# Ultra Low Latency File Synchronization Program Guide

Joshua Schaffer
Trevor Wright
Jayden Zebrowski
Jazmyn Zurita

## Client Setup & Configuration

### Dependencies

The client is implemented in Rust and relies on eBPF (Extended Berkeley Packet Filter) for system-level monitoring. The following dependencies are required for successful compilation and operation:

- **Rust (Nightly)** – Required for advanced features and compatibility with the Aya eBPF framework and cargo xtask.

- **BPF** – Enables loading and running of eBPF programs in the Linux kernel. Most modern distributions support this out of the box but must be enabled explicitly in some cases.

- **Clang** – Required for compiling the eBPF bytecode that runs in kernel space. Acts as the backend for bpf-linker.

- **Cargo bpf-linker** – A Cargo plugin necessary for linking user-space and kernel-space components of the eBPF programs.

- **Linux Kernel Headers** – Provide the necessary definitions for building eBPF programs that interface correctly with the kernel. These must match the currently running kernel version.

- **Build-Essential** – A meta-package (or equivalent) that ensures basic build tools such as make, gcc, and ld are available for compiling system-level components. Mandatory on Debian-based systems.

Note: All dependencies should be installed and configured prior to attempting a build. See platform-specific setup instructions for further detail.

---

### Setup (Arch)

- install rustup
- sudo pacman -S bpf
- sudo pacman -S linux-headers
- sudo pacman -S base-devel
- sudo pacman -S clang
- cargo install bpf-linker
- make install-aya-tool

- make bindings

---

**Setup (Debian)**

- Sudo apt install linux-headers-$(uname -r)
- sudo apt install build-essential
- sudo apt install clang
- cargo install bpf-linker
- make install-aya-tool
- make bindings
- Make run to run or cargo xtask run

---

**Additional Setup**

This section may not be required depending on your system configuration.

**Enable BPF LSM**

If BPF LSM is not enabled, follow these steps to enable it. Reference: [AYA Book – LSM]

---

**Requirements**

- **Kernel Version:** Must be at least 5.7
- **LSM Check:** BPF must be listed as an enabled LSM

---

**Verify BPF LSM**

---

**Run the following command:**

cat /sys/kernel/security/lsm

**Expected output should contain:**

capability,lockdown,landlock,yama,apparmor,bpf

**If bpf is missing from the list, BPF LSM must be enabled manually**.

---

## Enable BPF LSM via GRUB

---

1. **Edit the GRUB config:**

   sudo nano /etc/default/grub

2. **Locate the line beginning with GRUB_CMDLINE_LINUX and append ,bpf to the existing list of LSMs:**

   GRUB_CMDLINE_LINUX="lsm=[existing LSMs],bpf"

3. **Rebuild GRUB using one of the following commands (depending on your distro):**

   sudo update-grub2

   or

   sudo grub2-mkconfig -o /boot/grub2/grub.cfg

   or

   sudo grub-mkconfig -o /boot/grub/grub.cfg

4. **Reboot your system to apply changes:**

   sudo reboot

---

## Configs

Client configuration is controlled via the config.json file located in the project's root directory. Key configuration fields include:

- **Dns_web_addresses:** A list of server domain names or IP addresses to connect to. This list is dynamically re-read while the client is running, allowing updates without restart.

- **Watch_dir:** The local directory to monitor for file changes. **Important:** The full path to this directory should be **shorter** than the server's path for consistency in file indexing and transmission.

- **Server_port:** The port number used to connect to remote servers. By default, all connections use the same port, but this can be adjusted in code if multi-port support is desired.

- **Ignore_ext: (Deprecated):** Previously used to ignore files based on extension, now no longer used.

- **Ignore:** A list of .gitignore-style rules. Files or directories matching any rule will be excluded from synchronization. Each rule should be a separate string in the configuration list.

- **File_store_time_minutes:** Defines how long (in minutes) a file's full contents are kept in RAM before being purged. This controls RAM persistence for cached files.

