文本框

形状

Executor in SuperScaler

Specifications

Date: Jun 16, 2020   
Version: V0.1

**RESTRICTED**

THIS DOCUMENT AND ANY ASSOCIATED DATA CONTAIN RESTRICTED INFORMATION THAT IS MICROSOFT PROPERTY. DO NOT DISCLOSE TO OR DUPLICATE FOR OTHERS EXCEPT AS AUTHORIZED BY MICROSOFT CORPORATION.

(c) 2018 Microsoft Corporation.  All rights reserved.

This document is provided "AS-IS" with no warranties. Information and views expressed in this document may change without notice. You bear the risk of using it.

Microsoft and/or third parties may have intellectual property rights covering the subject matter in this document.  Except as may be expressly provided in a separate agreement, if any, the furnishing of this document does not grant any licenses to any such intellectual property rights.  Instead, this document is for your internal reference purposes.  This document and its contents are confidential and proprietary to Microsoft.  It is disclosed and can be used only pursuant to a non-disclosure agreement between you and Microsoft. **Microsoft Corporation Technical Documentation License Agreement (Standard)**   
**READ THIS!** THIS IS A LEGAL AGREEMENT BETWEEN MICROSOFT CORPORATION ("MICROSOFT") AND THE RECIPIENT OF THESE MATERIALS, WHETHER AN INDIVIDUAL OR AN ENTITY ("YOU"). IF YOU HAVE ACCESSED THIS AGREEMENT IN THE PROCESS OF DOWNLOADING MATERIALS ("MATERIALS") FROM A MICROSOFT WEB SITE, BY CLICKING "I ACCEPT", DOWNLOADING, USING OR PROVIDING FEEDBACK ON THE MATERIALS, YOU AGREE TO THESE TERMS. IF THIS AGREEMENT IS ATTACHED TO MATERIALS, BY ACCESSING, USING OR PROVIDING FEEDBACK ON THE ATTACHED MATERIALS, YOU AGREE TO THESE TERMS.   
   
1. For good and valuable consideration, the receipt and sufficiency of which are acknowledged, You and Microsoft agree as follows:   
   
(a) If You are an authorized representative of the corporation or other entity designated below ("**Company**"), and such Company has executed a Microsoft Corporation Non-Disclosure Agreement that is not limited to a specific subject matter or event ("**Microsoft NDA**"), You represent that You have authority to act on behalf of Company and agree that the Confidential Information, as defined in the Microsoft NDA, is subject to the terms and conditions of the Microsoft NDA and that Company will treat the Confidential Information accordingly;   
   
(b) If You are an individual, and have executed a Microsoft NDA, You agree that the Confidential Information, as defined in the Microsoft NDA, is subject to the terms and conditions of the Microsoft NDA and that You will treat the Confidential Information accordingly; or   
   
(c)If a Microsoft NDA has not been executed, You (if You are an individual), or Company (if You are an authorized representative of Company), as applicable, agrees: (a) to refrain from disclosing or distributing the Confidential Information to any third party for five (5) years from the date of disclosure of the Confidential Information by Microsoft to Company/You; (b) to refrain from reproducing or summarizing the Confidential Information; and (c) to take reasonable security precautions, at least as great as the precautions it takes to protect its own confidential information, but no less than reasonable care, to keep confidential the Confidential Information. You/Company, however, may disclose Confidential Information in accordance with a judicial or other governmental order, provided You/Company either (i) gives Microsoft reasonable notice prior to such disclosure and to allow Microsoft a reasonable opportunity to seek a protective order or equivalent, or (ii) obtains written assurance from the applicable judicial or governmental entity that it will afford the Confidential Information the highest level of protection afforded under applicable law or regulation. Confidential Information shall not include any information, however designated, that: (i) is or subsequently becomes publicly available without Your/Company’s breach of any obligation owed to Microsoft; (ii) became known to You/Company prior to Microsoft’s disclosure of such information to You/Company pursuant to the terms of this Agreement; (iii) became known to You/Company from a source other than Microsoft other than by the breach of an obligation of confidentiality owed to Microsoft; or (iv) is independently developed by You/Company. For purposes of this paragraph, "Confidential Information" means nonpublic information that Microsoft designates as being confidential or which, under the circumstances surrounding disclosure ought to be treated as confidential by Recipient. "Confidential Information" includes, without limitation, information in tangible or intangible form relating to and/or including released or unreleased Microsoft software or hardware products, the marketing or promotion of any Microsoft product, Microsoft's business policies or practices, and information received from others that Microsoft is obligated to treat as confidential.    
   
