Today

- advanced RPC
- RPC implementation
- portmap
- UNIX domain sockets

RPC review

- emulate function-call
- hide details of network
- hide details of hetergeneous machines
- several levels of implementation
- part of a DCE

advanced RPC

- broadcast discovery (service brokers) – find a server on the LAN
- batch client requests
- at-most-once (*svcudp_enablecache()*)
- asynchronous client (call-back)
- server concurrency (inetd)
- transport-independent RPC (TIRPC) (rpcbind)

broadcast

client main

```
...
   if (argc > 1){
       server = argv[1];
      if ( (cl = clnt_create(server, THD_PROG, THD_VERS, "udp"))
          == NULL) {
         clnt_pcreateerror(server);
         exit(2);
      }
   } else if (locate_server() == ERROR){
      printf("unable to locate server\n");
      exit(3);
   }
...
```

```
int locate_server()
{
   enum clnt_stat stat;
   int retval = OK;

   /* make a broadcast call looking for our version of the server */
   stat = clnt_broadcast(THD_PROG, THD_VERS, (u_long) 0,
     xdr_void, NULL, xdr_void, NULL, each_server);

   /* check whether we found a server */
   if(stat != RPC_SUCCESS) { retval = ERROR; }
   return(retval);
}

bool_t each_server(out, addr)
char *out;
struct sockaddr_in *addr;
{
   struct sockaddr_in server_addr;
   char *server, *inet_ntoa();
   /* save the address */
   bcopy((char *)addr, (char *)&server_addr,
   sizeof(struct sockaddr_in));
   server = inet_ntoa(addr->sin_addr);
   printf("Got a response from %s\n", server);

   if((cl = clnt_create(server, THD_PROG, THD_VERS, "udp")) == NULL) {
           return((bool_t) 0);
   } else { return((bool_t) 1); }
}
```


RPC implementation

- server registers with portmap (actually using RPC)
- client asks portmap for port to use for service
- client does RPC with server
- library takes care of UDP timeout/retransmission
- XDR used for data conversion

there's no magic, it's TCP/UDP underneath

RPC message

/usr/include/rpc/rpc_msg.h

```
struct rpc_msg {
        u_long                  rm_xid;
        enum msg_type           rm_direction;
        union {
                struct call_body RM_cmb;
                struct reply_body RM_rmb;
        } ru;

struct call_body {
        u_long cb_rpcvers;      /* must be equal to two */
        u_long cb_prog;
        u_long cb_vers;
        u_long cb_proc;
        struct opaque_auth cb_cred;
        struct opaque_auth cb_verf; /* protocol specific - provided by client */
};

struct reply_body {
        enum reply_stat rp_stat;
        union {
                struct accepted_reply RP_ar;
                struct rejected_reply RP_dr;
        } ru;
```



```
  cetus1a% rpcdate altair
  time on host altair = 941145633
  time on host altair = Thu Oct 28 17:20:33 1999

CETUS1A.41375 > ALTAIR.sunrpc: udp 56 (DF)
            4500 0054 0b7a 4000 ff11 b37f 80a9 5e15
            80a9 5d37 a19f 006f 0040 d89e 3815 92c7
            0000 0000 0000 0002 0001 86a0 0000 0002
            0000 0003 0000 0000 0000 0000 0000 0000
            0000 0000 3123 4567 0000 0001 0000 0011
            0000
ALTAIR.sunrpc > CETUS1A.41375: udp 28
            4500 0038 a8b3 0000 3c11 1963 80a9 5d37
            80a9 5e15 006f a19f 0024 0000 3815 92c7
            0000 0001 0000 0000 0000 0000 0000 0000
            0000 0000 0000 12fa


CETUS1A.41375 > ALTAIR.4858: udp 40 (DF)
            4500 0044 0b7b 4000 ff11 b38e 80a9 5e15
            80a9 5d37 a19f 12fa 0030 4ce9 3815 92c8
            0000 0000 0000 0002 3123 4567 0000 0001
            0000 0001 0000 0000 0000 0000 0000 0000
            0000 0000
ALTAIR.4858 > CETUS1A.41375: udp 28
            4500 0038 a8b5 0000 3c11 1961 80a9 5d37
            80a9 5e15 12fa a19f 0024 0000 3815 92c8
            0000 0001 0000 0000 0000 0000 0000 0000
            0000 0000 3818 be21
```

```
CETUS1A.41375 > ALTAIR.4858: udp 44 (DF)
            4500 0048 0b7c 4000 ff11 b389 80a9 5e15
            80a9 5d37 a19f 12fa 0034 56a5 3815 92c9
            0000 0000 0000 0002 3123 4567 0000 0001
            0000 0002 0000 0000 0000 0000 0000 0000
            0000 0000 3818 be21
ALTAIR.4858 > CETUS1A.41375: udp 56
            4500 0054 a8b6 0000 3c11 1944 80a9 5d37
            80a9 5e15 12fa a19f 0040 0000 3815 92c9
            0000 0001 0000 0000 0000 0000 0000 0000
            0000 0000 0000 0019 5468 7520 4f63 7420
            3238 2031 373a 3230 3a33 3320 3139 3939
            0a00
```


