fRender: Distributed Rendering for Blender

Neena Dugar, Jessy Lin, Stef Ren

1 The Problem

The current suggested method for distributed computation, from the Blender docs reads as follows:

> ...you can do WAN rendering, which is where you **email or fileshare the .blend file** (with packed data!) across the Internet, and use anyone's PC to perform some of the rendering. They would in turn **email you the finished frames as they are done**. If you have reliable friends, this is a way for you to work together.

We believe it is possible to build a better solution. For our final project, we built fRender, a distributed computing system that allows a group of users to collaboratively render a model on their personal laptops, across the internet. While there are some systems that already allow distribution of work (CrowdRender, BitWrk), we were interested in creating a system that is fault-tolerant and inherently robust against malicious peers in the system.

2 System Design

We use a Golang backend and interface with Blender through the command line. Our system consists of three main components:

- The **master**: a main registry that keeps track of active friends and distributes rendering jobs across users in the system.
- The **clients:** users in the fRender system. Each user registers itself separately as two entities:
    - **requester**: submit Blender jobs to be rendered by other people. This component runs virtually on a managed fRender server (see *Figure 1*), to ensure that there are no bad requesters. The client is able to upload files to the requester and state the number of friends they want; they will receive rendered frames in return.
    - **friend**: render files on their personal computers.

  The master keeps the requesters and friends as two separate lists internally, but they can be linked to each other through the unique username that the client registers both the friend and requester with.

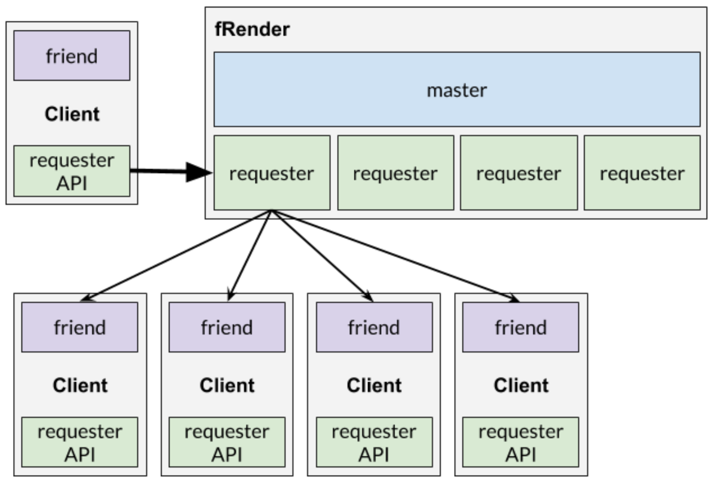


Figure 1. System diagram. A client submits jobs to virtual requesters located on fRender servers, which then distribute jobs to friends.

Communication in the system is done with RPCs, except when friends and requesters exchange files over a raw TCP socket. The friends heartbeat the master periodically to communicate that they are alive, whether they are busy rendering a job or available to be assigned more work, and the last job number (global across all requesters) they completed. The master keeps track of the last time that it received a heartbeat from each friend, whether each friend is available (as of the last heartbeat), and the last job that a friend completed.

When requester submits a job of *M* frames requesting *N* friends (the requester can pay more points to render with more friends in parallel) to the master, the master returns a list of addresses the first *N* friends who have sent a heartbeat in the last *friendTimeout* milliseconds, where we have set *friendTimeout* = 200. The master marks these friends as busy and records the (global) job number *J* that this friend has been assigned to. These friends will not be marked available until they each report they have finished job *J* on the next heartbeat, so that future jobs will not be assigned to them.

When it receives the list of *N* addresses back from the master, the requester communicates directly with the friends. The requester sends .blend files to be rendered and receives rendered frames back from the friends. When all the frames are eventually rendered (with retries as necessary, described in **2.1** and **2.2**), the requester reports how many frames each friend rendered.

2.1 Fault-tolerance

We use a MapReduce style scheduler to account for friends who do not respond. If a friend does not respond to the requester within a given timeout, the requester retries the same task on a different friend. We do not account for the case where all friends go down, but this would be a simple fix – the requester could request new friends from the master. We do not address the case of master failure.

2.2 Verification Scheme

We designed a verification scheme to account for malicious friends in the system who intentionally return blank or mis-rendered frames automatically. In current distributed rendering systems like BitWrk, the rendered frames would have to be checked and restarted manually by the requester.

To illustrate the challenges of the verification problem, we describe a few intuitive first passes at a scheme and characteristics of our design that strive to address their shortcomings:

Potential Solution 1: Requester as the verifier
The requester checks n frames from every friend. If the requester finds that { at least one , the majority } of frames are mis-rendered, then the friend's frames are discarded and the friends are punished with the incentive scheme (e.g. the one we implement, as described in **2.2.3**).
***Analysis**:* This scheme may be off-loading too much work to the requester. Additionally, one bad requester who arbitrarily reports that all its friends have rendered its frames wrong could punish a lot of good friends.
Our scheme should distribute the verification work among many users, both to minimize the work that requesters need to do and minimize the damage done by a malicious requester who denies that its friends did any work.