2. You may review these Materials only (a) as a reference to assist You in planning and designing Your product, service or technology ("Product") to interface with a Microsoft Product as described in these Materials; and (b) to provide feedback on these Materials to Microsoft. All other rights are retained by Microsoft; this agreement does not give You rights under any Microsoft patents. You may not (i) duplicate any part of these Materials, (ii) remove this agreement or any notices from these Materials, or (iii) give any part of these Materials, or assign or otherwise provide Your rights under this agreement, to anyone else.   
   
3. These Materials may contain preliminary information or inaccuracies, and may not correctly represent any associated Microsoft Product as commercially released. All Materials are provided entirely "AS IS." To the extent permitted by law, MICROSOFT MAKES NO WARRANTY OF ANY KIND, DISCLAIMS ALL EXPRESS, IMPLIED AND STATUTORY WARRANTIES, AND ASSUMES NO LIABILITY TO YOU FOR ANY DAMAGES OF ANY TYPE IN CONNECTION WITH THESE MATERIALS OR ANY INTELLECTUAL PROPERTY IN THEM.   
   
4. If You are an entity and (a) merge into another entity or (b) a controlling ownership interest in You changes, Your right to use these Materials automatically terminates and You must destroy them.   
   
5. You have no obligation to give Microsoft any suggestions, comments or other feedback ("Feedback") relating to these Materials. However, any Feedback you voluntarily provide may be used in Microsoft Products and related specifications or other documentation (collectively, "Microsoft Offerings") which in turn may be relied upon by other third parties to develop their own Products. Accordingly, if You do give Microsoft Feedback on any version of these Materials or the Microsoft Offerings to which they apply, You agree: (a) Microsoft may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any Microsoft Offering; (b) You also grant third parties, without charge, only those patent rights necessary to enable other Products to use or interface with any specific parts of a Microsoft Product that incorporate Your Feedback; and (c) You will not give Microsoft any Feedback (i) that You have reason to believe is subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any Microsoft Offering incorporating or derived from such Feedback, or other Microsoft intellectual property, to be licensed to or otherwise shared with any third party.   
   
6. Microsoft has no obligation to maintain confidentiality of any Microsoft Offering, but otherwise the confidentiality of Your Feedback, including Your identity as the source of such Feedback, is governed by Your NDA.   
   
7. This agreement is governed by the laws of the State of Washington. Any dispute involving it must be brought in the federal or state superior courts located in King County, Washington, and You waive any defenses allowing the dispute to be litigated elsewhere. If there is litigation, the losing party must pay the other party’s reasonable attorneys’ fees, costs and other expenses. If any part of this agreement is unenforceable, it will be considered modified to the extent necessary to make it enforceable, and the remainder shall continue in effect. This agreement is the entire agreement between You and Microsoft concerning these Materials; it may be changed only by a written document signed by both You and Microsoft.

**Revision History**

|  |  |  |
| --- | --- | --- |
| **Revision/  Date** | **Notes** | **Updated by** |
| V0.1Jun/16/2020 | Working in Process | Wenhao Shi |
| V0.2  Sep/22/2020 | Add detailed Executor design and implementation | Xuhao Luo |

Table of Contents

[Table of Contents 4](#_Toc51699098)

[1. Introduction 5](#_Toc51699099)

[2. Design 5](#_Toc51699100)

[2.1 Executor’s Design 5](#_Toc51699101)

[2.1.1 Create Task 5](#_Toc51699102)

[2.1.2 Input 6](#_Toc51699103)

[2.1.3 Output 6](#_Toc51699104)

[2.1.4 Task Manager 6](#_Toc51699105)

[2.1.5 Task Scheduler 7](#_Toc51699106)

[2.1.6 Worker Scheduler 8](#_Toc51699107)

[2.1.7 Worker 8](#_Toc51699108)

[2.1.8 Event queue and polling thread 8](#_Toc51699109)

[2.2 Channel Design 8](#_Toc51699110)

[2.2.1 Interface 9](#_Toc51699111)

[2.2.2 Cuda Channel 10](#_Toc51699112)

[2.2.3 RDMA Channel 11](#_Toc51699113)

[3. Implementation 12](#_Toc51699114)

[3.1 Task Scheduler 12](#_Toc51699115)

[3.1.1 Dependency Graph 13](#_Toc51699116)

[3.1.2 Runnable Queue 13](#_Toc51699117)

[3.2 Worker Scheduler 13](#_Toc51699118)

[3.2.1 Schedule Policy 13](#_Toc51699119)

[3.3 Worker 13](#_Toc51699120)

[3.4 Cuda Channel 14](#_Toc51699121)

[3.4.1 Connection 14](#_Toc51699122)

[3.4.2 Shared Memory 14](#_Toc51699123)

[3.4.3 Lock free FIFO 15](#_Toc51699124)

[3.4.4 Data transfer 15](#_Toc51699125)

[3.5 RDMA Channel 15](#_Toc51699126)

[3.5.1 Connection 15](#_Toc51699127)

[3.5.2 RDMA FIFO 16](#_Toc51699128)

# Introduction

This document describes the design and implementation of Executor, which is a module of SuperScaler project.

