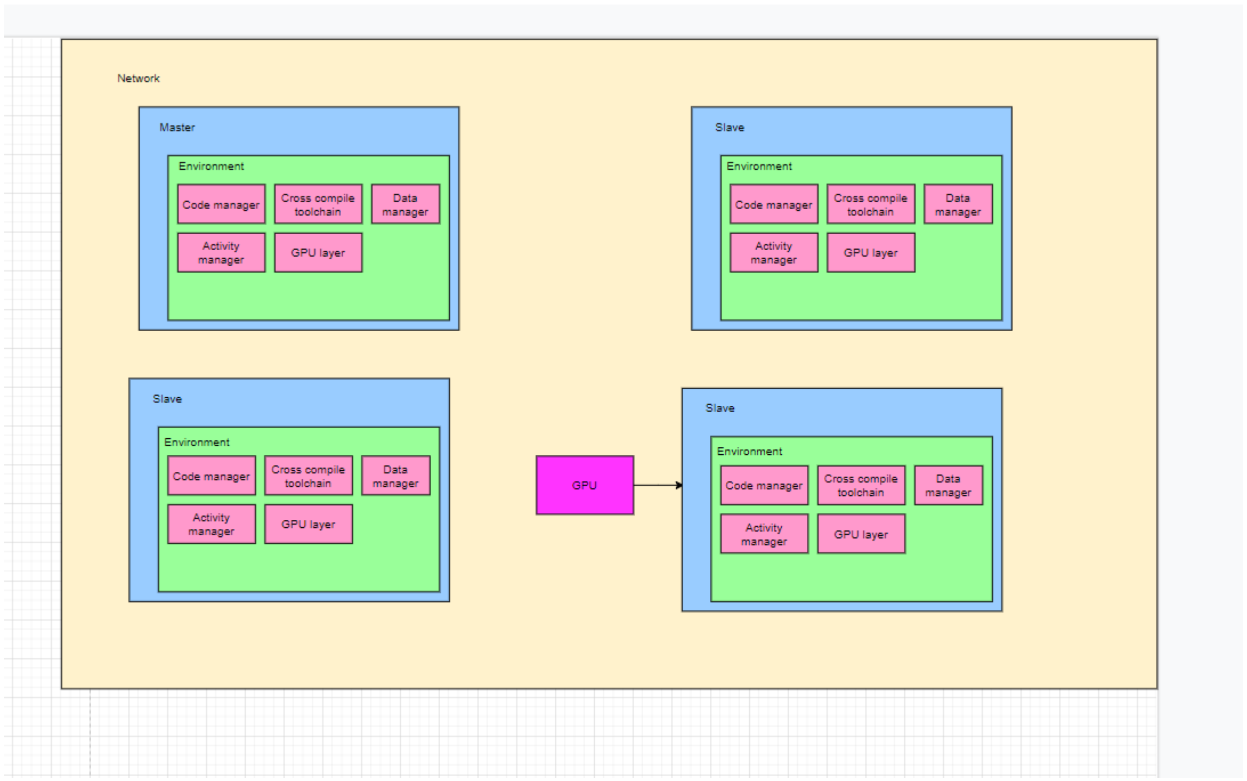
Introduction

Main concepts

Building blocks



Several components are listed in the following table

Name	**Description**	**Tasks**
Environment	Sets up environment variables and ensures that communication between components is possible as well as component startup	• Component startup • Platform capabilities checks • Environment variable setting like PATH • Component communication tools • Framework used by other components • Configuration management
Code manager	Responsible for transferring codebetween machines	• Invoking toolchain specific to machine

		• Sending compiled code to slaves • Loading received code from masters
Data manager	Ensured proper data exchange between nodes	• Sending data from master to slave • Sending data from slave to master • Ensuring proper encoding • Encryption
Activity manager	Ensures management of running tasks and machine discovery	• Slave discovery • Connecting to master • Running tasks • Keeping track of running tasks • Function call statistics • Smart task delegation based on profiling statistics
GPU layer		• Exchanging data CPU-GPU • Patching code for GPU • GPU discovery

Every component is started by environment startup as daemon. Daemons can be controlled through daemon control tool provided by environment. Other components are structured as control scripts which use framework provided by environment to communicate to respecting daemon for that component and also daemon services which are run on environment startup.