portmap

- started at boot time BEFORE inetd
- directory service RPC (port 111)
- an RPC server itself
- register, unregister, locate services
- list all services, call-through for broadcast service queries (e.g., yp name server)
- library of functions **pmap_set(), pmap_unset(), pmap_getport(), pmap_getmaps(), pmap_rmtcall()**

rpc_pmap_prot.h

```
#define PMAPPORT                ((u_short)111)
#define PMAPPROG                ((u_long)100000)
#define PMAPVERS                ((u_long)2)
#define PMAPVERS_PROTO          ((u_long)2)
#define PMAPVERS_ORIG           ((u_long)1)
#define PMAPPROC_NULL           ((u_long)0)
#define PMAPPROC_SET            ((u_long)1)
#define PMAPPROC_UNSET          ((u_long)2)
#define PMAPPROC_GETPORT        ((u_long)3)
#define PMAPPROC_DUMP           ((u_long)4)
#define PMAPPROC_CALLIT         ((u_long)5)

struct pmap {
        long unsigned pm_prog;
        long unsigned pm_vers;
        long unsigned pm_prot;
        long unsigned pm_port;
};
struct pmaplist {
        struct pmap     pml_map;
        struct pmaplist *pml_next;
};
```


portmap.c

```
main()
{
   SVCXPRT *xprt;
   int sock, pid, t;
   struct sockaddr_in addr;
   int len = sizeof(struct sockaddr_in);

#ifndef DEBUG
   pid = fork();
   if (pid < 0) {
      perror("portmap: fork");
      exit(1);
   }
   if (pid != 0)
      exit(0);
   for (t = 0; t < 20; t++)
      close(t);
   open("/", 0);
   dup2(0, 1);
   dup2(0, 2);
   t = open("/dev/tty", 2);
   if (t >= 0) {
      ioctl(t, TIOCNOTTY, (char *)0);
      close(t);
   }
#endif
```

portmap.c

```
if ((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
   perror("portmap cannot create socket");
   exit(1);
}

addr.sin_addr.s_addr = 0;
addr.sin_family = AF_INET;
addr.sin_port = htons(PMAPPORT);
if (bind(sock, (struct sockaddr *)&addr, len) != 0) {
   perror("portmap cannot bind");
   exit(1);
}

if ((xprt = svcudp_create(sock)) == (SVCXPRT *)NULL) {
   fprintf(stderr, "couldn't do udp_create\n");
   exit(1);
}
```


portmap.c

```
   if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
      perror("portmap cannot create socket");
      exit(1);
   }
   if (bind(sock, (struct sockaddr *)&addr, len) != 0) {
      perror("portmap cannot bind");
      exit(1);
   }
   if ((xprt = svctcp_create(sock, RPCSMALLMSGSIZE, RPCSMALLMSGSIZE)
       == (SVCXPRT *)NULL) {
      fprintf(stderr, "couldn't do tcp_create\n");
      exit(1);
   }

        (void)svc_register(xprt, PMAPPROG, PMAPVERS, reg_service, FA
   svc_run();
   fprintf(stderr, "run_svc returned unexpectedly\n");
   abort();
}
```


portmap.c

```
reg_service(rqstp, xprt)
   struct svc_req *rqstp;
   SVCXPRT *xprt;
{
   switch (rqstp->rq_proc) {
   case PMAPPROC_NULL:
      if ((!svc_sendreply(xprt, xdr_void, NULL)) && debugging) {
         abort();
      }
      break;
   case PMAPPROC_SET:
      if (!svc_getargs(xprt, xdr_pmap, &reg))
         svcerr_decode(xprt);
      else {
         fnd = find_service(reg.pm_prog, reg.pm_vers, reg.pm_prot);
         if (fnd && fnd->pml_map.pm_vers == reg.pm_vers) {
            if (fnd->pml_map.pm_port == reg.pm_port) {
               ans = 1;
               goto done;
            }
            else {
               ans = 0;
               goto done;
            }
         } else {
```


