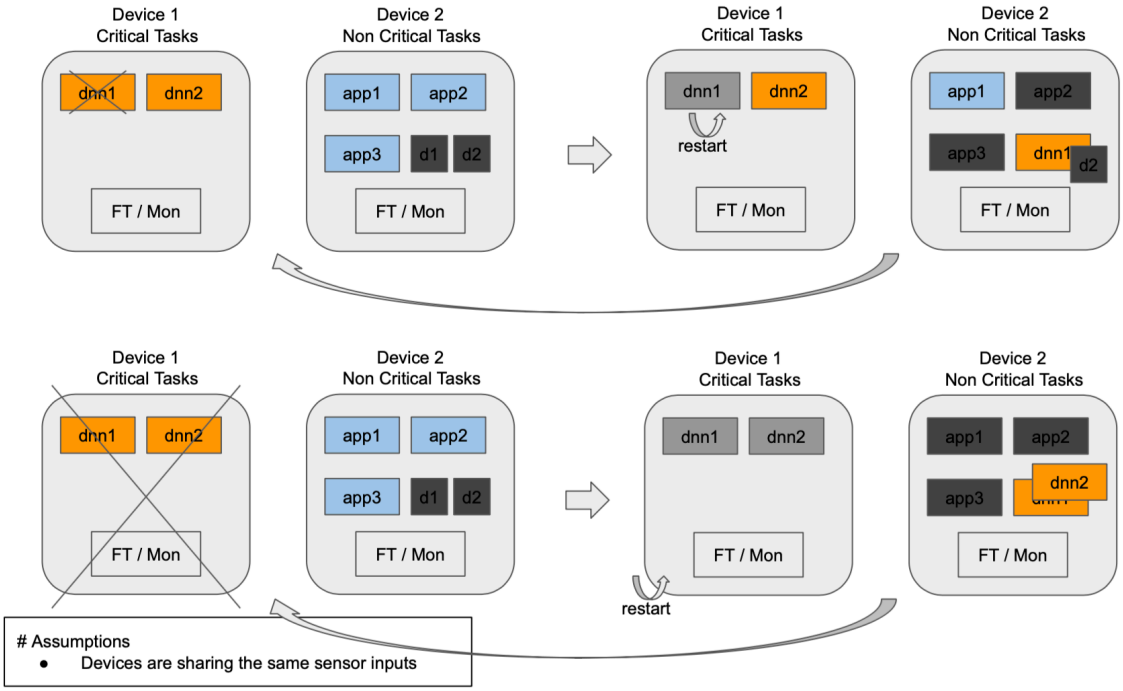
Fault-tolerant Scheme for Multiple Jobs on a Single Xavier

Motivation

FT manager for multiple devices

The initial target of this project is about FT mechanism for multiple HPC systems, i.e. Xaviers. It aims to develop a fault-tolerant manager that can effectively allocate hardware resources (multiple Xaviers in this case) for both significant and non-significant applications. In the design, these two different kinds of applications are running on two different Xaviers separately. One device will only have programs for significant applications, while the other device will have non-significant applications running in most of time and also have a copy of significant applications in the first device. This copy of applications will be in sleep mode if the first device works properly. Once the first device faces some unrecoverable issues which cause the significant applications to die, the replica of these applications on the second device will be activated with the halt on non-significant applications.

As shown in the following diagram from the project, FT mechanism for multiple HPC systems, there are two main cases. In both cases, device 1 is intended to run all critical tasks(dnn1, dnn2), while device 2 is responsible to all non-critical tasks(app1, app2, app3). In the first case(row 1 in the diagram), when one critical application running in device 1, dnn1, is accidentally died, device 1 will restart the dnn1 and device 2 will take over the task to run dnn1 with the cost of stopping non-critical tasks, app2 and app3. It should be noted that the d1 and d2 in the diagram below indicate the replica of dnn1 and dnn2. In the second case, when the entire system running critical applications is died, i.e. device 1 is died, it will be rebooted by fault-tolerance manager, while device 2 will be in charge of all critical tasks implementation and stop all the non-critical tasks if necessary.



For example, if we have a deep learning algorithm for imaging processing as a significant application running on device 1, at the meantime, we have some audio applications running as non-significant applications on device 2. The different scenarios are listed in the following table.

