Comunicação entre Cliente e Servidor

Sistemas Operativos

Assume-se que o cliente comunica com o servidor por envio de códigos definidos no ficheiro Common.h. O cliente comunica com o servidor por um dos pipes com nome, enquanto que o servidor comuncia os resultados por um outro pipe com nome. Para o cliente poder comunicar com o servidor é que o pacote enviado contenha nos primeiro 4 bytes a operação que se pretende realizar, seguido pelos respectivos argumentos da função, quando necessários. Assume-se que os dados recebidos pelo servidor são válidos em termos sintáticos.

Porém, continua ainda a ser responsabilidade do servidor materializar este pacote e construir de forma organizada os argumentos passados. Isto pode ser feito de forma inteligente, utilizando alguns bytes de overhead, com a seguinte nomenclatura.

`[ID COMANDO] [Nº Argumentos] [OFFSET arg0] [OFFSET arg1] ... [DADOS]`

O overhead corresponde a tudo que não seja dados, desta forma explicitada, o servidor vai poder automaticamente materializar este pedido, podemos então definir isto da seguinte forma:

```
typedef struct request {

  int ID;             /* Primeiros 4 bytes */
  char **argv;        /* Argumentos utilizados */

} *Request;

Request materialize_request(char* buffer, ssize_t length) {
  /* Deve haver pelo menos 2 * sizeof(int) bytes */
  int i, nArgs;
  Request request = NULL;
  char *tmp, *offsetdata = NULL;
  ssize_t arglen, *offsets, base;

  if(length >= 2 * sizeof(int)) {
    request = malloc(sizeof(struct request));

    // Estabelece o numero de pedido.
    request->ID = *(int*)buffer;
```

```
    // Estabelece o numero de argumentos.
    nArgs = *(int*)(buffer + sizeof(int));

    // Partindo disto, podemos construir os offsets;
    offsetdata = buffer + 2 * sizeof(int);
    offsets = (ssize_t*)malloc(sizeof(ssize_t) * request->nArgs);
    for(i = 0; i < nArgs; i++) {
      offsets[i] = *(ssize_t*)(offsetdata + sizeof(ssize_t) * i);
    }

    // Assim sendo, passa a ser possivel povoar os argumentos em si.
    argv = malloc(sizeof(char*) * (nArgs + 1));
    argv[nArgs] = NULL;

    i = 0;
    base = 2 * sizeof(int) + nArgs * sizeof(ssize_t);
    // Define os primeiros N - 1 argumentos;
    for( ; i < nArgs - 1; i++) {
      arglen = offsets[i + 1] - offsets[i];
      tmp = malloc(sizeof(char) * (arglen + 1));
      memcpy(tmp, buffer + base + offsets[i], arglen);
      tmp[arglen] = '\0';
      request->argv[i] = tmp;
    }

    // Define o ultimo argumento.
    if(nArgs > 0) {
      arglen = length - offsets[i];
      tmp = malloc(sizeof(char) * (arglen + 1));
      memcpy(tmp, buffer + base + offsets[i], arglen);
      tmp[arglen] = '\0';
      request->argv[i] = tmp;
    }
  }

  return request;
}
```

Assim, já temos uma forma de comunicar informação do cliente para o servidor, e potencialmente materializar esta informação. Falta agora saber de que forma processar esta informação. O ideal será sermos capazes de fazer este processamento dinamicamente, com um array em que cada posição fica associada a uma função processadora. Então definimos

```
typedef void (*DispatchFunc)(char **);
DispatchFunc requests[LIST_HISTORY + 1];
```

De seguida só temos de associar a cada request a sua função processadora.

```
typedef void (*DispatchFunc)(char **);
DispatchFunc req_dispatch[LIST_HISTORY + 1];

req_dispatch[SET_PIPE_TIMEOUT] = process_pipe_timeout;
req_dispatch[SET_EXEC_TIMEOUT] = process_exec_timeout;
req_dispatch[EXECUTE_TASK] = process_exec_task;
req_dispatch[LIST_IN_EXECUTION] = process_list_execs;
req_dispatch[TERMINATE_TASK] = process_kill_task;
req_dispatch[LIST_HISTORY] = process_list_history;
```

Com este modelo possuimos a capacidade de uma escabilidade muito maior, tornando cada componente no seu formato mais atómico, então as funções podem ser definidas da seguinte forma:

```
void process_pipe_timeout(char** argv) {
  g_pipe_timeout = atoi(argv[0]);
}

void process_exec_timeout(char** argv) {
  g_exec_timeout = atoi(argv[0]);
}

void process_exec_task(char** argv) {
  // Devera executar um novo processo

  // Novo processo serializa task e comunica o respectivo id.
}

void process_list_execs(char** argv) {
  // Criar novo processo

  // Processo comunica tasks em aberto.
}

void process_kill_task(char** argv) {
  // Envia um SIGKILL a uma task.

  // Assinala comando como COMMAND_TERMINATED
}

void process_list_history(char** argv) {
  // Faz dump do processo de historico.
}
```

Neste contexto de dispatch dinâmico, basta fazer o seguinte excerto de código para processar qualquer pedido vindo do pipe.

```
while( coisas ) {
  N = read(in_pipe_fd, buffer, MAX_BUFFER_SIZE);
  request = materialize_request(buffer, N);

  if(request != NULL) {
    // Conseguiu materializar, efetua dispatch
    (*req_dispatch[request->ID])(request->argv);
  } else {
    // Erro, nao conseguiu materializar
    write(2, "ERROR: shit happend\n", ... );
  }
}
```

Porém, o que acontece no caso do servidor querer dar a informação resultante de volta ao cliente? Para tal basta que existam dois pipes com nome, em cada sentido, então, o servidor simplesmente escreve o que deseja no nó, sem esperar nada em retorno. Assumindo que no cliente existe um processo a fazer *scanning* do pipe, e sempre que capta alguma informação, automaticamente reproduz essa informação no *stdout*. Por exemplo, para comunicar dados de volta, basta fazer:

```
void process_list_history(char** argv) {
  // Ler o historico pode implicar ler um ficheiro grande
  // O que pode ser penoso, daí destaca-se um processo só Para
  // este efeito.
  if(!fork()) {
    ssize_t size;
    char * list = get_list_somehow(&size);
    write(out_pipe_fd, list, size);
    _exit(0);
  }
}
```