        }

        for( i = 0 ; i < 3 ; i++){
                serialize_company_t (obj->dream_companies[i], b);
        }
}
```

Serializing and De-Serializing arrays

```
struct _person_t_ {

            char name[32];
            company_t   dream_companies[3];
            company_t  *non_core_companies[3];
} person_t;


typedef struct _company_t_ {
    char comp_name[32];
    int emp_strength;
    person_t * CEO;
} company_t;
```

```
person_t * de_serialize_person_t (ser_buff_t *b){

        SENTINEL_DETECTION_CODE;

        person_t *obj = calloc(1, sizeof(person_t));
        de_serialize_data((char *)&obj->name, b, sizeof(char) * 32);
        int i = 0;

        (for i = 0; i < 3; i++){
                company_t *company = de_serialize_company_t(b);
                obj->dream_companies[i] = *company;   /*shallow copy*/
                free(company); /*shallow free*/
        }

        (for i = 0; i < 3; i++){
                company_t *company = de_serialize_company_t(b);
                obj->non_core_companies[i] = company;  /*No copy required*/
        }
        return obj;
}
```

Limitations

- Hmm ! Do you see any limitation !
- How will you serialize & De-serialize structures with back pointers , say, Circular Linked Lists, Graphs etc ? Do you see a problem.
- Yes, the problem is your (De)-Serializers will go in infinite loop with cyclic Objects ! I leave it to you to try !
- Note that, Structures and not Cyclic, its objects (instantiation of structures) which can be cyclic
- So, do not try to Deserialize the objects with back-pointers

Code Walk

Example of Serialization and Deserialization of Structures

Download :
git clone http:://github.com/csepracticals/RPC

Go inside Dir : RPC/De_Serialization

Files :
person_t.h
sentinel.h
serialization_person.c
serialize.c
serialize.h

Steps to compile :
gcc -g -c serialization_person.c -o serialization_person.o
gcc -g –c serialize.c –o serialize.o
gcc -g serialization_person.o serialize.o -o exe

Run :
./exe

(De)Serializing generic Data structures
And
handling void pointers

Generic Data structures

- Generic Data structures are Structures which work with every other data structure
- These data structure are not tied to any specific application, and can be used with any application in general

For example , Linked List, Stacks, Queues, Trees, Graphs and many others

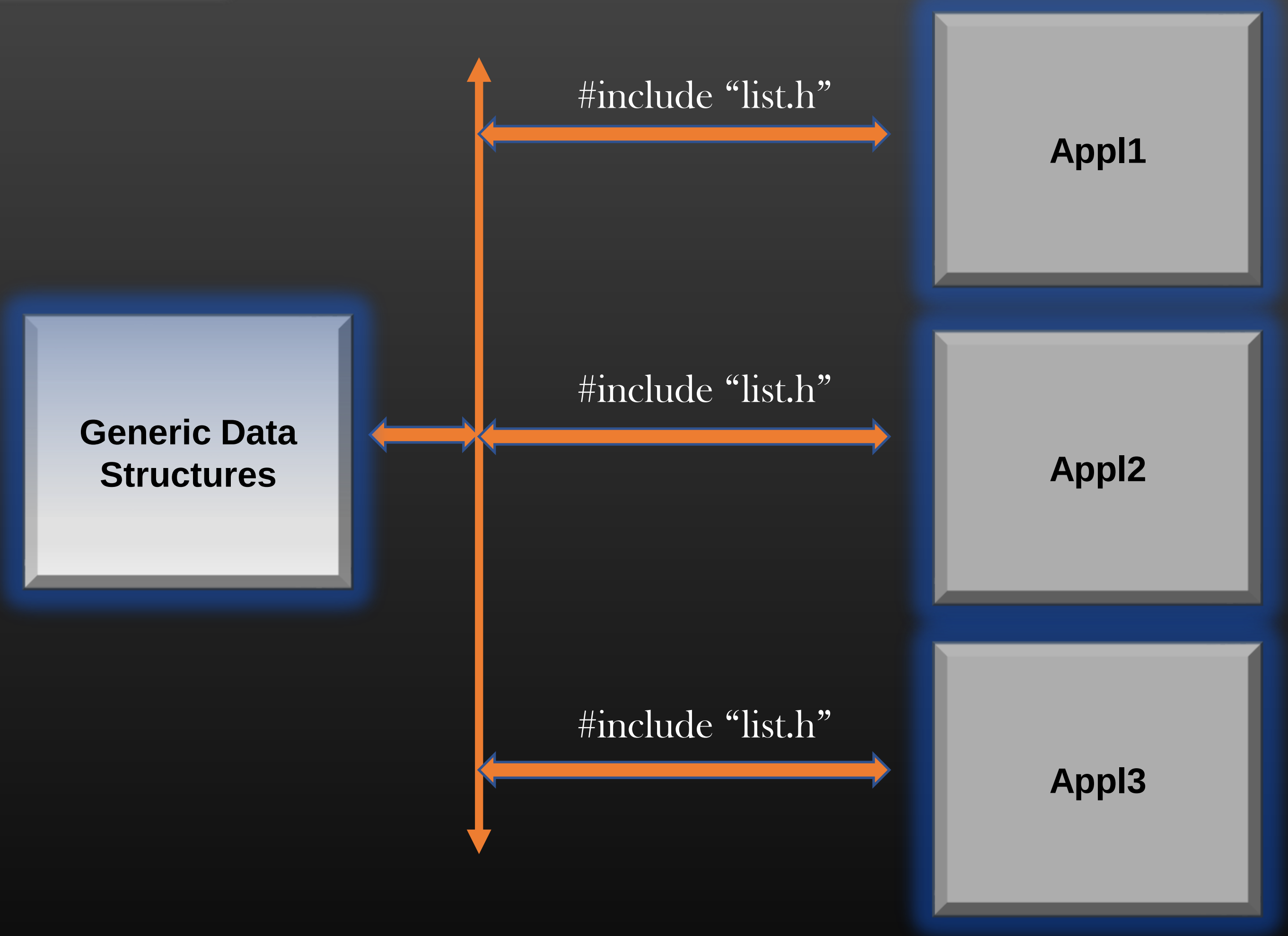
```
struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};
```

✖

```
struct list_node_t {
        void *data;
        struct list_node_t *right;
};
```

✔

Generic Data structures



Thumb Rule : You cannot write any application specific code/Data structure in Generic DS files

Data Serialization and Data Deserialization

(De)Serializing generic Data structures

Generic Linked List

```
struct list_node_t {
        void* data;
        struct list_node_t *right;
};
```

So, how would you write the Serializing/ De-serializing routines for this modified list_node_t structure ? ☺

```
void  serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b)
{
        /*Sentinel Insertion Code*/
        serialize_XXX (obj->data, b);   /* Problem : What is XXX ??*/
        serialize_list_node_t (obj->right, b);
}
```

```
struct list_node_t  * de_serlialize_list_node_t (ser_buff_t *b)
{
        /* Sentinel Detection Code */
        struct list_node_t *obj = calloc(1, sizeof(struct
list_node_t));
        obj->data =  de_serialize_XXX (b);  /*Problem !!*/
        obj->right = de_serlialize_list_node_t (b);
        return obj;
}
```

We need to know the data type of *data* field to write the serializing/deserializing routines !
But this info is not available. Then how to Serialize/Deserialize the generic data structures ??

(De)Serializing generic Data structures

Whenever we Need to write a code which Must be generic and should support all Data types , we must think of Function pointers !!

Example : Write a sorting program to sort the array of Data type T !

Let's Use the function pointers to help us out !

(De)Serializing generic Data structures

```
struct list_node_t {
        struct person_t *data;
        struct list_node_t *right;
};
```

```
struct list_node_t {
        void* data;
        struct list_node_t *right;
};
```

Old code

```
void  serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b)
{
        serialize_person_t (obj->data, b);
        serialize_list_node_t (obj->right, b);
}
```

New code

```
void  serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b,
                                            void (*serialize_fn_ptr)(void
*, ser_buff_t *b))
{
        serialize_fn_ptr(obj->data, b); /*Generic, work for all Data types !!*/
        serialize_list_node_t (obj->right, b);
}
```

Data Serialization and Data Deserialization

(De)Serializing generic Data structures

Generic Linked List

```
struct list_node_t {
        void *data;
        struct list_node_t *right;
};

struct list_t {
        struct list_node_t *head;
};
```

Example :

- Suppose we are using generic linked list to store the data objects of type struct person_t
- Our goal is to use the below API for serializing our linked list without making the linked list Serializing routine user defined data structure specific

```
void serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b,
                            void (*serialize_fn_ptr)(void *, ser_buff_t *))
```

(De)Serializing generic Data structures

Step 1 :

Write a wrapper function for *void serialize_person_t (struct person_t *obj, ser_buff_t *b).* Step 1 will give us a generic interface.

