

Writing parallel applications with PHP

Andrey Hristov
100 DAYS – Software Projects

International PHP Conference 2004
Nov 10th, 2004

About me

- BSc in Computer Engineering
- Student at University of Applied Sciences, Stuttgart, Germany
- Practicing web programming since year 2000
- Author of pecl/stats
- Working for 100 Days, Germany

Survey

How many of you know about IPC in
means of
shared memory, semaphores,
message queues?

How many of you have worked
with these in PHP?

Agenda

- Terminology
- Semaphores
- Shared memory
- Message queues
- Signals, forking
- PHP API
- Procedural examples of IPC primitives
- OO approach to IPC
- Use cases

Processes and threads

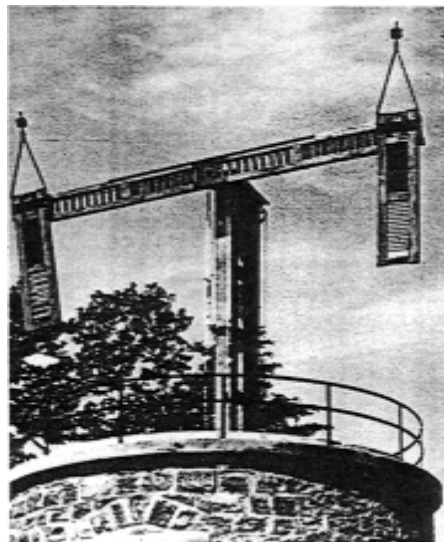
- A process is an entity residing in memory which has its own stack, heap and code segment. A process is not a program. The latter is a file on disk.
- Processes are scheduled for execution by the OS. Always in one of (usually) 3 states : ready, running, blocked.
- Threads are sometimes called light-weight processes (LWP). They live inside a single process.
- Threads implementation is either on user level or in the kernel of the OS (monolithic kernels).
- Problem with kernels unaware of threads is blocking of a whole process when a thread is being blocked. No other thread will have chance to use a timeslice till the process becomes "ready".

Deadlocks, race conditions, reentrancy and critical sections.

- Deadlock is a state of 2 or more processes waiting forever for a specific resource. Both processes hold locks but only the total amount of locks is enough to access the resource. If both processes don't release the locks then the deadlock is endless. Preventing algorithms exist.
- Critical section is a piece of code that has to be executed by only one thread (process) at a time. This is performed with synchronization primitives.
- Code part is reentrant if several threads can execute it at the same time. If the code is not written with reentrancy in mind race conditions may appear.
- Some operations has to be executed atomically. If such a operation is not atomically written, the OS scheduler may break the execution of a thread in the middle and start another one, which accesses inconsistent data.

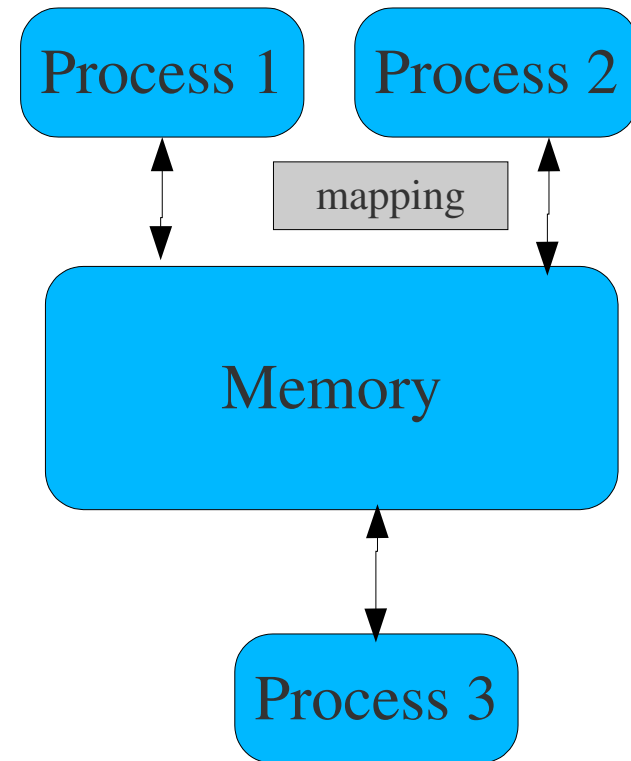
Semaphores

- Invented by E. Dijkstra . P(s) and V(s) operations
- Used for synchronization
- Binary semaphores are also called mutexes or locks.
- Non-binary semaphores