portmap.c

```
            pml = (struct pmaplist *)
                malloc((u_int)sizeof(struct pmaplist));
            pml->pml_map = reg;
            pml->pml_next = 0;
            if (pmaplist == 0) {
               pmaplist = pml;
            } else {
               for (fnd= pmaplist; fnd->pml_next != 0;
                   fnd = fnd->pml_next);
               fnd->pml_next = pml;
            }
            ans = 1;
         }
      done:
         if ((!svc_sendreply(xprt, xdr_long, (caddr_t)&ans)) &&
             debugging) {
            fprintf(stderr, "svc_sendreply\n");
            abort();
         }
      }
      break;
```

portmap.c

```
case PMAPPROC_UNSET:
   if (!svc_getargs(xprt, xdr_pmap, &reg))
      svcerr_decode(xprt);
   else {
      ans = 0;
      for (prevpml = NULL, pml = pmaplist; pml != NULL; ) {
         if ((pml->pml_map.pm_prog != reg.pm_prog) ||
            (pml->pml_map.pm_vers != reg.pm_vers)) {
            prevpml = pml;
            pml = pml->pml_next;
            continue;
         }
         /* found it; pml moves forward, prevpml stays */
         ans = 1;
         t = (caddr_t)pml;
         pml = pml->pml_next;
         if (prevpml == NULL)
            pmaplist = pml;
         else
            prevpml->pml_next = pml;
         free(t);
      }
      if ((!svc_sendreply(xprt, xdr_long, (caddr_t)&ans)) &&
          debugging) {
         fprintf(stderr, "svc_sendreply\n");
         abort();
      }
   }
   break;
```


portmap.c

```
case PMAPPROC_GETPORT:
   /*
    * Lookup the mapping for a program,version and return its por
    */
   if (!svc_getargs(xprt, xdr_pmap, &reg))
      svcerr_decode(xprt);
   else {
      fnd = find_service(reg.pm_prog, reg.pm_vers, reg.pm_prot);
      if (fnd)
         port = fnd->pml_map.pm_port;
      else
         port = 0;
      if ((!svc_sendreply(xprt, xdr_long, (caddr_t)&port)) &&
          debugging) {
         fprintf(stderr, "svc_sendreply\n");
         abort();
      }
   }
   break;
```


portmap.c

```
find_service(prog, vers, prot)
   u_long prog;
   u_long vers;
{
   register struct pmaplist *hit = NULL;
   register struct pmaplist *pml;

   for (pml = pmaplist; pml != NULL; pml = pml->pml_next) {
      if ((pml->pml_map.pm_prog != prog) ||
         (pml->pml_map.pm_prot != prot))
         continue;
      hit = pml;
      if (pml->pml_map.pm_vers == vers)
          break;
   }
   return (hit);
}
```


portmap.c

```
case PMAPPROC_DUMP:
   /*
    * Return the current set of mapped program,version
    */
   if (!svc_getargs(xprt, xdr_void, NULL))
      svcerr_decode(xprt);
   else {
      if ((!svc_sendreply(xprt, xdr_pmaplist,
           (caddr_t)&pmaplist)) && debugging) {
         fprintf(stderr, "svc_sendreply\n");
         abort();
      }
   }
   break;
case PMAPPROC_CALLIT:
   callit(rqstp, xprt);
   break;
```

portmap.c

```
callit(rqstp, xprt)
{
   struct authunix_parms *au = (struct authunix_parms *)rqstp->rq_clntcred;
   struct rmtcallargs a;
   a.rmt_args.args = buf;
   if (!svc_getargs(xprt, xdr_rmtcall_args, &a))
       return;
   if ((pml = find_service(a.rmt_prog, a.rmt_vers, IPPROTO_UDP)) == NULL)
       return;
   port = pml->pml_map.pm_port;
   get_myaddress(&me);
   me.sin_port = htons(port);
   client = clntudp_create(&me, a.rmt_prog, a.rmt_vers, timeout, &socket);
   if (client != (CLIENT *)NULL) {
      if (rqstp->rq_cred.oa_flavor == AUTH_UNIX) {
         client->cl_auth = authunix_create(au->aup_machname,
            au->aup_uid, au->aup_gid, au->aup_len, au->aup_gids);
      }
      a.rmt_port = (u_long)port;
      if (clnt_call(client, a.rmt_proc, xdr_opaque_parms, &a,
          xdr_len_opaque_parms, &a, timeout) == RPC_SUCCESS) {
         svc_sendreply(xprt, xdr_rmtcall_result, &a);
      }
      AUTH_DESTROY(client->cl_auth);
      clnt_destroy(client);
   }
   (void)close(socket);
}
```