	Description	Device 1	Device 2
Scenario 1(normal)	Device 1 and 2 work as normal	Deep learning algorithm	Audio applications
Scenario 2	Device 1 is dead, device 2 works as normal	Dead	Suspend audio applications; Run Deep learning algorithm.
Scenario 3	Device 1 is rebooted, device 2 works as normal	Deep learning algorithm	Audio applications

Based on the above design, the fault-tolerant manager will be implemented as following:

1. Having heartbeat data to keep checking whether main, i.e. significant application, is died. Only when the main is alive, the main application will keep updating the **heartbeat** every **1ms.**
2. Having **checkpoint** data to help to restore the main application. When in scenario 1(normal), the main application will have a checkpoint every **10ms** with **400kB** data stored in a period of **100ms**. When device 1 is dead, device 2 will read from the most recent checkpoint stored by device 1 and activate the replica of the main application. This replica will continue store checkpoints until device 1 is recovered. Once device 1 is ready to run main again, it will start from the most recent checkpoint stored by the replica.

The table below summaries the implementation:

FT manager data	Update period	Size of data
Heart beat	Every 1ms	4 bytes(int)
Check point	Every 10 ms for each checkpoint; every 100ms for each round	400kB

FT manager in the middleware, CARSS

The goal of the project was modified later in this semester to have a similar FT manager work ing with a client-server middleware, CARSS. This middleware works on a single device and is able to handle up to **100** jobs. With the similar mechanism used to manage different types of client jobs to run on the device, a FT manager was developed to handle multiple client jobs and their replicas.

Assume we have a main application and its replica running on the device, the following table indicates the behaviour of them under different scenarios.

	Description	Device
Scenario 1(normal)	Main and replica are working	Main is running; replica is suspending
Scenario 2	Main is killed, replica is not killed	Replica, the copy of main, is running

	Description	Device
Scenario 3	Main is called again, replica is not killed	Replica is killed; Main is running.

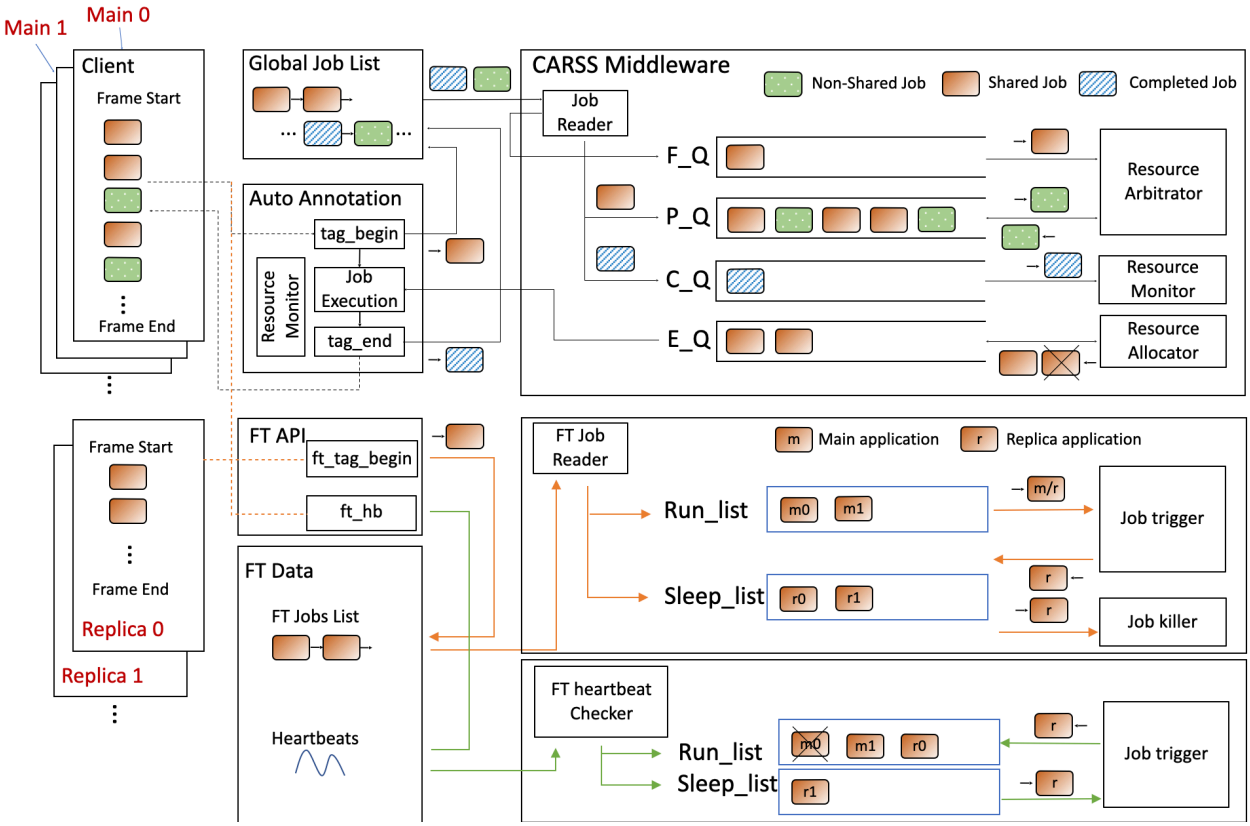
At the moment, the FT mechanism can handle up to **100** client mains and replicas. The details of implementation will be discussed in the following sections.