# Design

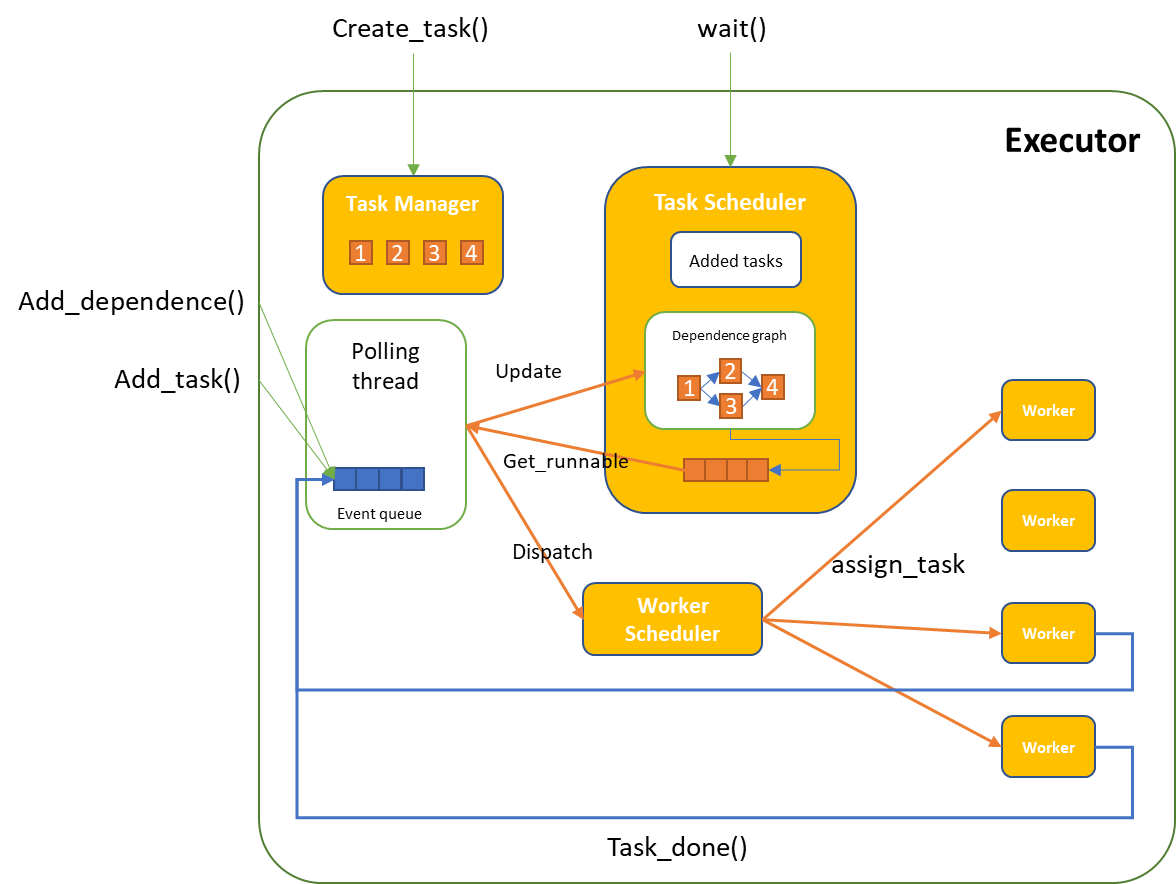


Figure 2‑1 Executor's Design

## Executor’s Design

As shown in Figure 2‑1, executor is formed by Task Manager, Tasks Scheduler, Worker Scheduler and Workers. An event queue and a polling thread sits on top of these components. It uses task info to create Tasks and takes Task and Task dependence as the input and output Execution Info. In most of tasks, executor will communicate with some data.

### Create Task

1. task\_id\_t CreateTask<TaskType>(TaskInfo…);

The Task Manager creates and manages all the Tasks. The info needs to create a Task includes:

* Task Type
* Call back function
* Data information

The task type tells the task scheduler which type of task to create. Data type includes communication task and computation task. Communication tasks are tasks about data copy, such as cuda tasks and RDMA tasks. Computation tasks are tasks include computation, add task for example. We pay attention to communication tasks at the first stage.

The Task manager will return a Task ID to the user. Task ID will be unique for unfinished tasks at least in the same process.

Call back function will be called after task finished. There will be a call back engine to execute call back functions in the future.

Data information including the pointer to data and its length. In most cases, the data is cuda data.