```
void serialize_person_t_wrapper(void *obj, ser_buff_t *b) {
        serialize_person_t (obj, b);
}
```

Step 2 :

Change the signature of serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b) to

```
void  serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b,
                              void (*serialize_fn_ptr)(void *, ser_buff_t *))
```

Step 3 :

Since, serialize_list_node_t() is invoked from serialize_list_t (), we need to pass the wrapper fn_ptr as additional argument to serialize_list_t() also which in turn will pass this additional fn_ptr argument to serialize_list_node_t().
New definitions of Serializing routines for linked list is captured on next slide.

(De)Serializing generic Data structures

Step 4 :

Change the signature of serialize_list_t (struct list_t *obj, ser_buff_t *b) to

```
void serialize_list_t (struct list_t *obj, ser_buff_t *b, void (*serialize_fn_ptr)(void *, ser_buff_t *))
```

Step 5 :

Simply invoke the new modified serializing routine as follows :

```
serialize_list_t ( obj, b, serialize_person_t_wrapper);
serialize_list_node_t (obj, b, serialize_person_t_wrapper);
```

(De)Serializing generic Data structures

New Modified Serializing Routines for Generic Linked List

```
void serialize_list_t (struct list_t *obj, ser_buff_t *b,
                void (*serialize_fn_ptr)(void *, ser_buff_t *));
{
                serialize_list_node_t ( obj->head, b, serialize_fn_ptr);
}
```

```
void  serialize_list_node_t (struct list_node_t *obj, ser_buff_t *b,
                                    void (*serialize_fn_ptr)(void *,
ser_buff_t *))
{
            /*Generic, work for all Data types !!*/
              serialize_fn_ptr ( obj->data, b);
              serialize_list_node_t (obj->right, b);

}
```

- You can see, that, Linked List serialization code is independent of any application specific data type !
- It is the responsibility of the application while serializing the linked list to pass the function pointer for the data type for which linked list is a holder

How to invoke : *serialize_list_t (obj, b , serialize_person_t_wrapper);*

(De)Serializing generic Data structures

Assignment !!

Apply the same technique to write *de_serializing* routines for Generic Linked Lists !

Data Serialization and Data Deserialization

Coding Assignments

Data (De)Serialization Implementation

Thank you

Agenda

- What is RPC ?
- RPC Architecture and Design
- Implementation of RPC from scratch
  - Client Stubs
  - Server Stubs
- What problem RPC solves ?
- Summary

Pre-requisite : Minimal socket programming, C programming, Data Serialization

Remote Procedure Calls (RPC)

Remote Procedure Calls (RPC)

What is RPC ?

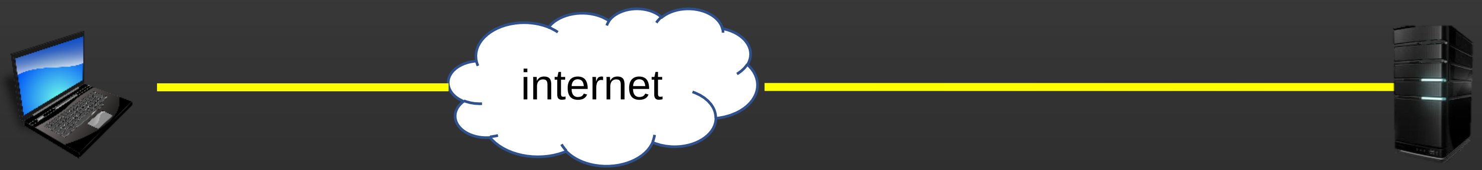
- As the name suggest, RPC means, invoking a function/procedure which is implemented and running on a remote machine in the network
- In programming, you usually call functions , but those functions are the part of same program
- In RPC, your application invokes the functions which resides on a remote machine
- RPC creates an illusion as if you are invoking the local function, but in reality, that function is remote
- You cannot implement that remote function locally, because internally that function does certain things which is not supported on your local machine

Eg : Accessing other company's confidential Database (Security)
Function which are computationally intensive and require more resources

What is RPC ?

Let us take an Example of Multiplying two numbers to understand RPC steps

A Big Picture




```
void main(){

        int a = multiply(2,3);
        print a;
}
```

Remote invocation (a, b)

RPC Result (c)

```
int multiply(int a, int b){

        int c = a*b;
        return c;
}
```

- Your machine do not know how to Multiply two integers
- You rely on service from remote servers

- Instead the remote server performs the operation
- Return the required result back to you

Obviously, Since the two machines are interacting over a network, underlying communication Channel is Network – Socket programming needs to be used to send and receive RPC data

Remote Procedure Calls (RPC)

RPC Steps

The entire RPC mechanisms can be divided into 9 Steps :

Step 1 : Client program invoking the RPC with arguments
eg : multiply (2, 3)

Step 2 : Client program Converting the arguments into Serialized form of Data
(Data Serialization/Marshalling)

Step 3 : Client program Shipping the Serialized Data over network to RPC Server

Step 4 : RPC Server receiving the Serialized Data obtained in step 3 from client

Step 5 : RPC Server Un-marshalling the Serialized data recvd from client in step 4
(Data De-Serialization/Un-Marshalling)

Step 6 : RPC Server Invoking the actual RPC with Argument
eg : result = multiply (2, 3) invocation on Server side

Step 7 : RPC Server Serializing the *result* of RPC invoked in step 6
(Data serialization/Marshalling)

Step 8 : RPC Server sending the Serialized result of Step 7 back to client

Step 9 : Client UnMarshalling the result recvd and print/process it
(Data De-Serialization/Un-Marshalling)

Phase 1 : Client Side Steps
Communication from Client to Server Side

1 : Serialization/Marshalling

Phase 2 : Server Side Steps
Communication from Server back to Client Side

1 : DeSerialization/UnMarshalling
2 : Serialization/Marshalling

Phase 3 : Client Side Final Step

1 : DeSerialization/Un-Marshalling

RPC Steps

Phase 1 : Communication From Client to Server

RPC Client

Application
(multiply.c)

Step 1

int
multiply_rpc
(int a, int b)

multiply_client_stub
_marshal
(int a, int b)

Step 2
Marshalling/
Serialization

Serialized Data
rpc_send_recv (. . .)

0 1 0 1 0 1 0 0 1

Step 3
Sending Data to Server

RPC Server

Client Side Steps

RPC Step # 2 Marshalling of Arguments on Client side

INPUT **OUTPUT**

RPC : multiply_rpc (int a, int b)

- Signature : ser_buff_t * multiply_client_stub_marshal(Arg1, Arg2, . . .)
- *multiply_client_stub_marshal()* routine is responsible to serialize the arguments of the RPC

RPC Step # 3 Sending and Receiving Data from Server