- **Max_total_size_mb:** Maximum total size (in MB) of all files held in RAM. If this limit is reached, new files will be skipped for caching and transmitted immediately without local storage.

---

**Code info**

The client implementation is organized into multiple components. Below is a breakdown of the structure and functionality:

- Directory Structure
  - All user-facing code resides in the ullfs directory.
  - The ullfs-ebpf directory contains BPF-specific logic and may require modification if additional detections are needed.
  - All other code outside these directories should generally remain unchanged.

- Client_tcp
  - This module is responsible for handling all data transmissions.
  - If enhancements such as compression or encryption are desired, the packet format should be adjusted within this module.

---

**ULLFS Protocol – Packet Format**

The client sends data packets to the server following a custom protocol designed for performance and efficiency:

1. Filepath
   a. Each character in the local file path is sent as a u8, relative to the watch_dir.
2. Null Terminator
   a. A single 0u8 byte is used to terminate the filepath string.
   b. (File paths cannot contain a null byte, ensuring unambiguous termination.)
3. Flag (u8)
   a. One-byte flag indicating the type of operation:
      i. 1: Full file send

    ii. 2: Delta send
    iii. 3: Delete file
    iv. 4: Move/Rename file
    v. 5: Create empty file
    vi. 6: Create directory

Additional Data by Flag

- Flag 1 – Full File Send
    - u64: File size in bytes
    - Raw file data
- Flag 2 – Delta Send
    - u64: Start byte (offset from beginning of file)
    - u64: End byte (offset from beginning of file)
    - u64: Data length
    - u64: Hash of the original file
    - Raw delta data
- Flags 3, 5, 6
    - No additional data beyond the flag.
- Flag 4 – Move/Rename File
    - New path as a string
    - Terminated with 0u8

---

**Packet Reception (Retransmission)**

The client does not yet fully handle retransmission. If a retransmission packet is received from the server, it contains only a 0u8-terminated string (the file path). The client is expected to send back the full file, but this case is not yet implemented. The structure is in place for future support.

---

## Server Setup & Configuration

---

**Dependencies**

The server is implemented in **Rust** and built on top of the **Steady-State Framework**. The following dependencies are required for successful compilation and execution:

- **Rust (Nightly)** – Required for compatibility with the Steady-State Framework macros and features.

- **Clang** *(optional)* – May be needed for certain underlying system dependencies.

- **Build-Essential** *(optional)* – Recommended for compiling native dependencies on Debian-based systems.

  **Note:** In rare cases, the server may require additional dependencies that are typically needed by the client. However, these are not generally expected for standard operation.

---

### Configuration

Server configuration is controlled via the config.json file located in the project's root directory. Key configuration fields include:

- **Watch_dir**: The directory to monitor for changes. Files within this directory are watched and changes are pushed to clients.

- **Server_ip**: The server should always be bound to 0.0.0.0 to accept connections on all network interfaces.

- **Server_port**: Set this to match the port expected by the client. Avoid using port 9100, which is reserved by the Steady-State Framework for internal telemetry and visualization.

  **Ignore**: This field is inherited from the client configuration and is currently unused in the server.

---

### Running the Server

To start the server:

cargo run

The application will compile and launch using the parameters defined in the config.json.

---

### Monitoring Telemetry

The Steady-State Framework provides real-time telemetry through a web interface. Once the server is running:

1. Open a web browser.

2. Navigate to: [http://127.0.0.1:9100](http://127.0.0.1:9100)

This dashboard allows you to observe actor activity, channel throughput, and system utilization in real time.

---

**Codebase Overview**

This server operates within the **Steady-State Framework**, which enforces a strict structural organization across the codebase. The primary logic for actor and channel creation resides in /src/main.rs. Developers should begin here when working on the core system configuration.

**Actor and Channel Configuration**

Channel and actor creation follow global rules defined in graph.channel_builder() and graph.actor_builder(). These configurations apply uniformly unless explicitly overridden. The foundational method base_channel_builder() is used to instantiate a channel, taking two parameters: the transmitter and the receiver.