### Input

1. Bool add\_task(task\_id\_t task\_id);
2. Bool add\_dependence(task\_id\_t who, task\_id\_t whom);

The input of executor is the task and the dependency graph of Tasks. *Add\_task* adds a created task into the executor to execute. *Add\_dependence* adds a dependency between two Tasks.

### Output

1. ExecInfo wait(task\_id\_t task\_id);
2. ExecInfo wait();

The output of executor is execution info. The execution info has two forms: synchronous one and asynchronous one.

The synchronous one is call Task Schedule’s *wait* function synchronously, it will block until the task is finished (either success or failed). There are two interfaces for *wait*, differ in whether to specify a specific Task to wait or not. The first one will block until the specified task is finished. The second will block until any one task is finished. Regardless which interface to use, the user must get the execution info for every task, which marks the end of the task’s lifecycle. Otherwise the task will always stay in Task Manager.

The asynchronous one indicates the callback function. The callback function is the one in Task Info. If will be called when task finished.

### Task Manager

Task Manager manages the lifecycle of all tasks. It creates tasks from the given task info delete tasks when they are finished. To the user, it only exposes the Task ID to prevent user from manipulate Task directly. To other executor components, it offers interface to get the pointer to task from Task ID.

Interfaces:

Task\_id\_t create\_task<TaskType>(TaskInfo…);

Create a Task of *TaskType* using the *TaskInfo* provided. Return a unique Task ID to user.

Task \*get\_task(task\_id\_t task\_id);

Get the pointer to the Task with the specified Task ID.

Bool delete\_task(task\_id\_t task\_id);

Delete the specified task from Task manager.

### Task Scheduler

Task scheduler has three main functions:

1. Task Scheduler stores the dependency relationship of all tasks and whether a task has been added for execution.
2. Task scheduler solves tasks’ dependence, one task can only be executed when it has been added for execution and all its dependences have been executed successfully. While a cyclic dependency is found, Task Scheduler will raise an error.
3. Task scheduler gets tasks’ execution result and return them to the caller, both synchronously and asynchronously. Once the task scheduler notices the caller the result synchronously, the executor will call Task Manager to delete the task.

Interfaces:

Void add\_task(task\_id\_t task\_id);

Add the specified Task into the dependency graph.

Void add\_dependence(task\_id\_t who, task\_id\_t whom);

Add a dependency that Task *who* depends on Task *whom* into the dependency graph.

Task \*get\_runnable();

Get an executable task from the dependency graph.

Bool task\_done(task\_id\_t task\_id);

Solve dependence in the dependency graph related with the specified Task.

ExecInfo wait();

Block to wait for one task to finish and get its execution result.

ExecInfo wait(task\_id\_t task\_id);

Block to wait for the specified task to finish and get its execution result.

### Worker Scheduler

Worker scheduler schedule workers execute tasks as quick as possible.

Worker scheduler get tasks from task scheduler without dependence. That is to say, all tasks that worker scheduler get can be executed directly and parallelly.

Worker scheduler manages workers. A worker means a working thread can execute tasks.

Worker scheduler should also find stop abnormal workers. When a task has been executed too long time, worker scheduler should stop the worker and reschedule the task.

When one task is finished, the worker scheduler should report the result to task scheduler. Then the task scheduler will solve dependence related and user can get the task’s execution result.

Interface:

Bool dispatch\_task(Task \*task);

Dispatch a task to Worker Scheduler to schedule.

### Worker

Basically, worker means a working thread executing tasks. Besides, it has some components supporting some kind of tasks.

There are two mean tasks, as mentioned before. Communication tasks and computation tasks. The communication tasks need to transfer data between processes or servers. Thus, channels are needed, to help worker transfer data. Cuda channel is used to transfer cuda data between processes. RDMA channel is used to transfer data between servers.

Computation tasks usually performs computation on GPU. It uses the specified computation kernel to perform computation on the data the user provides in the Task.

Interface:

Void add\_task(Task \*task);

Add a task to worker to execute.

### Event queue and polling thread

The event queue is used to serialize all operations to dependency graph to avoid locks on the dependency graph. There are three kinds of events related to the dependency graph, the *add task* event, the *add dependence* event, and the *task done* event. The polling thread will poll the events from the event queue and use Task Scheduler’s API to perform operation on dependency on the dependency graph accordingly.