```
void rpc_send_recv
(ser_buff_t  *send_ser_data,  ser_buff_t*empty_ser_buffer);
```

- send_ser_data - Serialized data prepared in step 2 by client_stub_marshal() to be sent to server
- empty_ser_buffer - Empty buffer to recv the data (result) received from server

What we have done so far ?

- We have learnt , how to Marshal the arguments into single buffer on client side (*client_stub_marshal()*)
- Sending the buffer to the server over the network *(rpc_send_recv())*
- Let us see the Actual Implementation . . .

RPC Steps

The entire RPC mechanisms can be divided into 9 Steps :

Step 1 : Client program invoking the RPC with arguments
eg : multiply (2, 3)

Step 2 : Client program Converting the arguments into Serialized form of Data
(Data Serialization/Marshalling)

Step 3 : Client program Shipping the Serialized Data over network to RPC Server

Step 4 : RPC Server receiving the Serialized Data obtained in step 3 from client

Step 5 : RPC Server Un-marshalling the Serialized data recvd from client in step 4
(Data De-Serialization/Un-Marshalling)

Step 6 : RPC Server Invoking the actual RPC with Argument
eg : result = multiply (2, 3) invocation on Server side

Step 7 : RPC Server Serializing the *result* of RPC invoked in step 6
(Data serialization/Marshalling)

Step 8 : RPC Server sending the Serialized result of Step 7 back to client

Step 9 : Client UnMarshalling the result recvd and print/process it
(Data De-Serialization/Un-Marshalling)

Phase 1 : Client Side Steps
Communication from Client to Server Side

1 : Serialization/Marshalling

Phase 2 : Server Side Steps
Communication from Server back to Client Side

1 : DeSerialization/UnMarshalling
2 : Serialization/Marshalling

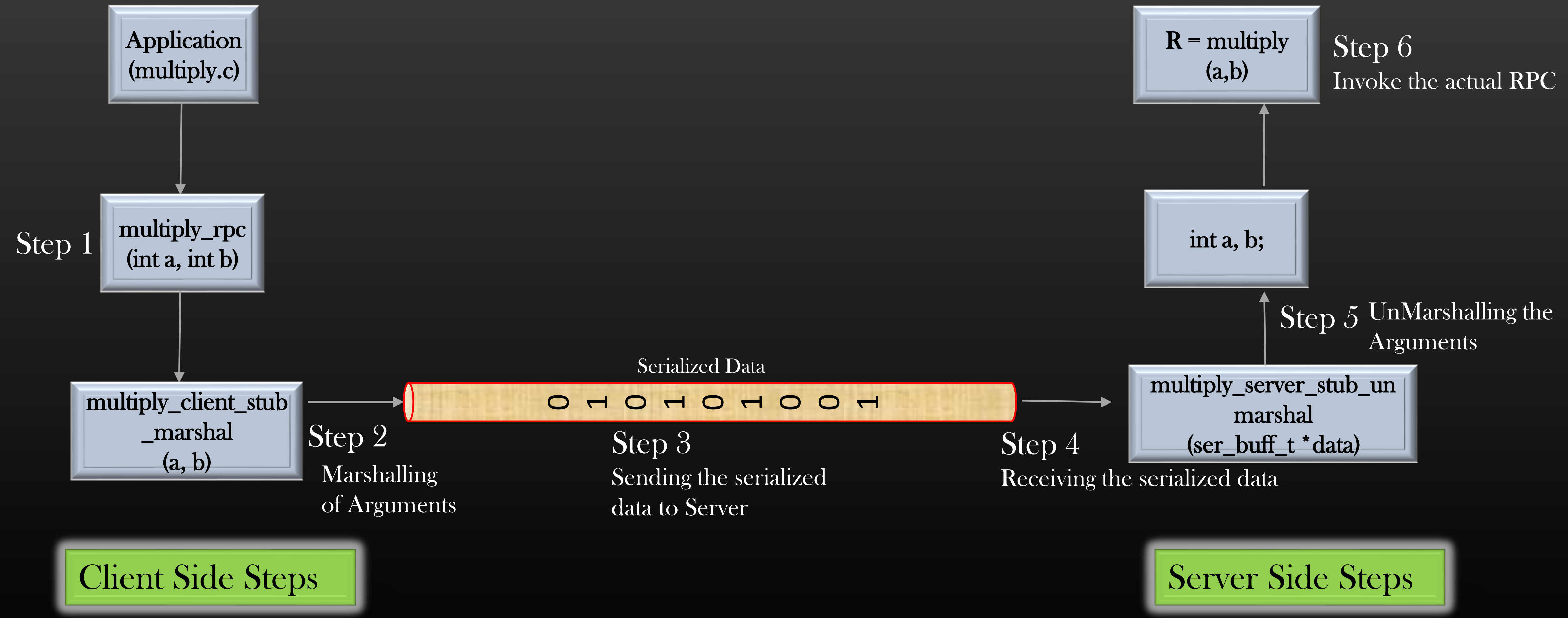
Phase 3 : Client Side Final Step

1 : DeSerialization/Un-Marshalling

Remote Procedure Calls (RPC)

RPC Steps

Steps 4,5,6



RPC Step # 4 Receiving the serialized data from client
Step # 5 Un-Marshalling of Serialized Data On Server side
Step # 6 Invoke the actual RPC on server side

Phase 2 : Communication From Client to Server

INPUT

OUTPUT

Step 4

a	b

Serialized data
Received from client

<rpc return type >
server_stub_unmarshall
(ser_buff_t *rcv_ser_data)

Step 5
Unmarshall the arguments

arg1 → Int a

arg2 → Int b

Remote Invocation