Toolchain

Environment also provides tooling for building code for various targets. On startup environment figures out host architecture while activity manager on master machine figures out target architectures of slaves.

Configuration

Environment reads configuration at its startup. Configuration is set of shell files which are sourced by environment. Depending on hostname, different configurations can be used or specific configuration file can be passed to environment.

Adding new components

New components can be added to environment which can enable more features. This includes:

- Adding new supported target architecture for CPU
- Adding new supported GPU
- Adding new component which will be started on environment startup
- Adding new language support to Code manager
    - Function discovery
    - Compilation
    - Loading
    - Unloading
- Adding new communication protocol to Activity manager

Logging

Framework provided by environment offers functions for logging. Logging can be controlled by configuration where certain levels of logging are logged. User can configure what levels are shown on console and what are written to log file for specific component. Environment also offers tool for aggregating and merging all logs and showing them to standard output.

Component implementation

All components are implemented in C++17 standard. Here are few things which are worth noting for component implementation:

- For thread management is used std::thread
- Code manager keeps track of compiled sources and for which architecture they are compiled, ensuring not doing two tasks twice. It also keeps track of hash of code such that it compiles only when code is changed.
- Code manager has inotify attached to filesystem watching for code changes.
- Activity manager notifies code manager of all needed architectures through named pipe
- When slave structure changes, code manager is notified and recompiles code for newly added architectures
- Data manager on master is responsible for data splitting and sending each chunk to slaves. When data is received from master, master then invokes activity manager to run Task on all slaves and master included. Code managers are notified by activity managers to load code and run task with arguments provided by data managers.

- Every component, when accessing data structure shared between threads, it has to synchronize this part to other threads.
- Every component has to initialize itself using framework library provided by environment
- Every component should use framework library provided by environment to communicate to other components
- Every component has to use framework for logging.
- Code manager, when loading code and running code, it should fork itself in case task fails. Code manager should never terminate unless stopped by environment
- Environment watches if all components are still running and if one daemon terminates, it should restart it and notify other components
- Other components should recognize failure of individual components and resend data again
- Environment, when component initialized, will create named pipe where other components can send data to this component
- Environment will send message to every component when all components are running and are ready to get messages.

Continious integration

All components are tesed on three levels:

- Unit tests
- Integration tests – for whole component
- System test – whole environment with communication with other machines

Profiling

Since only performance requirement is environment startup, measurement can be done from time environment is started until all component are ready to get messages.

Development environment

In the following list, tools are shown which will be used:

- Daemons: C++
- Framework: C++ and Boost
- Tooling from environment: C++ with developed framework

- Environment: Bash
- Controlling tools: C++ with framework
- Build system for tooling and daemons: CMake
- Container engine for development: Docker image will be created
- CI: Jenkins
- Unit tests: gtest and gmock
- Integration test: Bash
- System test: Multiple Docker containers in the same network and “empty” tool which executes and sends commands to environment
- Documentation handling: Microsoft Word

Code loading model

Code manager will be made of several classes:

- LoaderManager – class which searches loader path in CoNAL to discover all loader plugins available
- Loader interface which represents individual loader plugins:
    - Every plugin is shared object file with function createInstance which returns Loader instance
    - Loader interface has the following methods to implement:
        - verify(std::vector<std::string> params, std::map<std::string, std::string> machineParams) -> bool – returns if given loader can be used to load code given in parameters
        - load(std::vector<std::string> params, std::map<std::string, std::string> machineParams) -> Base64Array – loads code and return base64 representation suitable for sending, incase program needs to be compiled, this function searches for toolchain and compiles code.
        - start(Base64Array data) – extracts the code to the given space and runs it