rpcinfo

- query portmap
- rpcinfo -p hostname

```
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100029    1   udp    660  keyserv
    100003    2   udp   2049  nfs
    100026    1   udp    721  bootparam
    100024    1   udp    725  status
    100024    1   tcp    727  status
    100021    1   tcp    728  nlockmgr
    100021    1   udp   1028  nlockmgr
    100010    1   udp    740  etherstatd
    100007    2   tcp   1024  ypbind
    100001    2   udp   1041  rstatd
    100012    1   udp   1043  sprayd
    100008    1   udp   1044  walld
    100005    1   udp    932  mountd
 824395109    1   udp   1354
1073741824    1   udp   1880
 536870918    1   udp   2067
```


rpcinfo.c

```
pmapdump(argc, argv)
   if (argc == 2) {
      hostnm = argv[1];
   } else {
      gethostname(hoststr, sizeof(hoststr));
      hostnm = hoststr;
   }
   if ((hp = gethostbyname(hostnm)) == NULL) {
      fprintf(stderr, "cannot get addr for '%s'\n", hostnm);
      exit(0);
   }
   bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
   server_addr.sin_family = AF_INET;
   minutetimeout.tv_sec = 60;
   minutetimeout.tv_usec = 0;
   server_addr.sin_port = htons(PMAPPORT);
   if ((client = clnttcp_create(&server_addr, PMAPPROG,
       PMAPVERS, &socket, 50, 500)) == NULL) {
      clnt_pcreateerror("rpcinfo: can't contact portmapper");
      exit(1);
   }
   if ((rpc_stat = clnt_call(client, PMAPPROC_DUMP, xdr_void, NULL,
       xdr_pmaplist, &head, minutetimeout)) != RPC_SUCCESS) {
      fprintf(stderr, "rpcinfo: can't contact portmapper: ");
      clnt_perrno(rpc_stat);
      fprintf(stderr, "\n");
      exit(1);
   }
```


rpcinfo.c

```
   if (head == NULL) {
      printf("No remote programs registered.\n");
   } else {
      printf("   program vers proto   port\n");
      for (; head != NULL; head = head->pml_next) {
         printf("%10ld%5ld",
             head->pml_map.pm_prog,
             head->pml_map.pm_vers);
         if (head->pml_map.pm_prot == IPPROTO_UDP)
            printf("%6s",  "udp");
         else if (head->pml_map.pm_prot == IPPROTO_TCP)
            printf("%6s", "tcp");
         else
            printf("%6ld",  head->pml_map.pm_prot);
         printf("%7ld",  head->pml_map.pm_port);
         rpc = getrpcbynumber(head->pml_map.pm_prog);
         if (rpc)
            printf("  %s\n", rpc->r_name);
         else
            printf("\n");
      }
   }
}
```

RPC

- remote procedures
- hides communication and data conversion
- dynamic port assignment and service discovery
- funny syntax, hard to debug
- performance/latency
- DCE RPC is similar

just a wrapper for TCP/UDP/IP

UNIX domain

interprocess communication

- IPC within a machine (other: pipe, FIFO, sysV messages)
- use the socket paradigm (AF_UNIX)
- faster than AF_INET
- stream and datagram
- client MUST bind
- datagrams can be discarded! (ENOBUFFS)
- 4.3BSD uses them to implement pipes
- can pass access rights (file descriptors)
- name space is path names
- a file is created
  try file /dev/printer

used by: syslog, lpr, pvm

sock structs

```
struct sockaddr {
        u_short sa_family;              /* address family */
        char    sa_data[14];            /* up to 14 bytes of direct address */
};


struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
}


struct  sockaddr_un {
        short   sun_family;             /* AF_UNIX */
        char    sun_path[108];          /* path name (gag) */
};
```