Eg : multiply (int a, int b)

Step 6

- *server_stub_unmarshall()* is responsible to reconstruct the arguments from received serialized buffer
- Signature : *<rpc return type> server_stub_unmarshal (ser_buff_t *ser_data)*

What we have done so far ?

- We have learnt , how the server Unmarshal and reconstruct the arguments (*server_stub_unmarshal()*)
- With arguments reconstructed, Server calls the actual RPC
- Let us see the Actual Implementation . . .

Remote Procedure Calls (RPC)

RPC Steps

The entire RPC mechanisms can be divided into 9 Steps :

Step 1 : Client program invoking the RPC with arguments
eg : multiply (2, 3)

Step 2 : Client program Converting the arguments into Serialized form of Data
(Data Serialization/Marshalling)

Step 3 : Client program Shipping the Serialized Data over network to RPC Server

Step 4 : RPC Server receiving the Serialized Data obtained in step 3 from client

Step 5 : RPC Server Un-marshalling the Serialized data recvd from client in step 4
(Data De-Serialization/Un-Marshalling)

Step 6 : RPC Server Invoking the actual RPC with Argument
eg : result = multiply (2, 3) invocation on Server side

Step 7 : RPC Server Serializing the *result* of RPC invoked in step 6
(Data serialization/Marshalling)

Step 8 : RPC Server sending the Serialized result of Step 7 back to client

Step 9 : Client UnMarshalling the result recvd and print/process it
(Data De-Serialization/Un-Marshalling)

Phase 1 : Client Side Steps
Communication from Client to Server Side

1 : Serialization/Marshalling

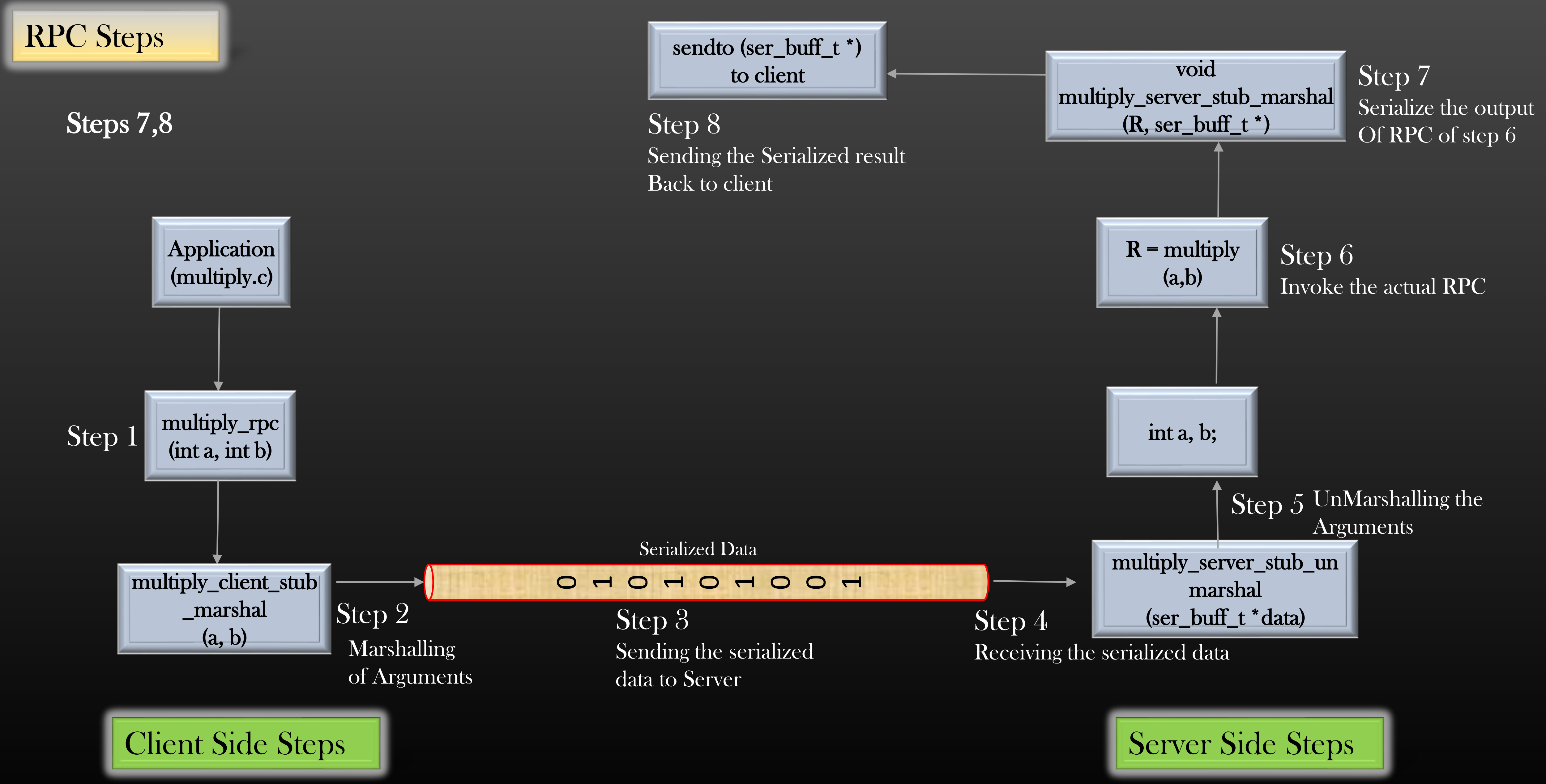
Phase 2 : Server Side Steps
Communication from Server back to Client Side

1 : DeSerialization/UnMarshalling
2 : Serialization/Marshalling

Phase 3 : Client Side Final Step

1 : DeSerialization/Un-Marshalling

Remote Procedure Calls (RPC)



Remote Procedure Calls (RPC)

RPC Step # 7 Serialize the RPC output
Step # 8 Send the serialized output back to client

Phase 2 : Communication From Server to Client

Step 8
Send the serialized data back to client
Over the network

void multiply_server_stub_marshall
(R, ser_buff_t *b)

Step 7

RPC result
R
Step 6

Remote Procedure Calls (RPC)

RPC Steps

The entire RPC mechanisms can be divided into 9 Steps :

Step 1 : Client program invoking the RPC with arguments
eg : multiply (2, 3)