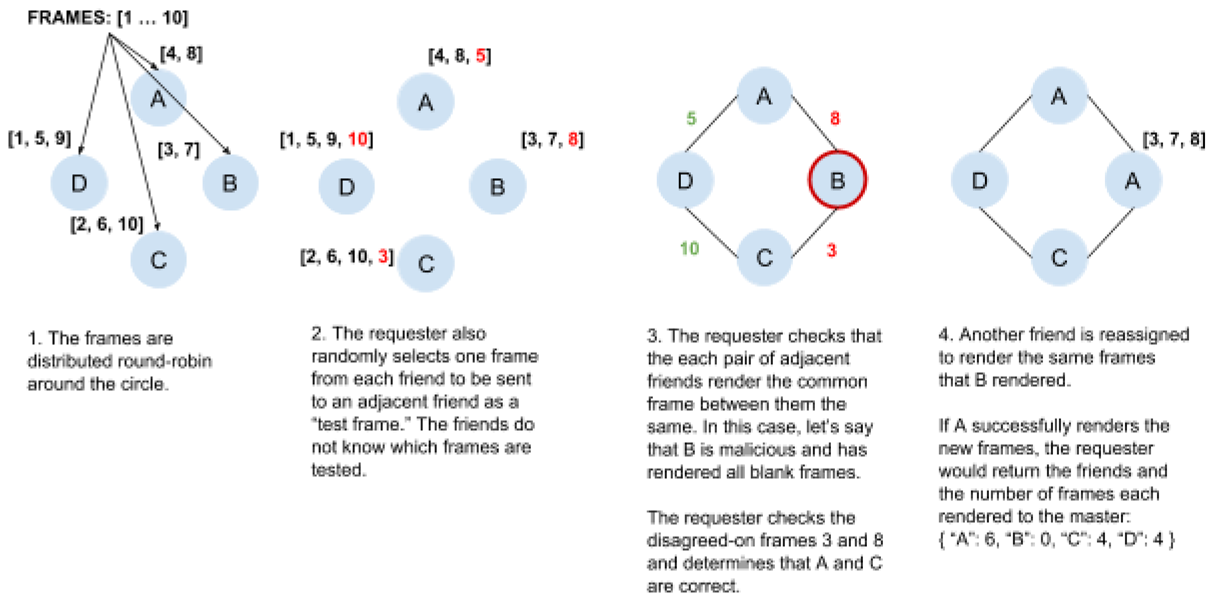
Potential Solution 2: Friends verify each other
Every frame is rendered by *n* (e.g. 3) distinct friends, and we only accept frames where there is majority agreement; other frames are retried on other friends. Friends who are in the minority are deemed malicious and punished.
***Analysis**:* This scheme works well at scale, when there are many friends assigned to each job (so that the probability of two friends owned by the same malicious user being assigned to the same subset of frames and outvoting the honest users is small). However, a lot of work is repeated since many frames is duplicated.
Our scheme should aim to minimize duplicated work, while maintaining that cheating is unreasonably difficult. It should also work reasonably well even when there are only a few friends assigned to a job.

We implement a hybrid scheme that combines ideas from these two simple schemes. In the normal use case (where most friends are honest), the verification work is done by the friends, and the requester only steps in to arbitrate / re-render frames when two friends render the same frame differently.

Friends form a "verification circle" where each friend renders a single "test" frame from the friend on its right. This ensures that the level of duplication is very low – the number of duplicate frames is equal to the number of friends. However, even with this low level of duplication, it is hard to cheat because a malicious user does not know which of its frames are being verified by other users that he does not control.

A malicious user is detected if they disagree with their adjacent friends; the requester serves as arbitrator on disagreements by re-rendering the contested frame. The malicious user will be removed entirely from the circle and will not be rewarded for rendering any frames, and the frames that it was assigned to render will be re-assigned to another friend.

An example of the verification process working in action is shown in the figure below.



1. The frames are distributed round-robin around the circle.

2. The requester also randomly selects one frame from each friend to be sent to an adjacent friend as a "test frame." The friends do not know which frames are tested.

3. The requester checks that the each pair of adjacent friends render the common frame between them the same. In this case, let's say that B is malicious and has rendered all blank frames.

The requester checks the disagreed-on frames 3 and 8 and determines that A and C are correct.

4. Another friend is reassigned to render the same frames that B rendered.

If A successfully renders the new frames, the requester would return the friends and the number of frames each rendered to the master: { "A": 6, "B": 0, "C": 4, "D": 4 }

Figure 2. Verification scheme.

2.2.3 Incentive Scheme

This verification scheme works in tandem with a point-based incentive system that specifies the punishments for misbehavior and reward for rendering properly. Submitting a job with *M* frames and *N* friends costs a client *N*+*M* points, and each frame that is rendered correctly (as reported by the requester) earns a friend one point. There is thus no incentive to requesters to submit dummy jobs to reward other machines that they own, since it costs the same number of points to submit a job as the maximum reward you could earn.
It would be easy to punish friends for rendering frames incorrectly, but we do not implement this for simplicity.