The Octopus `copy` device and API

The copy device is introduced to allow an application running on the PC to copy data from a (remote terminal) source to a (remote terminal) destination without itself (and the PC) being an intermediate. The motivation is to reduce the “distance” traveled by the data (thus also have less latency) as well as to reduce the processing / communication overhead at the PC.

The copy device is implemented as a Styx server program, using a suitable file-based interface. Applications may either access the device directly (to do this, the programmer must be aware of the internal access protocol details) or via a library offering a suitable high-level API.

To achieve more flexibility, the copy device is implemented to be as **independent** as possible of any assumptions and conventions that are specific to the octopus system. It can be invoked from any program, also from the command line. On the contrary, the copy library **relies** on several octopus-specific assumptions and conventions, and is intended to be invoked from within code that runs on the PC.

The copy device and library are described in more detail in the following.

Copy device: operation, protocol, semantics

A copy operation is performed by opening a special control file, named “`new`”, and writing a request into it. The write blocks until copying finishes or an error occurs. Copying is aborted if the write operation is flushed. The number of bytes copied can be retrieved by reading the control file (for repeatedly reading the file, the read position must be reset to 0).

Opening the control file returns a file descriptor to a “new” virtual control file. Thus, different application programs and threads may concurrently perform copy operations through the same copy device, without interfering with each other. A virtual control file is “removed” when the corresponding file descriptor is garbage collected (and the copy device server receives a Clunk for the respective fid).

A copy request is submitted using the format below:

```
<srcaddr> <srcpath> <dstaddr> <dstpath> <srcoff> <dstoff> <bytes>
```

where

`srcaddr,dstaddr`	the dial address for the src/dst terminal (“localhost” for local)
`srcpath,dstpath`	the filename to read/write
`srcoff,dstoff`	the offset from where to start reading/writing
`bytes`	the number of bytes to read/write (0 for until EOF)

The copy device server does not make any “smart guess” about paths and terminal addresses. This information must be provided to it explicitly. Filenames are given relative to the root of the file system being addressed. In case of a remote file system, filenames are interpreted relative to the root of the file system exported at the address given. In case of the local file system (on the same host as the copy device), the address must be “localhost” and filenames are interpreted relative to the root of the copy device (which can be specified when starting it).

If all the initialization steps succeed, the copy operation commences. No reply is sent to the client that invoked the operation until the copy completes (or aborts). The client remains blocked during the copy. In case of an error, a respective Styx error message is sent back as a reply.

Copy device: installation

The copy device can be installed at a given location using a mount command:

```
mount {copydevice [-d] <rootdir>} <mountdir>
```

The `-d` option enables the printing of debugging information on stderr. The root directory must be given in an absolute way, i.e., start with '`/`'. The mount point must be an existing directory.

If the copy device is installed successfully, a virtual file named "new" will appear in the directory.

Copy device: usage example

Assume that two remote file systems "fsA" and "fsB" can be mounted by dialling `addrA!portA` and `addrB!portB`, respectively. Also, assume that a copy device is installed at `/foo/cpd`.

To copy file `pathA/fA` from the fsA to file `pathB/fB` on fsB

```
      echo addrA!portA pathA/fA addrB!portB pathB/fileB 0 0 0
           > /foo/cpd/new
```

To copy file `pathA/fA` from fsA to file `pathB/fB` on the (local) copy device file system

```
      echo addrA!portA pathA/fA localhost pathB/fB 0 0 0
           > /foo/cpd/new
```

To copy file `pathA/fA` from the (local) copy device file system to file `pathB/fB` on fsB

```
      echo localhost pathA/fA addrB!portB pathB/fB 0 0 0
           > /foo/cpd/new
```

The copy device may be installed on any device, independently of the location of fsA and fsB.

Copy device: implementation details, limitations

It is not possible to read the number of bytes copied from the command line. This is because the virtual control file "created" when opening the "new" control file is anonymous; it can be accessed only via the file descriptor returned to the program that invoked the sys->open call.

To amortize the overhead of mounting a remote file system, the copy device keeps mounted file systems after a copy completes. They are unmounted when they remain unused for some time (currently, 60 seconds). To avoid using a separate thread, mounts are garbage collected in a lazy fashion, each time a new request arrives. They are also collected when the copy device is shut down (unmounted).

The copy device currently mounts remote file systems using Styx. Of course, it can be easily changed to use ofs instead.

Copy library: conventions and assumptions

Terminal file systems are mounted at the PC at `/terms/<termname>`. In turn, each terminal mounts `/pc` and `/terms` from the PC on its local name space at the same locations (it also overloads `/terms/<termname>` with the root of its own local name space exported to the PC in order to avoid loops).

To support p2p copying, a terminal installs a copy device locally at `/cpd` (at the root of the file system exported to the PC). Thus, the copy device of a terminal is accessible from any program running at the PC at `/terms/<termname>/cpd`. The root of the copy device (see previous section on installation) must coincide with the root of the file system exported by the terminal to the PC at `/terms/<termname>`.

A terminal must be published with the registry with the following attribute-value pairs (in addition to any others): `oterm=<termname>` to denote the name that is used to mount the terminal under `/terms` on the PC; `p2paddr=<ipaddr!port>` to denote the address to dial for mounting the terminal file system that is exported to the PC from another terminal.

Breaking any of these assumptions will hinder p2p copying via the copy device, or result in unspecified behavior.

Copy library: API and semantics

The copy library relieves the programmer who wishes to perform a file copy from (a) parsing the source and destination filenames and searching the ns file to determine whether these include a mount/bind to a terminal file system, (b) extracting the respective terminal names and file paths relative to the terminal file system exported to the PC, (c) retrieving the source terminal p2p address, and (d) conducting the interaction with the copy device of the destination terminal. A conventional copy operation is also given, to be used as a fall-back solution, in case p2p copying fails.

```
CopyLib: module
{
  copy_local: fn(src, dst: string,
                 srcoff, dstoff, n: big): (string, big);
  copy_p2p: fn(src, dst: string,
               srcoff, dstoff, n: big): (string, big);
  trace_p2p: fn(on: int);
};
```

The `copy_local` operation performs a normal copy, via the PC, by directly reading the source and writing into the destination file, irrespectively of whether the corresponding file system are local or remote (imported from a terminal). It returns an error string (nil if the copy was successful) and the number of bytes copied (this may be non-zero also in case of an error).

The `copy_p2p` operation performs a p2p copy, via the copy device of the destination terminal. It returns an error string (nil if the copy was successful) and the number of bytes copied (this may be non-zero also in case of an error).The operation fails by default if the source or destination file systems are local or it is not possible to retrieve the dial address of the source file system or it is not possible to properly communicate with the copy device of the destination file system. The `trace_p2p` operation can be used to enable the printing of verbose debug information for the p2p operation.