Step 2 : Client program Converting the arguments into Serialized form of Data
(Data Serialization/Marshalling)

Step 3 : Client program Shipping the Serialized Data over network to RPC Server

Step 4 : RPC Server receiving the Serialized Data obtained in step 3 from client

Step 5 : RPC Server Un-marshalling the Serialized data recvd from client in step 4
(Data De-Serialization/Un-Marshalling)

Step 6 : RPC Server Invoking the actual RPC with Argument
eg : result = multiply (2, 3) invocation on Server side

Step 7 : RPC Server Serializing the result of RPC invoked in step 6
(Data serialization/Marshalling)

Step 8 : RPC Server sending the Serialized result of Step 7 back to client

Step 9 : Client UnMarshalling the result recvd and print/process it
(Data De-Serialization/Un-Marshalling)

Phase 1 : Client Side Steps
Communication from Client to Server Side

1 : Serialization/Marshalling

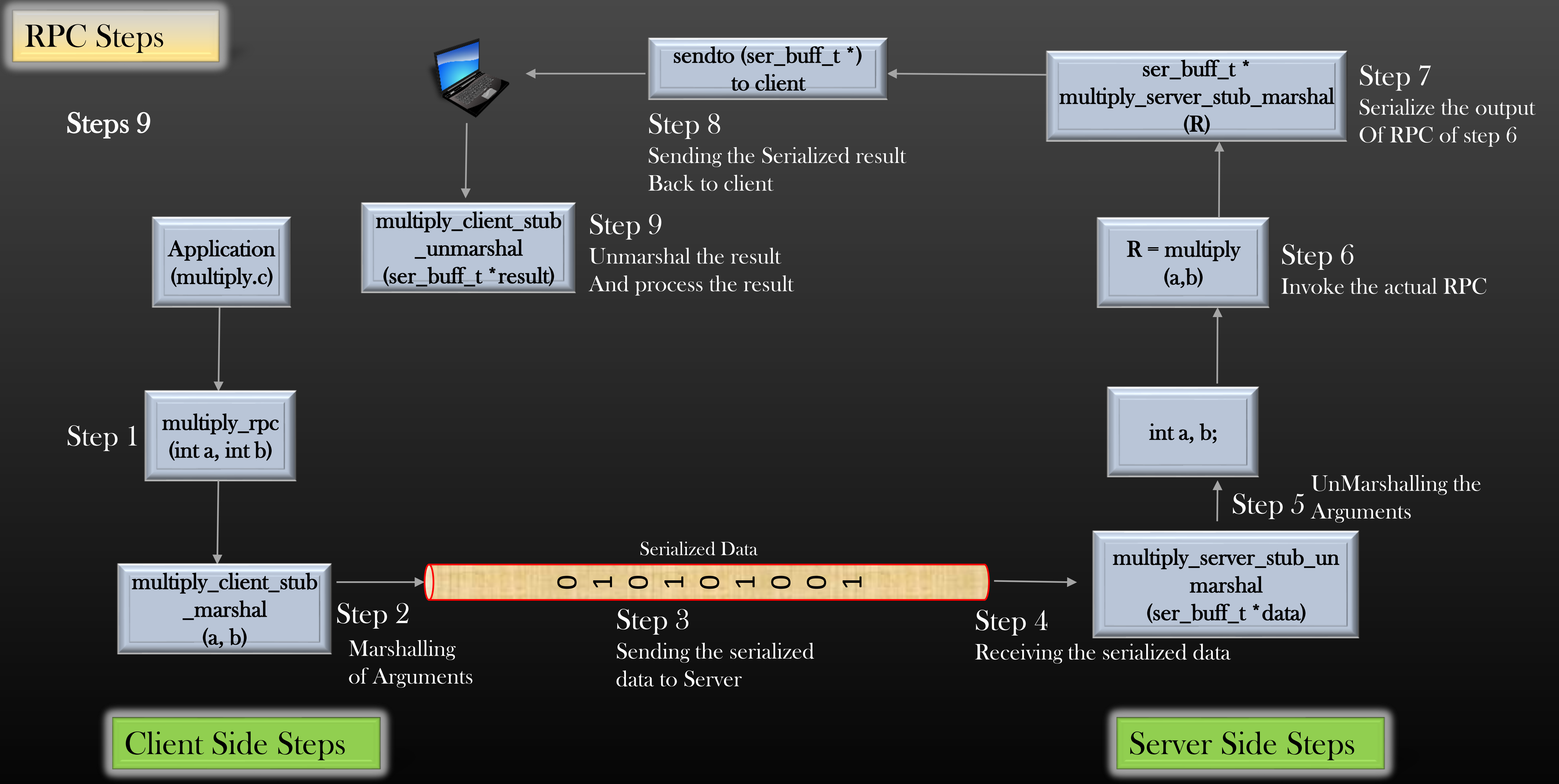
Phase 2 : Server Side Steps
Communication from Server back to Client Side

1 : DeSerialization/UnMarshalling
2 : Serialization/Marshalling

Phase 3 : Client Side Final Step

1 : DeSerialization/Un-Marshalling

Remote Procedure Calls (RPC)



RPC Step # 9 Unmarshal the Serialize data received from Server

Phase 3 : Client side Final Step

Got the final RPC result
Hurray !!

<rpc_return_type >
multiply_client_stub_unmarshal
(ser_buff_t *result)

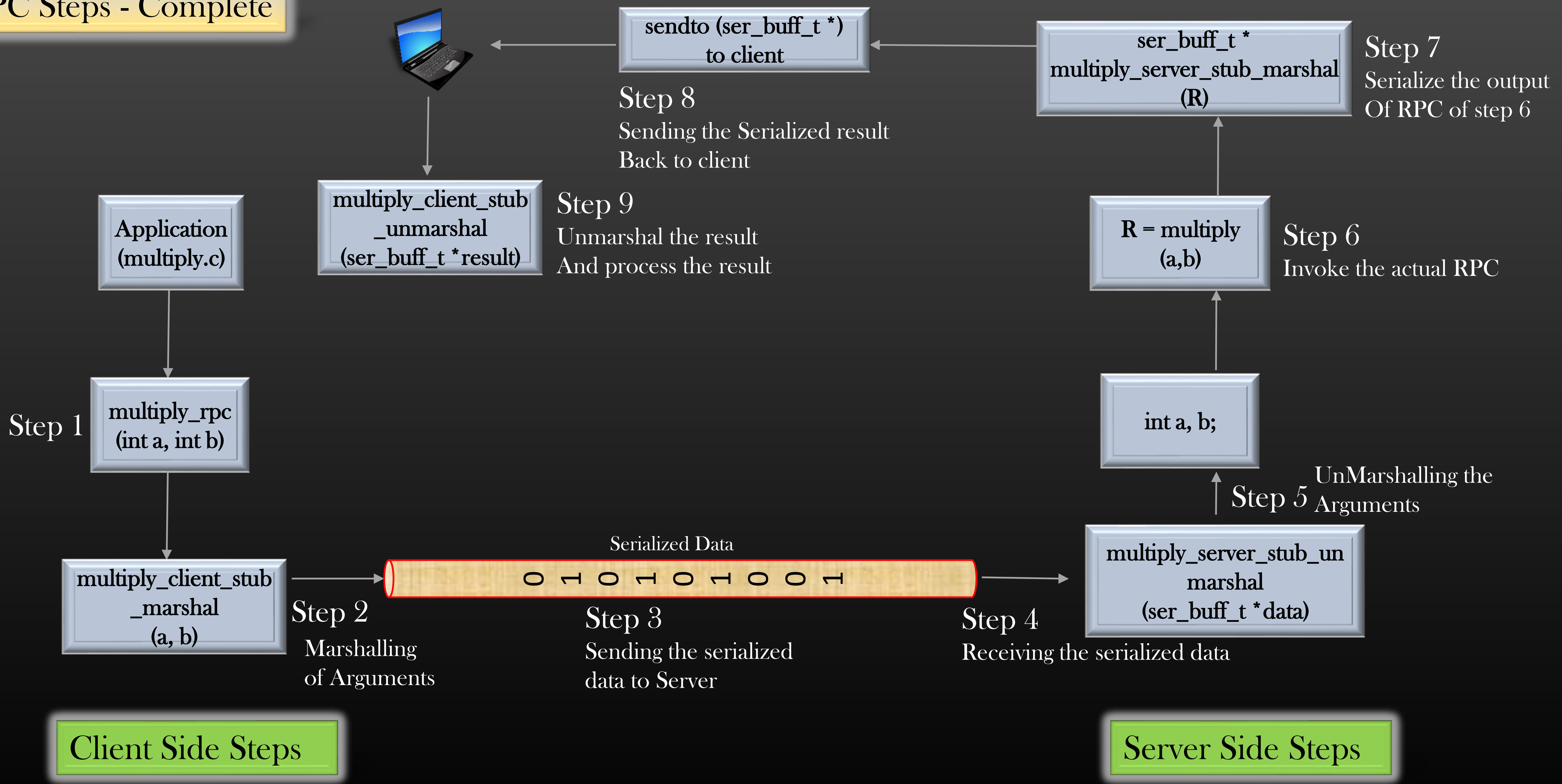
Step 9

Serialized data from server

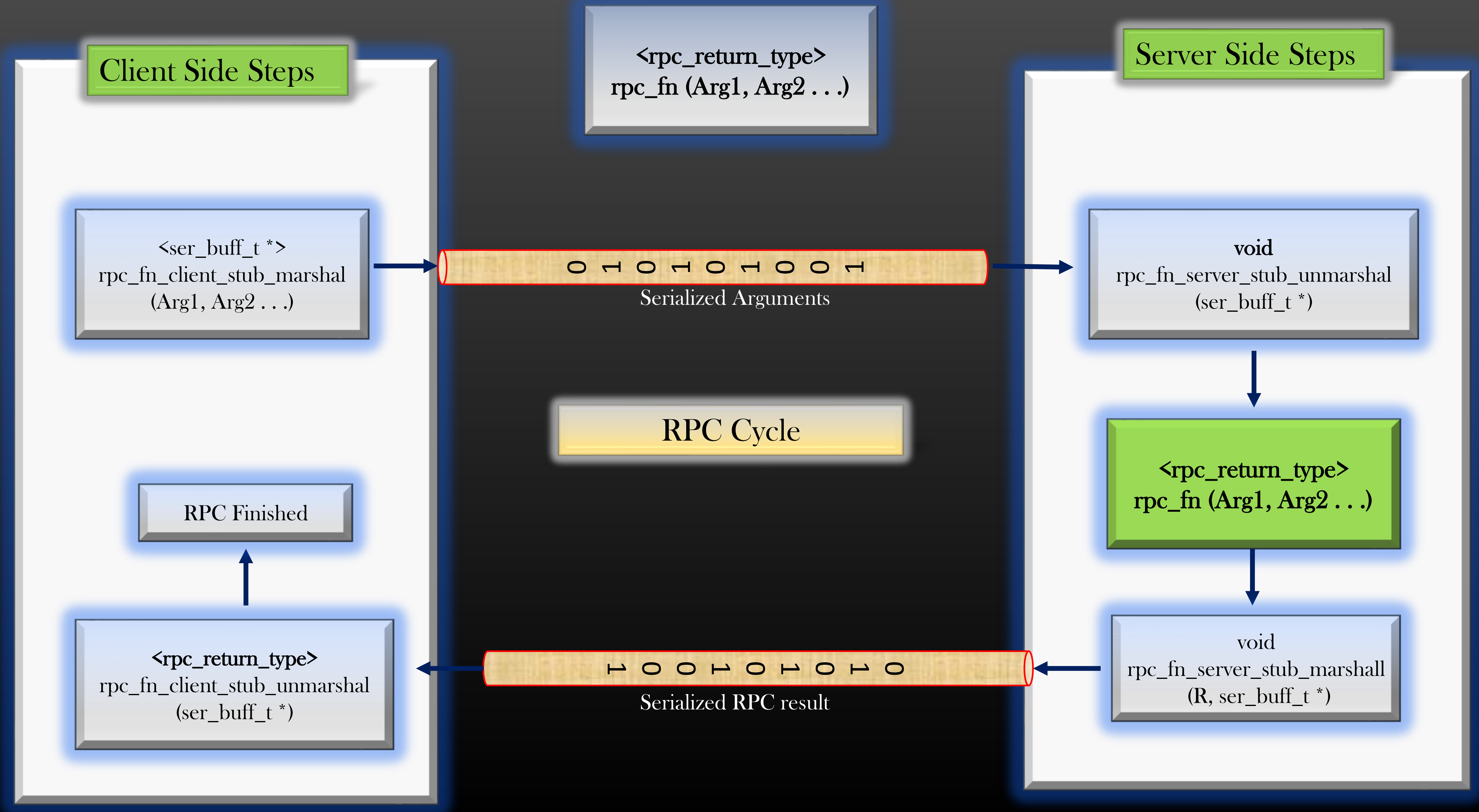
Step 8

Remote Procedure Calls (RPC)

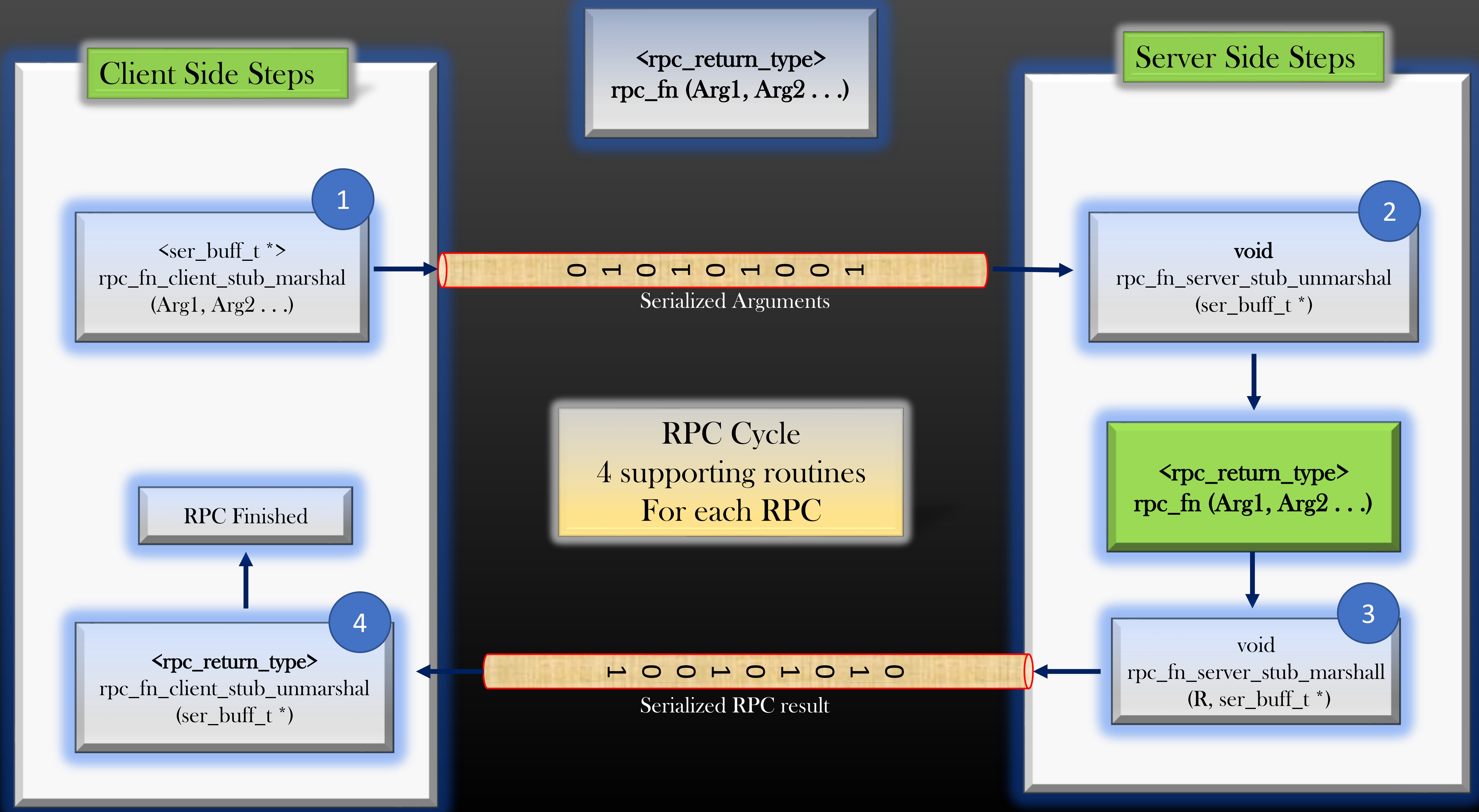
RPC Steps - Complete



Remote Procedure Calls (RPC)



Remote Procedure Calls (RPC)

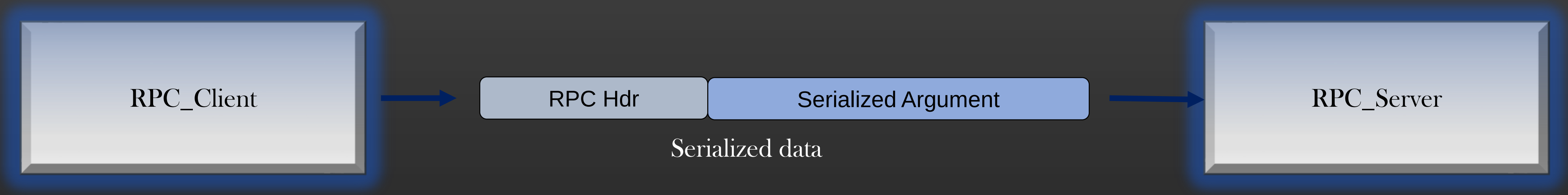


RPC Identity

- When RPC Client sends Serialized RPC Arguments to RPc Server, How RPC Server would know which RPC the client is trying to invoke ?
- In our example, client sent serialized arguments which were two integers
- RPC Client may want to *multiply, add, subtract* or may request any other operation on these two integers
- In Serialized data sent by client, we need to encode the information about RPC identity

Remote Procedure Calls (RPC)

RPC Identity



- RPC Client needs to attach a header to serialized arguments it is sending to RPC server
- This header should contain information about RPC Id.