- To configure a channel's capacity, use the .with_capacity() method, which sets the maximum number of messages that can be in-flight at any given time.

- Finalize and instantiate the channel using the .build() method.

Actor instantiation is handled via base_actor_builder.with_name(), where you can define the actor's name, specify the logical core it runs on, and attach its communication channels.

For channel-specific or actor-specific behavior, refer to the [Steady-State crate documentation](#) for implementation guidelines and customization options.

**Actor Implementation**

The behavior of each actor is defined in its corresponding source file located under /src/actor/{actor_name}.rs. These implementations follow a consistent structure based on the **FizzBuzz Steady-State** template.

The entry point for each actor is the run() function. Its parameters must align with those declared in the main file; as additional channels are connected, these parameters must be updated accordingly.

Telemetry is integrated via the into_monitor!() macro, which relays runtime statistics to the graph. The macro takes three arguments:

into_monitor!(context, [receiving_channels], [transmission_channels]);

**Important:** Rust's compiler and Rust Analyzer will not catch ordering mismatches in these macro arguments. If telemetry fails to output, this is a key area to inspect for potential issues.

The core logic of each actor is implemented in the internal_behavior() function. This function has the same parameters as run(), with an additional one for telemetry monitoring.

- Begin by locking all connected channels.
- Enter a while loop that processes until all transmission channels are **marked closed** and all receiving channels are **closed and empty**.
- Use try_take(channel_name) to consume messages from a receiving channel.
- Use send_async(channel_name, message, SendSaturation) to send messages through transmission channels.
- Call relay_stats() to update telemetry metrics, provided the corresponding attribute is defined in graph.actor_builder().

**Modular Design**

To simplify the codebase and improve maintainability, the server's processing logic has been modularized:

- All logic related to the **ULLFS protocol** is implemented in /src/actor/handle_client.rs.
- Configuration values, such as watch_dir, are centralized in the config.json file located in the root directory. This file acts as a single source of truth for critical runtime settings.
- Configuration parsing and accessors are implemented in /src/actor/file_filter.rs, allowing for easy instantiation and retrieval of configuration values using getter methods.

This modular and clearly structured approach ensures scalability and maintainability as the project grows.

---

---

**eBPF Components**

---

1. Kernel-space eBPF program (ullfs-ebpf/src/main.rs): Implements LSM and FExit hooks
2. User-space application (ullfs/src/actor/ebpf_listener.rs): Loads eBPF, processes events

---

## Kernel-Space Implementation

---

### eBPF Constraints

- No standard library (#![no_std])
- Limited stack space (512 bytes)
- Limited instruction count (1000000 instructions)
- No access to arbitrary memory
- No dynamic memory allocation
- Limited error handling

### Hook Types

1. LSM Hooks:
    - Primary source of filesystem events
    - Hooks: path_unlink, inode_create, path_rename, etc.
    - Used for capturing operations at security checkpoints
2. FExit Hook:
    - Currently only vfs_write is used
    - Captures write operations not visible through LSM hooks

### Kernel Structure Navigation

1. Dentry-centric design:
    - Path is only used to get to dentry: path → dentry
    - File is only used to get to path to get to dentry: file → f_path → dentry
    - All operations work with dentries as the primary data structure
2. Parameter extraction functions:
    - try_arg_path: Extracts dentry from path parameter
    - try_arg_file: Extracts dentry from file parameter
    - try_arg_dentry: Uses dentry parameter directly
    - try_arg_dentry_parent: Gets parent dentry when target is empty
3. Empty dentry issue:
    - When dentries are empty/incomplete at interception time
    - Only parent dentries contain valid info
    - Requires delayed inode verification in user space
4. Key kernel structures:
    - dentry: Directory entries forming filesystem hierarchy
    - inode: Unique file identifiers with metadata
    - path: File path representations (only used to access dentries)
    - qstr: Kernel string implementation for names