## Channel Design

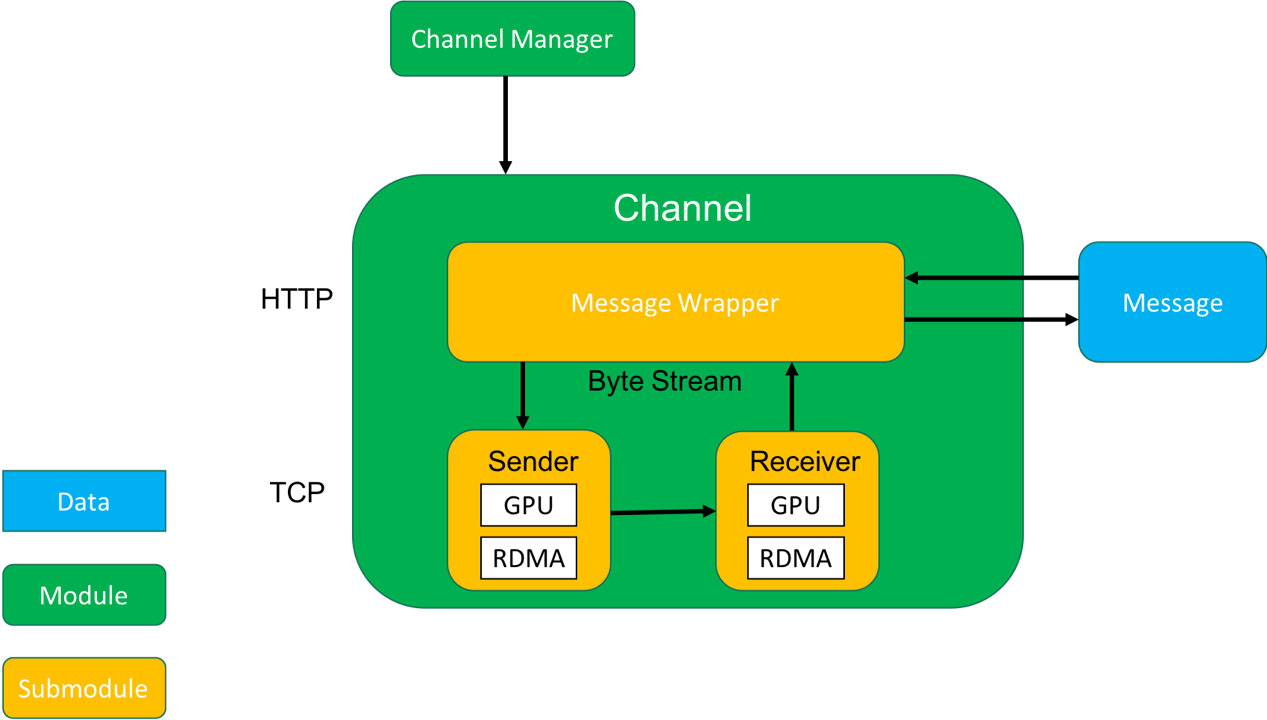


Figure 2‑2 Channel Design

As shown in the figure, we design two modules to send data. Here are the definitions and functions of modules and submodules:

* Channel Manager: Create, get and release channel by channel’s unique name. Should be singleton.
* Message: The data to be sent
* Channel: Module support message transfer between different device
  + Message Wrapper: Generate and parse control header for messages, fit messages into byte stream. Message wrapper write data to sender and receiver data from receiver.
  + Sender: Open connection to receiver, and send byte stream to receiver’s buffer.
  + Receiver: Listen sender’s connection and receive data from sender

Sender and receiver together offer a reliable byte stream, like TCP.

Message Wrapper use this reliable byte stream to transmit messages, works like HTTP.

### Interface

This section shows the interfaces of cuda channel’s sender and receiver

|  |  |  |
| --- | --- | --- |
| Type | Name | Describe |
| Enum | task\_state | Shows the task execution state: success, failed, wait, etc |
| Function | send(message\_id, void\* cuda\_data); | Sender’s interface. Send data to receiver |
| Function | receive(message\_id, void\* cuda\_buffer); | Receiver’s interface. Try to receive a message from sender |
| Function | wait() | Only for receiver in solution B. Wait for unfinished message |

### Cuda Channel

#### Usage

Cuda channel is a channel used coping data between different GPU and different processes. The channel is designed to be created when program start. And it will support all cuda data transfer between these two processes.

The channel design support dynamic creates and destroy, but this is not suggested.

#### Design

We can separate data into two parts: control plane and data plane. Control plane contains control data generated by message wrapper. Data plane contains message data. In cuda channel, we use different method to transfer these two types of data.

We also separate transfer process into two part: establishing a connection and data transfer. The cuda channel should establish connection first, then can transfer data.

We finally decide using named semaphore to establish connection, use shared memory to support control plan and use cudaIpcMemHandler support data plan.

#### Process

The sender and receiver use the shared memory to transmit control head. The shared memory contains two FIFOs, one is for sender and one is for receiver. The sender can only write to sender’s FIFO and read from receiver’s FIFO. The receiver can only write to receiver’s FIFO and read from sender’s FIFO.