```
typedef struct rpc_hdr_{
        uint rpc_id;
        uint pay_load_size;
} rpc_hdr_t;
```

Tip : Before preparing the serialized data from Arguments of RPC , insert the Header first in the serialized buffer.

Serialized Size of the structure

Eg :

```
struct person_t {
        char name[30];
        int age;
        int weight;
};
```

Size = 40 B

2 B of extra padding

Compiler size of structure : sizeof(struct person_t) = 40 Bytes

Serialized size of structure : sizeof(char) * 30 + sizeof(int) + sizeof(int) = 38 Bytes

= Sum of sizes of all individual fields of a structure

Remote Procedure Calls (RPC)

RPC Identity

```
/* client_stub_marshal() */
ser_buff_t *
multiply_client_stub_marshal (int a, int b){

    ser_buff_t *client_send_ser_buffer = NULL;
    init_serialized_buffer_of_defined_size(&client_send_ser_buffer,
                                           MAX_RECV_SEND_BUFF_SIZE);

    /*Reserve the hdr space*/
    serialize_buffer_skip(client_send_ser_buffer, serialized_hdr_size);
    /*prepare the hdr*/
    rpc_hdr_t rpc_hdr;
    rpc_hdr.rpc_id = MULTIPLY_ID;
    rpc_hdr.pay_load_size = 0; /*We don't know yet*/

    /*Serialize the first Argument*/
    serialize_data(client_send_ser_buffer, (char *)&a, sizeof(int));
    /*Serialize the second Argument*/
    serialize_data(client_send_ser_buffer, (char *)&b, sizeof(int));