    - All accessed safely via bpf_probe_read_kernel

**Data Structures**

EventData:

```
struct EventData {
   inod: u64,        // Primary inode number
   inod2: u64,       // Secondary inode (for rename operations)
   len: u16,         // Length of first path string
   len2: u16,        // Length of second path string (for rename)
   event_type: u8,   // Type code for the event
   rename_state: u8, // Rename state (different directory scenarios)
}
```

BPF Maps:

- INODEDATA: Stores watched directory inode
- PROGDATA: Contains process information
- BUF and BUFTWO: Per-CPU arrays for path storage
    - Per-CPU design prevents synchronization issues
    - Each CPU gets dedicated buffer space
- EVENTS: Perf event array for user space communication

**Path Extraction**

1. Path building process:
    - pathToMap: Traverses directory entries to build path
    - dnameToMap: Extracts name components from dentries
    - Path components stored in per-CPU arrays
    - Special character (¦) used for empty components
2. Directory context:
    - in_dir: Traverses up directory tree
    - Checks if operations occur in watched directories
    - Compares against watched inode from INODEDATA
    - Filters events to watched directory only
3. Limitations:
    - Kernel paths stored in reverse order (leaf to root)
    - Limited buffer sizes require chunking
    - Special handling for rename/move operations

**Event Communication**

1. Event details packaged in EventData structure
2. Path information stored in per-CPU arrays
3. Event notification sent to user space via perf event

---

## User-Space Implementation

---

**eBPF Program Management**

1. Loads compiled bytecode using Aya framework
2. Attaches 20+ LSM hooks and the vfs_write FExit hook
3. Initializes maps:
    - Watched directory inode in INODEDATA
    - Process ID in PROGDATA
4. Sets up per-CPU event processing

**Two-Phase Event Processing**

1. Initial phase (in eBPF):
    - Capture event type and available info
    - Extract inode numbers (often from parent dentry)
    - Store partial/full path information
2. Verification phase (in user space):
    - check_inode: Set when additional verification needed
    - Uses inode to find actual file/directory
    - Scans directory for matching inode
    - Reconstructs complete path from found file
    - Necessary because target dentries are often empty during syscall interception

**Path Reconstruction**

1. extract_filename function:
    - Reads components from per-CPU arrays
    - Handles placeholder characters
    - Reverses path (kernel stores leaf to root)
2. Path components joined with proper separators
3. Relative paths calculated from watched directory

**Operation Type Resolution**

1. Event type from kernel (0-31)
2. Context from inode verification
3. States (mode variable):
    - Mode 1: Rename operation
    - Mode 2: File creation
    - Mode 3: Directory creation
    - Mode 4: Deletion
    - Mode 5: Move operation
    - Mode 6: Move into watched directory
    - Mode 7: Creation of unknown type

**Special Cases**

1. Rename operations:
    - rename_state field tracks directory context:
        - State 1: Both in watched dir

        - State 2: Destination in watched dir (move to)
        - State 3: Source in watched dir (move from)
    - Requires tracking two paths simultaneously
2. Cross-boundary moves:
    - Files moved into watched area trigger full content send
    - Files moved out treated as deletes

**Final Processing**

1. Event categorized by type
2. TcpData structure created with:
    - Operation type
    - File path(s)
3. Data sent through channel to other components

---

## Troubleshooting

---

### Client

---

**Issue:** `cargo xtask run` does not work

**Solution:** Try running the command using explicit binary targeting:

cargo run --bin xtask run

This occasionally happens after toolchain or crate updates.

---

### Server

---

**Issue:** Client fails to connect to the server

**Possible Causes & Solutions:**

- **Mismatched Ports:** Ensure that the client and server are configured to use the same port.
- **Network Restrictions:** If running across different networks, make sure the target port is open and properly forwarded on the server's router.
- **Debugging by Reversion:** If unsure what broke the connection, revert recent changes and gradually reapply them. Even minor changes can cause communication issues.

---