Architecture

In this section, the architecture of the developed FT-CARSS will be discussed.

FT diagram on top of CARSS

As shown in the following diagram, we have both main and replica programs running as client applications. The top half of the system, including 'Global job list', 'auto annotation' and 'job reader' etc, is about handling the issue of hardware availability when different kinds of applications are running. The bottom half the system describes the mechanism of the developed FT manager. In the diagram, the orange lines indicate the flow of ft jobs, while the green lines show how heartbeat data is updated and detected. The FT manager works both for the client and server.



On the client side, FT API provides an interface for clients to submit their corresponding job metadata, like PID and TID, to a ft-jobs list, which is a record of all existing jobs in the system, either running or suspending. Meanwhile, the FT API will also helps to allocate a new heartbeat region for a new client, where the client will keep updating its heartbeat data until it dies.

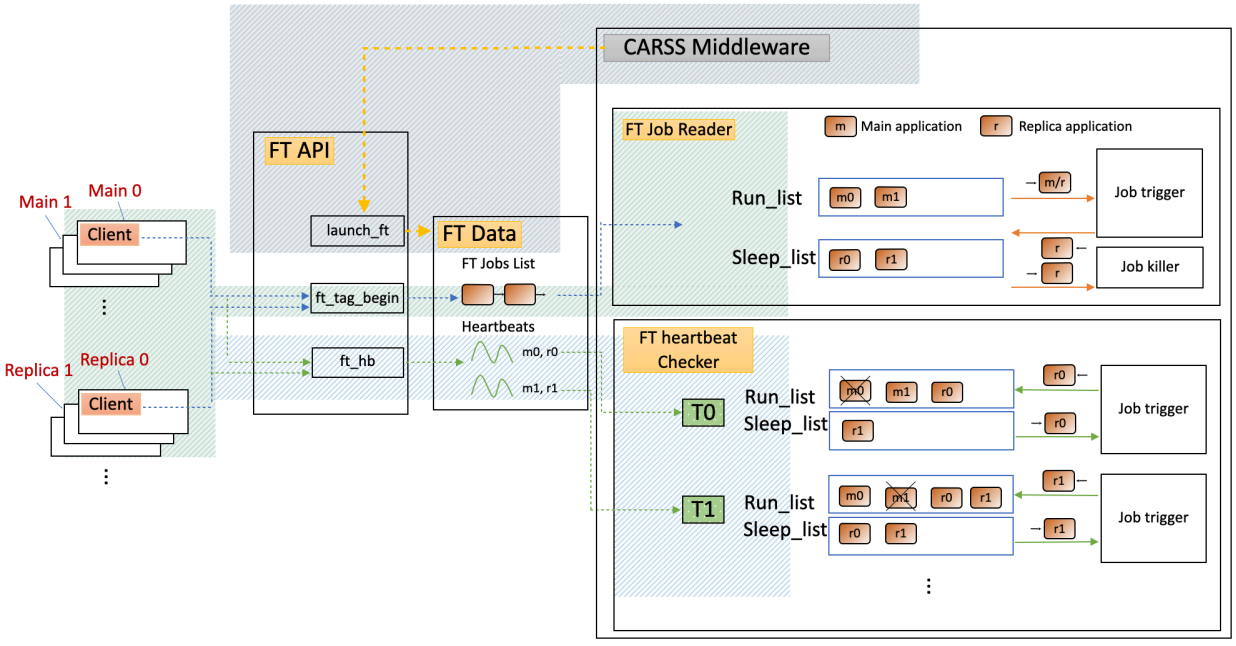
On the server side, FT API will be in charge of keeping tracking of the ft-jobs list to update two dynamic lists every 50us, i.e. run_list and sleep_list as shown in the diagram, through FT job reader. There are two rules followed. First, all the jobs in the run_list will be guaranteed to be triggered if it is main.Second, when a replica was waked up once, if it goes back to the sleep_list, it will be killed.

In terms of heartbeat, a FT heartbeat checker, including 100 threads by default, is able to check up to 100 heartbeats simultaneously every 1ms. When a main is dead, its replica will be trigged and moved to run_list in its corresponding thread.