    /*Now we know the size of payload, insert the hdr*/
     rpc_hdr.pay_load_size = get_serialize_buffer_data_size(client_send_ser_buffer) –
                                              - serialized_hdr_size);
    copy_in_serialized_buffer_by_offset (client_send_ser_buffer, sizeof(rpc_hdr.rpc_id),
                                         (char *)&rpc_hdr.rpc_id, 0);
    copy_in_serialized_buffer_by_offset (client_send_ser_buffer, sizeof(rpc_hdr.pay_load_size),
                                         (char *)&rpc_hdr.pay_load_size,
sizeof(rpc_hdr.rpc_id));
    return client_send_ser_buffer;
}
```

Remote Procedure Calls (RPC)

RPC Identity

Obtain the RPC ID

Obtain the Payload size

```
void
rpc_server_process_msg(ser_buff_t *server_recv_ser_buffer,
                        ser_buff_t *server_send_ser_buffer){

   rpc_hdr_t rpc_hdr;
   de_serialize_data((char *)&rpc_hdr.rpc_id, server_recv_ser_buffer,
              sizeof(rpc_hdr.rpc_id));
   de_serialize_data((char *)&rpc_hdr.pay_load_size, server_recv_ser_buffer,
              sizeof(rpc_hdr.pay_load_size));

   if (rpc_hdr.rpc_id == MULTIPLY_ID){
              int res = multiply_server_stub_unmarshal(server_recv_ser_buffer);
              multiply_server_stub_marshal(res, server_send_ser_buffer);
   }
   else if (rpc_hdr.rpc_id == ADDITION_ID){
              int res = addition_server_stub_unmarshal(server_recv_ser_buffer);
              addition_server_stub_marshal(res, server_send_ser_buffer);
   }
   else if ( . . . ){
              . . .
              . . .
   }
}
```

RPC use cases

- The RPC Servers can support thousands of RPCs.
- For RPCs to be usable by clients, RPC Servers have to publicly publish the RPC signatures
- Using RPCs, RPC Servers grant a monitored and controlled access to RPC client's on its (Server's) resources
  - For example : Airlines such as Indigo, Emirates, Air Asia etc can grant an access to their databases to websites such as Makemytrip.com Or Airbnb etc through RPC interface
  - RPC is a way by virtue of which the Resource holder (Airlines) allow the Resource requesters (Makemytrip.com etc) to access their resources (Airlines) in a way that holder's want

"Hey !! Here are my resources, I grant an access to you and you can perform only these operations on my resources. Not more than that !!"

- REST (**RE**presentation **S**tate Transfer) is a brother of RPC – Same purpose but slightly in a different way, with different benefits

RPC use cases

Favorable conditions to Use RPC

Tight coupling : The (distributed) components of the system are designed to work together, and changing one will likely impact all of the others. It is unlikely that the components will have to be adapted to communicate with other systems in the future.
Eg : OpenStack, Hadoop, Operating Systems

Uniform language : Systems in which All (or mostly all) components are be written in a single language. It is unlikely that additional components written in a different language will be added in the future

Assignment

Q1. Create a linked list of person_t objects. Insert some records in linked list.
Implement an RPC struct *list_t * rpc_get_senior_citizens (struct list_t *),* which returns the list of person_t objects who's age exceeds 60.

```
struct list_t {
        struct list_node_t *head;
 };

struct list_node_t {
        struct person_t *data;
        struct list_node_t *next;
};

struct person_t {
        char name[32];
        int age;
        int weight;
};
```

Q2. Implement another RPC which returns the record of the person_t object for eldest person in the list.

Make use of RPC hdr to distinguish between two RPC invocation on RPC Server side.

RPC Exercises

Q3. Write the RPC as below to compute the sum of integers stored in a linked list.

```
unsigned int rpc_compute_list_sum (ll_t *list);
```

Q4. Write the RPC as below to which takes an input the linked list of integers and return two sub-linked lists – one is the linked list of odd integers and other is the linked list of even integers.

```
ll_pair_t *rpc_linked_list_splitter (ll_t *list);
```

Where , ll_pair_t is defined as :

```
ll_pair_ {
        ll_t *even_lst;
        ll_t *odd_lst;
} ll_pair_t;
```

RPC Exercises

Q5. Write the RPC to find the sum of all nodes of a binary search tree

```
unsigned int rpc_find_sum_bst (tree_t *tree);
```

Note : You can serialize the BST by choosing one of Pre-order/In-order/Post-order traversals. You will find that deserialization of a tree do not results in construction of an exact clone tree on server machine (Infact it will result in a linked list). Knowing exactly one of Pre-order/In-order/Post-order traversal of a BST do not result in a unique BST.

Q6. Write the RPC to get the student details from Server