Firstly, the receiver pushes the destination pointer in the form of cudaIpcHandle to the receiver’s buffer. Then the sender will receive the handler, run data transmission, and then push an ACK to the sender’s FIFO tells receiver the transmission is finished.

For the sender, when sending a message, the sender will read all messages from receiver’s FIFO and push them to a hash map index by message id. If the sender can find the message id in the hash map, it means the receiver is papered and the sender can send the message. If the sender cannot find the message id in the hash map, the sender should set current sending task to wait status and try it again latter. After the message transmission, the sender should release all resources and send the messages.

|  |
| --- |
| 1. func Sender(message\_id) 2. { 3. **while** (!receiver\_fifo.empty()) 4. { 5. message\_meta = receiver\_fifo.pop(); 6. hash\_map.insert(message\_meta.id, message\_meta); //insert (key, value) 7. } 8. **if** (!hash\_map.find(message\_id)) 9. { 10. **return** wait; // Receiver not parpered 11. } 12. data\_transfer(); 13. sender\_fifo.push(message\_id); 14. **return** success; 15. } |

Code 1 Sender’s process

For the receiver, when tries to receive a message, the receiver will send the cudaIpcHandle of the destination to the receiver’s FIFO. And all the transmission will be done by the sender. The receiver periodically checks the sender’s FIFO, and tell these receiving tasks have been finished and release resources.

|  |
| --- |
| 1. func Receiver(message\_meta) 2. { 3. receiver\_fifo.push(message\_meta); 4. **return** wait; 5. } 7. func ReceiverWait(message\_id) 8. { 9. **while** (!sender\_fifo.empty()) 10. { 11. message\_id = sender\_fifo.pop(); 12. set\_success(message\_id); 13. } 14. } |

Code 2 Receiver's process

### RDMA Channel

The design of RDMA Channel is similar to Cuda Channel.

#### Usage

RDMA Channel should be established when there is requirement for communication between these two devices. It should be shut down when can make sure there is no more communication requirement between these two devices. Establish connection at the beginning of program is suggested. Because it will take some time to establish the connection, establishing connections repeatedly will lead to low performance.

#### Design

Similar to cuda channel. RDMA channel separate into two part: control plan and data plan. The control plan including control information such as message id, the key to data and so on. The data plan indicates the data need to be transferred.

By the way, RDMA Channel do not have connection issue like Cuda channel. Network does have connection, and does the same job.

RDMA have two kind of transformation: send/receive and read/write. The read and write operations is much faster which should be used in our transformation.

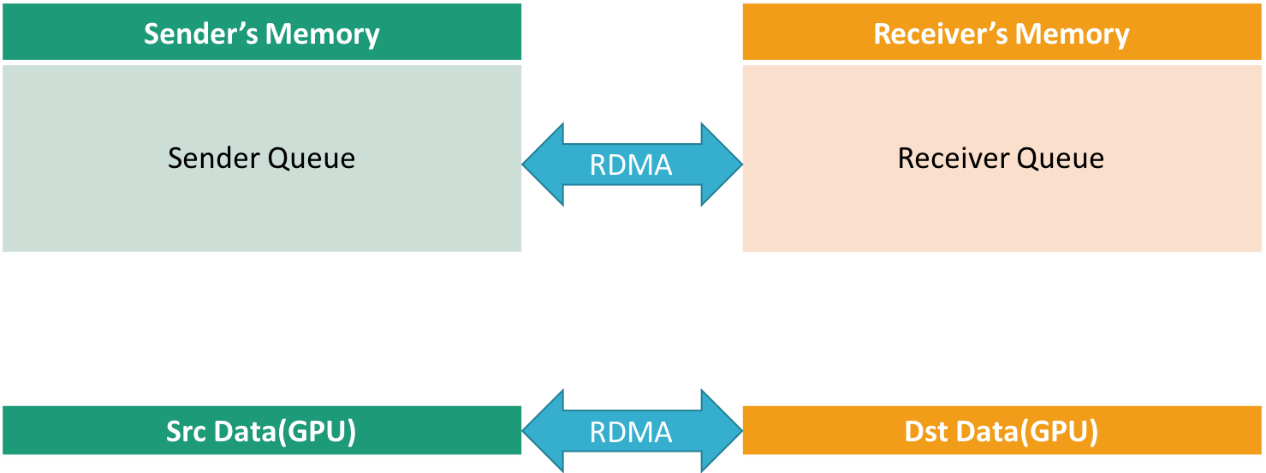


Figure 2‑3 RDMA Design

However, read and write operations need to know the key and address of peer. Each side hold a queue in its memory. Firstly, they swap the remote key RDMA needed with each other. Then both sides can get access to other’s queue by RDMA. Both sides can write to other’s queue and read from its queue and thus, the control message channel is established. The RDMA sender and receiver can use the control channel translate metadata such as remote key of message. With the metadata, RDMA channel can transfer messages in GPU.