unix.h

```
#include        <stdio.h>
#include        <sys/types.h>
#include        <sys/socket.h>
#include        <sys/un.h>

#define UNIXSTR_PATH    "/tmp/s.unixstr"
#define UNIXDG_PATH     "/tmp/s.unixdg"
#define UNIXDG_TMP      "/tmp/dg.XXXXXX"
```

```
/* unixdgcli.c */
main(argc, argv)
int   argc;
char  *argv[];
{
   int         sockfd, clilen, servlen;
   char        *mktemp();
   struct sockaddr_un   cli_addr, serv_addr;
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sun_family = AF_UNIX;
   strcpy(serv_addr.sun_path, UNIXDG_PATH); /* server */
   servlen = sizeof(serv_addr.sun_family) +
             strlen(serv_addr.sun_path);
   if ( (sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
      err_sys("client: can't open datagram socket");
   bzero((char *) &cli_addr, sizeof(cli_addr)); /* zero out */
   cli_addr.sun_family = AF_UNIX;
   strcpy(cli_addr.sun_path, UNIXDG_TMP);
   mktemp(cli_addr.sun_path);
   clilen = sizeof(cli_addr.sun_family)
      + strlen(cli_addr.sun_path);
   if (bind(sockfd,(struct sockaddr *)&cli_addr, clilen) < 0)
      err_sys("client: can't bind local address");
   dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, servlen);
   close(sockfd);
   unlink(cli_addr.sun_path);
}
```


dgcli.c +

```
dg_cli(fp, sockfd, pserv_addr, servlen)
FILE     *fp;
int      sockfd;
struct sockaddr   *pserv_addr;   /* ptr to sockaddr_XX */
int servlen;   /* actual sizeof(*pserv_addr) */
{
   int n,fromlen;
   struct sockaddr from;
   char  sendline[MAXLINE], recvline[MAXLINE + 1];

   while (fgets(sendline, MAXLINE, fp) != NULL) {
      n = strlen(sendline);
      if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n)
         err_sys("dg_cli: sendto error on socket");
      fromlen = sizeof(from);  /* +++ */
      n = recvfrom(sockfd, recvline, MAXLINE, 0,&from,&fromlen);
      if (n < 0)
         err_sys("dg_cli: recvfrom error");
      recvline[n] = 0;  /* null terminate */
      fputs(recvline, stdout);
   }

   if (ferror(fp))
      err_sys("dg_cli: error reading file");
}
```


unixdgserv.c

```
main(argc, argv)
int   argc;
char  *argv[];
{
   int         sockfd, servlen;
   struct sockaddr_un   serv_addr, cli_addr;

   if ( (sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
      err_sys("server: can't open datagram socket");
   unlink(UNIXDG_PATH); /* in case it was left from last time */
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sun_family = AF_UNIX;
   strcpy(serv_addr.sun_path, UNIXDG_PATH);
   servlen = sizeof(serv_addr.sun_family) +
             strlen(serv_addr.sun_path);
   if (bind(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
      err_sys("server: can't bind local address");

   dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));
      /* NOTREACHED */
}
```


dgecho.c

```
dg_echo(sockfd, pcli_addr, maxclilen)
int      sockfd;
struct sockaddr   *pcli_addr; /*  sockaddr_XX structure */
int      maxclilen;  /* sizeof(*pcli_addr) */
{
   int   n, clilen;
   char  mesg[MAXMESG];

   for ( ; ; ) {
      clilen = maxclilen;
      n = recvfrom(sockfd, mesg, MAXMESG, 0, pcli_addr, &clilen);
      if (n < 0)
         err_sys("dg_echo: recvfrom error");

      if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n)
         err_sys("dg_echo: sendto error");
   }
}
```

syslog()

```
char     logname[] = "/dev/log";
struct sockaddr SyslogAddr;
...
  SyslogAddr.sa_family = AF_UNIX;
  strncpy(SyslogAddr.sa_data, logname,
     sizeof SyslogAddr.sa_data);
  LogFile = socket(AF_UNIX, SOCK_DGRAM, 0);
  ....
  sendto(LogFile, outline, c, 0,
     &SyslogAddr, sizeof SyslogAddr);
```


lpr

```
#define SOCKETNAME "/dev/printer"
   struct sockaddr_un s_un;
   s = socket(AF_UNIX, SOCK_STREAM, 0);
   if (s < 0) {
      perr("socket");
      return (0);
   }
   s_un.sun_family = AF_UNIX;
   strcpy(s_un.sun_path, SOCKETNAME);
   if (connect(s, (struct sockaddr *)&s_un,
     strlen(s_un.sun_path)+2) < 0) {
      perr("connect");
      (void) close(s);
      return (0);
   }
   (void) sprintf(buf, "\1%s\n", printer);
   n = strlen(buf);
   if (write(s, buf, n) != n) {
      perr("write");
      (void) close(s);
      return (0);
   }
   if (read(s, buf, 1) == 1) {
      if (buf[0] == '\0') {      /* everything is OK */
         (void) close(s);
         return (1);
      }
...
```


next time

reliable UDP, broadcast, multicast