FT Manager / Client (main / replica) and Heartbeat flow

The following diagram shows the details on how the FT manager communicates with both client and server. The functions highlighted in yellow are the main FT APIs used in the system.

First of all, as shown in the grey area, the server will call launch_ft in FT API to settle down shared memory regions for FT data, including ft-jobs list and heartbeats. Secondly, as shown in the green area, the client is able to submit its job to the shared ft-jobs list via FT API, ft_tag_begin. Meanwhile, the server will keep checking this ft-jobs list using FT job reader API and allocate a new job to a proper destination, either run_list or sleep_list. Besides, the client is able to update its heartbeat in a shared region via ft_hb function in FT API and the server has 100 threads by default to check up to 100 heartbeat data by FT heartbeat checker API.



Implementation

In this section, the details of data structure and functions in FT will be discussed.

FT functions

In the following three tables, the data structure, server utilities and client utilities are discussed.

FT data structure(ft_lib.h)	Description
ft_data_t	Contains heart beat data, heart_beat[FT_HB_DATA_MAX_DATA]
ft_job_t	Indicate each client job
ft_jobs_t	Indicate a list of client jobs, including both main and replica

FT server utilities(ft_utils_server.cpp)			Description
init_ft_jobs			Create a ft jobs list
launch_ft_man			Launch FT manager
	helper_thread [0]—> ft_jobs_thread		Keep checking ft jobs list, pass new coming client job to either running list or sleeping list. The structure of running and sleeping list is “unordered_map”. The key for each job is “main_<index>” or “replica_<index>”. The index comes from command line, which helps to differentiate different pairs of main and replica; Job trigger checks running list and triggers any new main applications; Job killer checks sleeping list and kills any executed replica, i.e. the replica that executed before and needs to be terminated due to a new coming main. Checking period: **50us**
	helper_thread [1 - 101]—> ft_hb_thread		Create 100 helper threads to check up to 100 clients’ heartbeats
		init_ft_data	Create a heartbeat region, heart_beat[FT_HB_DATA_MAX_DATA]. Each couple of main and replica has the same index to access the same area in the heart_beat array.

FT server utilities(ft_utils_server.cpp)			Description
		while(1)	Each heartbeat thread keep s checking the corresponding heartbeat value with the argument, index. E.g. curr_ftdata->heart_beat[index] Checking period: **1ms**

FT client utilities(ft_utils_client.c)			Description
ft_init_wait			Submit the client job to ft jobs list; Start keeping updating the correspond heartbeat.
	tag_ft_job_begin		Submit the client job with name, jobname_<pid>; Wait with a semaphore until job trigger wakes up it
	init_ft		Create a helper thread to keep updating its heartbeat, specified by the argument, index, coming from command line
		helper_thread->heartbeat_thread While(1)	Update its heartbeat every 1ms. E.g. client_FT_data->heart_beat[index] ++; Checking period: **1ms**

Demonstration

Command lines example to run c and python client	
Format	Program_name <type> <number> <type> : main, replica <number>: 0 ~ 99
C example	./main_c.o main 0 ./main_c.o replica 0
Python example	Python3 main_py.py main 1 Python3 main_py.py replica 1

It should be noted that there is an assumption made when running the demo. When the main is going to be killed, there must be a replica alive in the system. The current FT manager doesn't consider the case when the main is killed without a replica existing in the system.

Future work

Checkpoint

The following two figures shows the logic of a basic checkpoint mechanism with dummy layer functions(mimicking the functions in a neural network).

As shown in the first figure, variable x indicates the return data from a dummy layer function, i.e add_1, add_2 or add_3. Right after calling each layer function, the return data, x, is stored as a checkpoint via function, update_checkpoint.

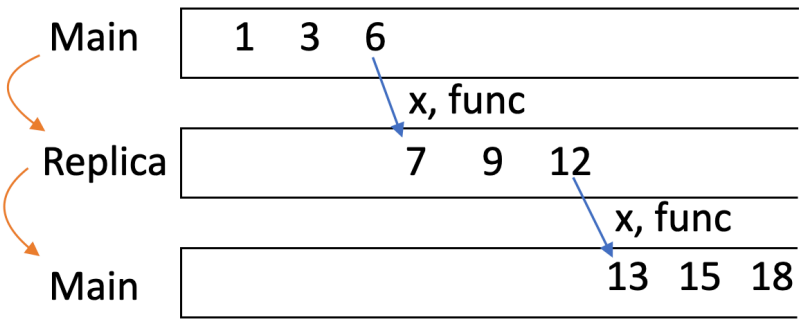
Figure 2 shows the performance of the checkpoint mechanism. First, x is initialized as 0, and main is processing with it, producing data, 1, 3, 6. Then main is killed and replica is waked up, and based on the current checkpoint, it starts from 6 and calls function add_1(also recored in checkpoint), which produce the subsequent value, 7. Finally, the main is called again which kills the running replica and runs with the current checkpoint data, i.e. 12.