#### Process

To transfer a message, firstly, the receiver will get the key to the receive buffer and send it with the message id to the sender queue by RDMA. Then the sender will transfer the data by RDMA’s write. And send an ACK to the receiver’s queue. At this point, the send task is finished successfully. After the receiver get the ACK, the receiver task is done successfully.

For receiver, the receive method get and send the control data to the sender’s queue. After that, it returns true. However, the receive task is not finished yet. The receiver should periodically check its queue and find the task is finished or not.

The sender’s work is simple, when get a send task, it checks its queue and push all received metadata to a hash map. Then find the metadata of the send task in the hash map. If there is metadata belongs to this task, the sender should transfer the data. Otherwise, the send is failed because the receiver is not prepared. Sender’s caller can retry this until receiver is prepared.

# Implementation

This section tells some detail of implementation.

## Task Scheduler

### Dependency Graph

The dependency graph is implemented with two hash table as bellow:

1. std::unordered\_map<task\_id\_t, std::unordered\_set<task\_id\_t> > m\_dependences;
2. std::unordered\_map<task\_id\_t, size\_t> m\_dependence\_count;

The first hash map maps the ID of a task to the IDs of the tasks that depends on it and the second maps the ID of a task to the number of tasks it depends on. Whenever the task scheduler is informed that a task is finished, we can find all tasks that depend on it and decrease their dependence count by 1. While a task’s dependence count decrease to 0, we will add the ID of the task to a queue, say *runnable queue*, which means these tasks have all of their dependency solved and are ready for run. Then, the entry of the finished task in both hash maps will be deleted, and its execution result will be generated and stored in the task scheduler for user to fetch.

### Runnable Queue

The polling thread will fetch runnable tasks from the queue via the *get\_runnable* call. However, only tasks have been added into the executor can be sent to executor. The *get\_runnable* call will first check if the task has been added. If true, the task will be removed from the queue and the pointer to that task will be returned, else it will remain in the queue.

The runnable queue is used to avoid searching through the dependency graph every time we fetch a task. Whenever the queue is not empty, *get\_runnable* directly fetch task from the queue. Else, *get\_runnable* will first perform a sync to search through the dependence graph and put runnable task into the queue, which usually happens at the beginning of executing a graph.

## Worker Scheduler

### Schedule Policy

Currently we use a simple greedy based policy to schedule task onto Workers. The worker scheduler has a max limit of the number of workers it can manage. We partition the workers into two groups, the idle group and the busy group. At the start of the program we have 0 worker running. When a new task is given to worker scheduler, if the idle group is not empty, we take a worker from it to run that task and put the worker into busy group. Else, we first check if the number of workers have reach the worker limit, if not, we spawn a new worker directly, else, we choose the worker with the minimum work load in the busy group and assign the task to it. Workers will be moved between idle group and busy group.

**TODO**: Need adjustment to schedule policy to better suit the real scenario. E.g., for an allreduce execution plan, as we now use a synchronous send/recv, in every round of data transmission, the send and recv task can’t be scheduled on the same worker, otherwise will result in a deadlock.

