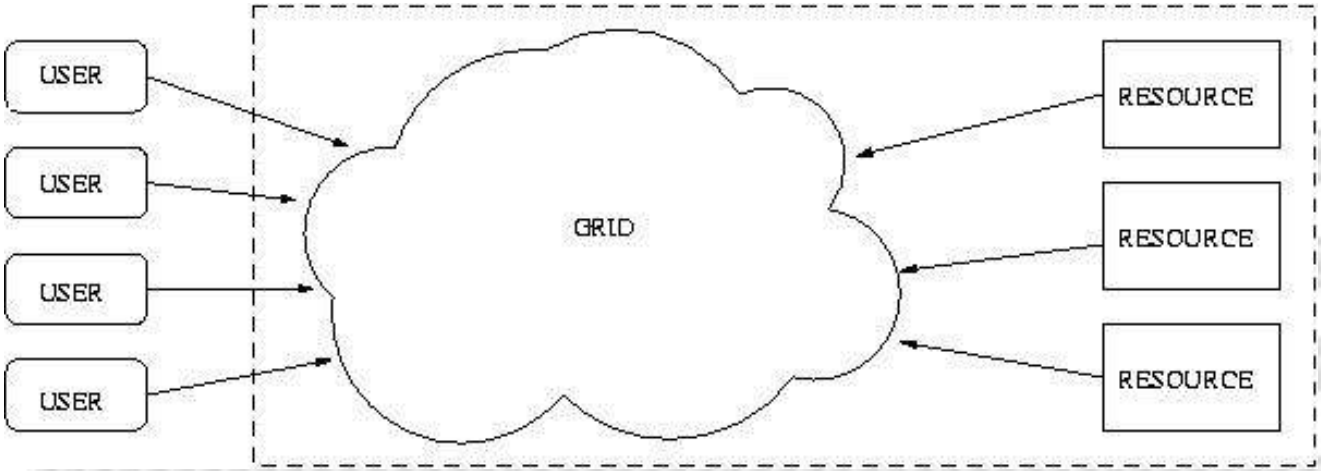
Minimum intrusion Grid

Transparent Remote On-demand File Access

Design and implementation

Introduction

MiG Model:



- This project deals with file access from the resource
- All users own a home catalog residing on MiG servers

Motivation

- Huge input files incur a number of problems:
  - Download time vs. total execution time
  - Job execution on the resource is delayed
  - Storage requirements on resources
- Often only small scattered fragments of input files are needed
- How about automatic on-demand download of needed data?

Example

```
int fd = open(“inputfile”, O_RDONLY);
while ((i=read(fd, &buffer, 2000)) >0){
    /* process buffer */
}
```

- User applications need not be recompiled or rewritten using a custom MiG API

Objectives

- A file transfer protocol supporting range requests
- Catch file access routines
  - Direct them to the server holding the inputfile
  - Direct local file access to the native file system
- Ensure minimum intrusion on the resource

File Transfer Protocol

- HTTP supports a “range” parameter in get request:

```
GET /inputfile HTTP/1.1
HOST: MiG_server.imada.sdu.dk
Range: bytes=2000-3000
```

- No range support in put requests
- In order to support writing to remote files, a custom web server is developed

File Access

- Override a subset of file manipulating routines:
  - open, close, read, write, seek, dup, sync, etc.
- Preload this library using the `LD_PRELOAD` environment variable (requires user applications to be dynamically linked)
- Forward local file access to the native file system using the `dlfcn` library with the `RTLD_NEXT` handle

Minimum Intrusion

- “Minimum intrusion” implies:
  - No root install of software on the resource
  - No requirements on the firewall configuration
- All we need is a local grid user and an ssh-connection
- Thus everything must run in user space

Block Size

- Simple solution:
  - general purpose block size based on n/2-analysis
- Advanced solution:
  - depends on the user application:
    - Nature: sequential vs non-sequential file access
    - The block size used in the application
- Introduce prefetching (1 block read-ahead)
- Adjust the block size dynamically
  - based on the prefetching and the time taken to transfer a block

Experiments

- 4 experiments:
  - Overhead: read a one byte file
  - I/O intensive application: Checksum a 1 GB file
  - I/O balanced application: Process a 1 GB file
  - Partial file traversal: Search a 360 MB B+ tree for a random key
- 3 test setups:
  - Local execution
  - Copy model
  - Remote access model

Results

Experiment	Local execution	Copy model	Remote access model
1 byte file	0.0002	0.1520	0.0080
Checksum	50.1100	130.1000	108.5800
Fibonacci	638.8300	721.2200	708.7200
B+ tree	0.0002	30.6920	0.0186

Conclusion

- Download time of input files is eliminated; the job is started immediately
- Only needed data is transferred
- Jobs previously impeded from grid submission due to huge input files are now ready for the grid
- The model outperforms a standard copy-model
- The library is entirely user-level