In the future, this work can be used as a basic idea for a real checkpoint implementation.

```
x = add_1(x);  # mimic layer 0
print(x , end = '\n')
update_checkpoint(x, 1); #  ---> checkpoint 1
time.sleep(1)
x = add_2(x);  # mimic layer 1
print(x , end = '\n')
update_checkpoint(x, 2); #  ---> checkpoint 2
time.sleep(1)
x = add_3(x);  # mimic layer 2
print(x , end = '\n')
update_checkpoint(x, 3); #  ---> checkpoint 3
time.sleep(1)
print(" ")
```

x: 1, 3, 6, 7, 9, 12, 13, 15, 18

func: add_1, add_2, add_3



Reference

1. Paper for multiple Xaviers idea

CoLoR: Co-Located Rescuers for Fault Tolerance in HPC Systems

2. Paper for CARSS

CARSS: Client-Aware Resource Sharing and Scheduling for Heterogeneous Applications

3. Paper for checkpoint

Error Vulnerabilities and Fault Recovery in Deep-Learning Frameworks for Hardware Accelerators

Appendix I

Control of the number of helper threads for heartbeat created in server

The number of threads created on the server side to keep checking each heartbeat data is defined as shown in the table below.

ft_lib.h	
#define FT_HB_DATA_MAX_DATA	**100**

Appendix II

FT utilities in Python

The following table shows how the c and c++ FT functions are imported and used in a python program. First, in the makefile, **libft.so** is created which includes both ft_utils.cpp and ft_utils_client.c. Then in a python program, the library is loaded as **libft**, through which the functions, tag_job_begin, tag_job_end and ft_init_wait, are modified to fit in python environment.

	Lines	Description
Makefile	ft_server_lib.o: ft_utils_server.cpp $(CXX) $(INCL_FLAGS) $(CPP_FLAGS) $(MIDFLAGS) -o ft_server_lib.o ft_utils_server.cpp -c -fPIC	ft_utils.cpp and ft_utils_client.c are compiled to ft_lib.o and ft_client_lib.o, which are wrapped to **libft.so** used in python program
	ft_client_lib.o: ft_utils_client.c $(GCC) $(INCL_FLAGS) $(MIDFLAGS) -o ft_client_lib.o ft_utils_client.c -c -fPIC	
	libft.so: tag_state.o tag_lib.o tag_frame.o libmid.so ft_server_lib.o ft_client_lib.o $(EDIT_LD_PATH) $(CXX) -shared -o lib/libft.so tag_state.o tag_lib.o tag_frame.o ft_server_lib.o ft_client_lib.o $(LOAD_MID)	

	Lines	Description
Python client	from ctypes import cdll, c_uint, c_int, c_void_p, c_char_p, c_ulonglong, c_double, c_longlong, c_bool	Based on tag_layer.py and tag_layer_fps.py in python folder, a new library, **libft** is added in python client directly in order to use the utilities defined in ft_utils.cpp and ft_utils_client.c. In a python program, there are three FT APIs used, **tag_job_begin**, **tag_job_end**, and **ft_init_wait**.
	###### Create interface for tag functions ######	
	libc = cdll.LoadLibrary('libc.so.6')	
	libft = cdll.LoadLibrary("/home/ruiyingw/rw_work/cuMiddleware/lib/libft.so")	
	###### Modify the res and argtypes of tag function interface ######	
	libft.tag_job_begin.restype = c_int	
	libft.tag_job_begin.argtypes = [c_uint, c_uint, c_char_p, c_longlong, c_bool, c_bool, c_ulonglong]	
	libft.tag_job_end.restype = c_int	
	libft.tag_job_end.argtypes = [c_uint, c_uint, c_char_p]	
	libft.ft_init_wait.restype = c_int	
	libft.ft_init_wait.argtypes = [c_uint, c_uint, c_char_p, c_int]	

Appendix III

Hardware platform: JETSON AGX XAVIER

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU, 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x | 137GB/s
Storage	32GB eMMC 5.1
DL Accelerator	(2x) NVDLA Engines
Vision Accelerator	7-way VLIW Vision Processor
Encoder/Decoder	(2x) 4Kp60 | HEVC/(2x) 4Kp60 | 12-Bit Support
Size	105 mm x 105 mm x 65 mm
Deployment	Module (Jetson AGX Xavier)