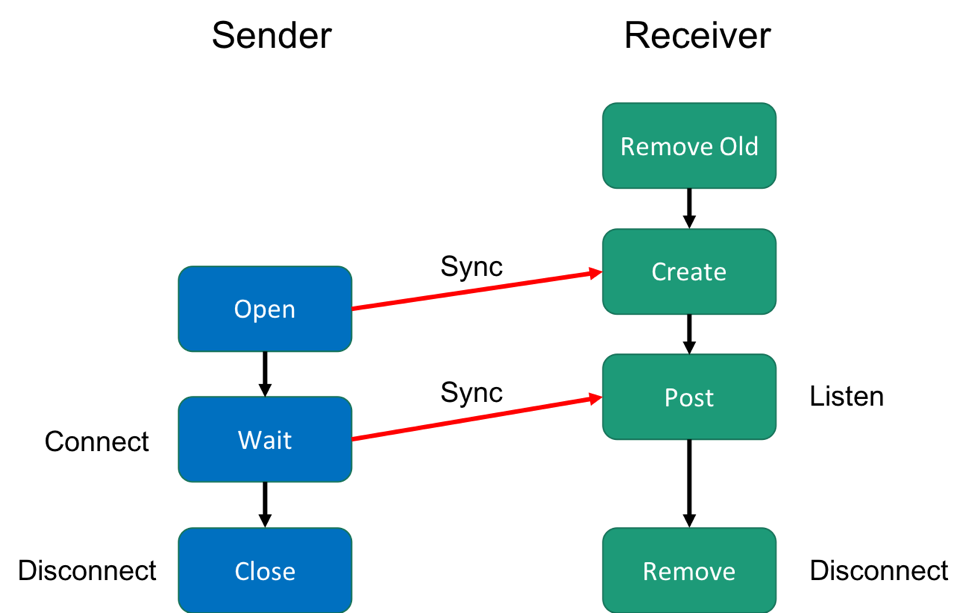
## Worker

Every worker is a running thread with a task queue, which stores the pointer of tasks assigned to it. The thread will keep polling put tasks from its own task queue and executes it, and generates *task done* event into the top-level event queue. If there is no task in the task queue, the worker will inform worker scheduler to move it into idle group and sleep until new task is added.

## Cuda Channel

The cuda channel uses semaphore to establish connections and use shared memory and build lock free FIFO on it.

### Connection



The sender and receiver need a unique ID to find each other. The ID should be unique at least on the host.

Sender and receiver use named semaphore to set up connection. The semaphore’s name is generated from the unique ID by adding prefix “SEMAPHORE\_”, for example “SEMAPHORE\_UniqueID”.

The receiver use sem\_unlink to clean up old semaphore with the same name (if exist). The semaphore with the same name is generated by crashed processes, and the named semaphore will not be cleared when process exit. Then the receiver creates a named semaphore with 0 value. The receiver posts the semaphore when listen called. The semaphore’s value going back to 0 means the sender is connect to the receiver.

The sender first tries to open the semaphore with the unique name. If the semaphore cannot be opened, it means the receiver is not prepared, the sender should wait until the receiver is ready. Then the sender waits the semaphore. If wait successful, the connection is established. If the wait failed, it means the sender is not listening. Or another sender is already connected to the receiver, which is not possible because of the unique name.

To close the connection, the sender and receiver can just close and unlink the semaphore. The same name can be reused after the receiver unlink the semaphore.

### Shared Memory

As all of the solutions use the shared memory, and the shared memory works the same, we discuss about shared memory first.

We use named shared memory, and the shared memory’s name should be generated from the unique ID the sender and receiver have, by adding prefix “SharedMem\_”, like “SharedMem\_UniqueID”.

The receiver should hold the shared memory. As the receiver also hold the receiver buffer and there will be multiple senders in solution D. When initialization, the receiver should first remove existing shared memory with the same name, which could be created by crashed receiver process. Then create the shared memory and initialize it. All these initialization job should be finished before connection established. When receiver destroy, receiver should destroy the shared memory.

**Caution:** There maybe resource leak. If one receiver crashes without destroying its shared memory and named semaphore. If the receiver restart using another ID, nobody will recycle these resources.

### Lock free FIFO

Cuda channel uses lock free FIFO to transport control data between different process. The FIFO only support single writer and single reader. Each side of channel owns a FIFO.

The FIFO is a ring buffer. The head and tail of ring buffer is shared between processes. The writer should write data into buffer first and update the tail then. The reader read data from buffer first and update the head then. This can make sure the FIFO works between 2 process without lock. The head and tail should be volatile to prevent out of order because of compiler optimization.

This lock free FIFO can only be used on X86 machines, because memory order is required and some other architectures like ARM do not guarantee the memory order.

### Data transfer

There are two design of data transfer, one is simple but slower, the other is more complicated but faster.

1. Synchronous transfer. The sender owns a cuda stream. The sender uses its stream transfer data and wait for it finish.
2. Asynchronous transfer. The sender owns a cuda stream. Call the asynchronous memory copy and insert a cuda event. The copy process won’t block the thread. And other thread can add memory copy task when other tasks unfinished. The caller can wait the cuda event synchronously to make sure the task is finished or not.

## RDMA Channel

### Connection

The RDMA Channel use RDMA’s send and receive to transfer the first metadata. The first metadata contains the information to get access to each other’s queue. After this, the connection is established.

If the RDMA API does not support this feature, or it will take too much time to write the part, we can use TCP to translate the first metadata. As the size of metadata will not be too large. And using TCP won’t affect the performance.

### RDMA FIFO

Like Cuda Channel, RDMA uses ring buffer to transport messages. But unlike cuda channel. The RDMA Channel cannot transport ring buffer’s head and tail efficiently.

We can use the method in FaRM[[1]](#endnote-1).

1. NARAYANAN D, HODSON O, CASTRO M, et al. FaRM: fast remote memory, The 11th USENIX Conference on Networked Systems Design and Implementation, April 2-4, 2014, Seattle, USA. Berkeley: USENIX Association, 2014: 401-414. [↑](#endnote-ref-1)