```
student_t *rpc_get_student_details (unsigned int rollno);
```

Assume, Server maintains a database of students

What Next ?

- In this course, we discussed in Detail about Data Serialization and De-Serialization
- We also witness, how RPC is heavily dependent on Data Serialization
- The Story is not yet over , but yes, the hard part is Gone !
  - In the rest of the course, We shall explore what more you can achieve using Data Serialization, that includes
    - Concept of Application State Transfer/Synchronization
    - Concept of Check-pointing
    - Auto (De) - Serializer's Code Generation

In Release 2 of this course ! Stay Tuned ! ☺

Thank you !

Remote Procedure Calls (RPC)

I have put in so much hard work for you ,
You should repay me back with your honest review !

Application
State
Synchronization

State Synchronization

- Application State Synchronization means, Syncing the Application State to a redundant machine running redundant backup instance of the application

- Application State Means the snapshot of all data structures kept in the memory
  - Heap Memory
  - Global Variables

- Application State keeps on changing – Data structures keeps updating with every instruction executed
  - Deletion
  - Addition
  - Update

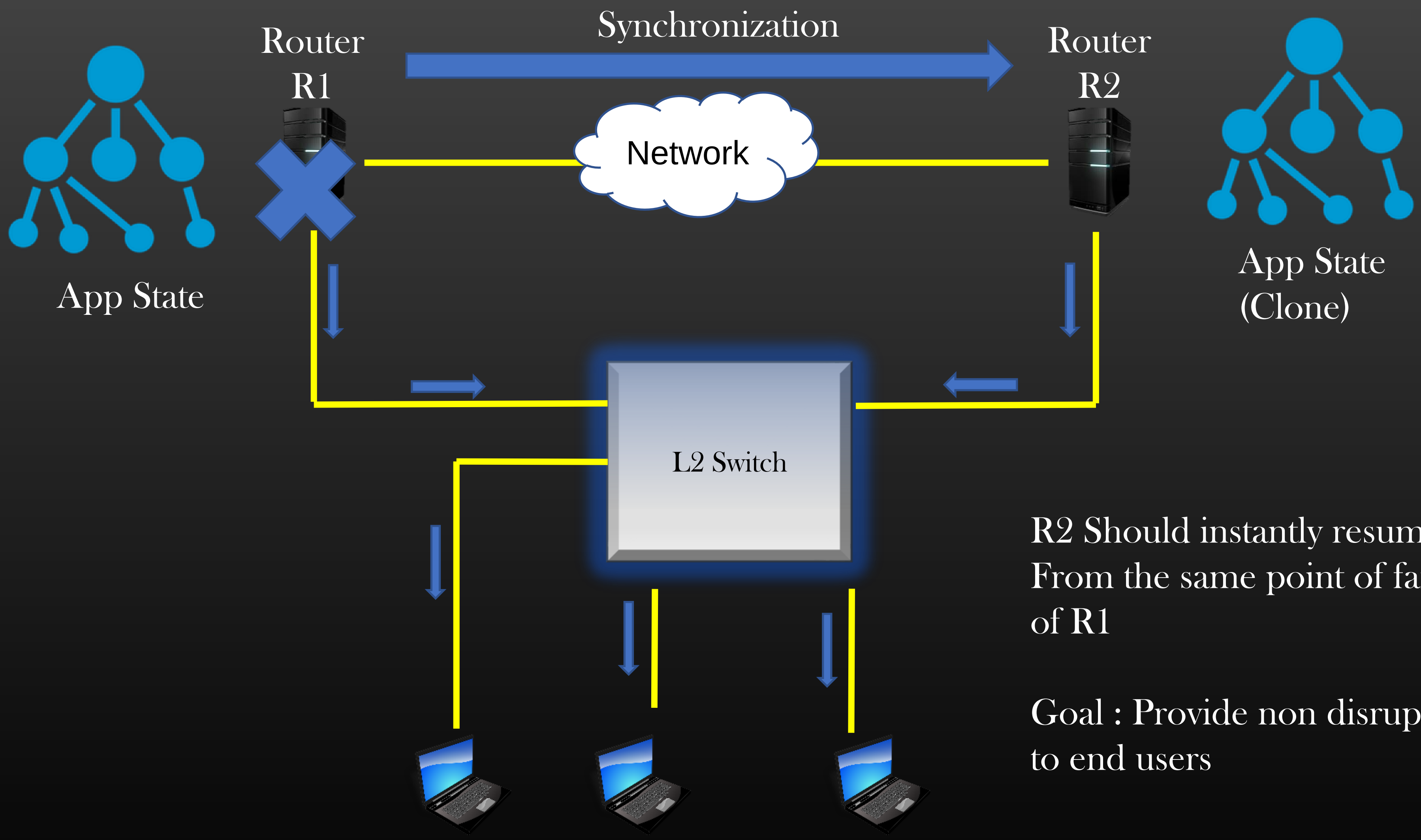
- It is desirable for Applications which needs to up and running 27x7 to have backup instance running on alternate machine
- In case the primary machine fails for any reason, back up instance should take over and continue to provide application service

- For example, Network equipment's such as Router disrupt many Network users when they go down. It is desirable to have a backup Router which takes over the Primary Router *instantly* when Primary router goes down

- Backup equipment must resume from the same state where primary equipment failed

State Synchronization

Practical Use case



R2 Should instantly resume
From the same point of failure
of R1

Goal : Provide non disruptive service
to end users

State Synchronization

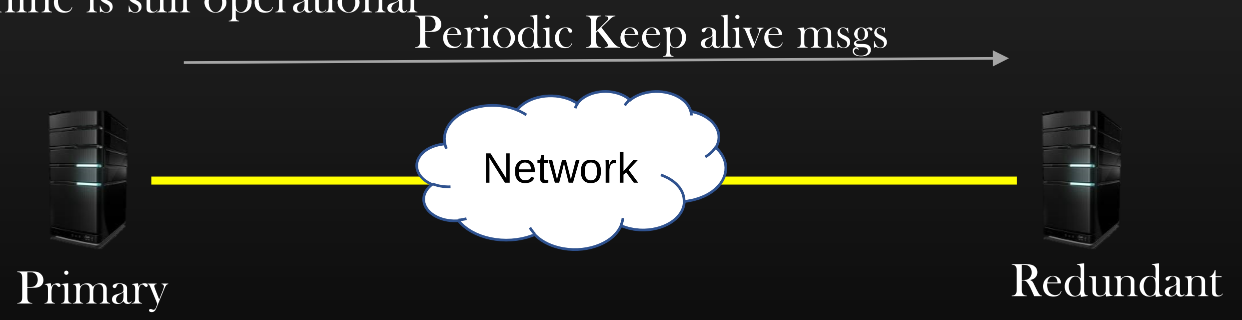
- We learnt, Receiving machine de-serialize the serialized data it has received and is able to reconstruct the clone of data structures

- We can use this concept to implement data redundancy

- The primary machine can choose to periodically synchronize its entire application state to redundant machine(s), may be, once in 10 min

- Redundant machine, refresh, its application state by rebuilding the data structures from serialized data received from primary machine

- Primary machine can send *keep alive* messages to redundant machine. Keep alive messages tells the redundant machine that primary machine is still operational



- If Redundant machine do not receive keep alive message for 5 seconds, Redundant machine can assume Primary machine has gone down, Redundant machine takes up the role of Primary and start delivering the service

Project on State Synchronization

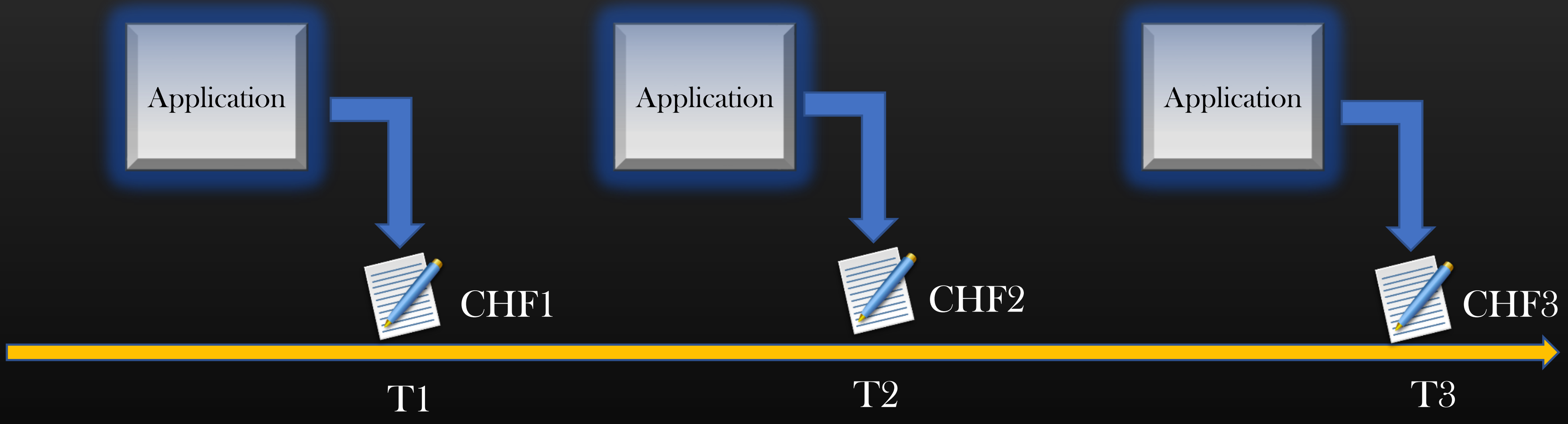
- In the Industry, this concept is popularly termed as *High Availability*
- You can convert any program/project you have done before and add HA feature to it
- We have all tools and knowledge required to do this project
  - Socket Programming
  - Data Serialization/DeSerialization
  - C Programming Or your choice !
- Your Application can have Various data structures
  - Linked Lists
  - Trees
  - Queues . . .etc
- Serialize all data structures -> Send Serialized Data to Redundant machine using Sockets
- Add necessary information in *Header* like size of serialized payload, Identity of data structure being sync'd
- Redundant machine must reconstruct the Data structure by performing deserialization, and flush off the old copy
- Redundant machine must switch its role to primary if it don't see *keep alive* msg for T seconds

Demonstration

Application
State
Check Pointing

Application State Check-pointing

- Check-pointing is the process of saving the application running state to persistent storage
- Application running state is the snapshot of data structures kept in the memory
  - Heap Memory
  - Global Data structures
- Once the application state is saved into persistent storage, Same application state can be retrieved back from the persistent storage at some later time



- Application State is saved in persistent storage, which can be retrieved whenever desired

Application State Check-pointing

Appln Save its State in CH File

Serialization of Data structures

Simply write the Serialized Buffer in a file

Appln Flush its current state, and load the state from CH file

De-Serialization of Data Structures

Simply read the serialized data from CH file and rebuild Objects

Let us discuss the implementation details regarding how check pointing can be done

Application State Check-pointing

Industry Application of Check-pointing

- In Industry, devices are configured with loads of configuration at any point of time – Whether Admin configured Or Dynamic
- Different Customers of Network equipment's have different requirements and configure the devices differently as per their needs
- It is important to save the equipment state persistently, so, that in future we can revert back the device state as desirable
- Even the windows OS comes with the feature of check-pointing if you explore !

Hope you Like the course, and you got to learn really some cool stuff !

Please leave a review if not already

If you have reached this slide, then probably you loved the course !

Thank you !!