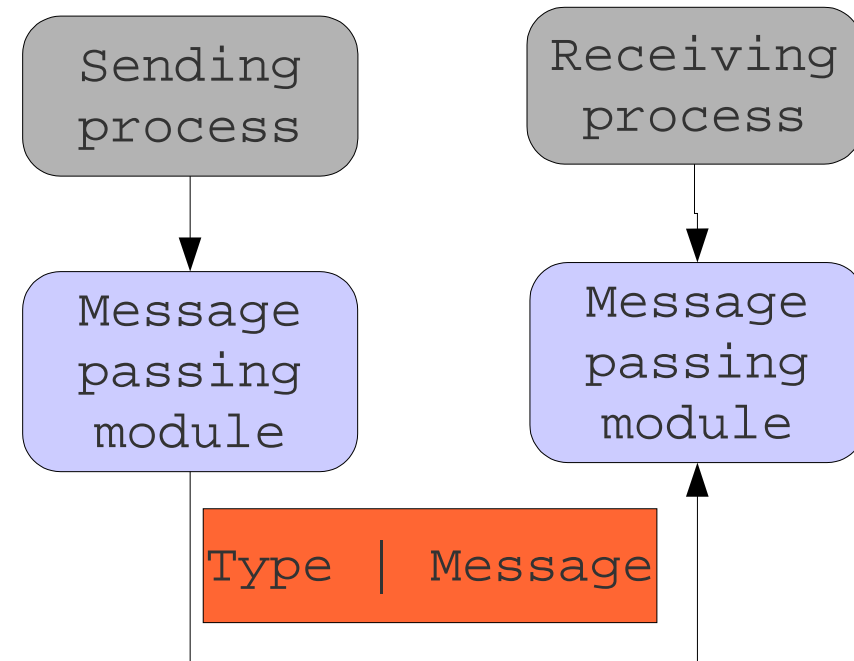
Shared memory

- Memory segments available for more than one process.
- The processes map their virtual space to the shared memory segment.
- In System V implementations the access is permission based.
- Synchronization is needed in cases of mixed reads/writes. Mutex must be used. Remember Murphy's Law: "If anything can go wrong, it will"



Message queues

- Simple way of data exchange: send messages.
- IPC messaging allows sending and receiving in arbitrary order – on the contrary of reading from a stream. One can fetch an element from the queue even if it's not the first one.
- Blocking and non-blocking receive.



Multiprocessing or multithreading with PHP?

- Threads are good since they are lighter than processes and one can easily share memory. Unfortunately, *PHP* is not threads- ready (wasn't written with them in mind).
- On the other hand 2 or more processes may work in parallel and synchronize and communicated by using IPC primitives.
- Process control extension (`--enable-pcntl`) is available since 4.1.0 (*nix). It gives the possibility to create new process out of the current *PHP* interpreter or so called forking or spawning. *C* programmers are familiar with these concepts. The forked process has a copy of the memory of the process that did fork. All open files are still open including `stdin`, `stdout` and `stderr`. Still they are 2 different processes with independent memory areas.

PHP API for System V IPC (1)

- `ext/sysvshm` `--enable-sysvshm` ($\geq 3.0.6$)
- `ext/sysvsem` `--enable-sysvsem` ($\geq 3.0.6$)
- `ext/sysvmsg` `--enable-sysvmsg` ($\geq 4.3.0$)

PHP API for System V IPC (2)

- shm_attach()
- shm_detach()
- shm_get_var()
- shm_put_var()
- shm_remove_var()
- shm_remove()
- sem_acquire()
- sem_get()
- sem_release()
- sem_remove()
- msg_get_queue()
- msg_receive()
- msg_remove_queue()
- msg_send()
- msg_set_queue()
- msg_stat_queue()

Process control extension API

- `pcntl_alarm()` Creates a timer (SIGALRM). Only one per process.
- `pcntl_fork()` Forks the interpreter. 2 scripts run independently.
- `pcntl_exec()` Executes an external program in the current process.
- `pcntl_getpriority()` Returns the priority of a process (the current or other).
- `pcntl_setpriority()` Sets the priority of a process (the current or other).
- `pcntl_signal()` Installs a signal handler.
- `pcntl_wait()` Waits on or returns the *status* and the pid of a child.
- `pcntl_waitpid()` Like `pcntl_wait()` but for specific child
- `pcntl_wexitstatus()` Returns the return code, of a SIGTERM child, from *status*
- `pcntl_wifexited()` Returns TRUE if *status* code was successful exit.
- `pcntl_wifsignaled()` Returns TRUE if *status* is due to exit caused by a signal.
- `pcntl_wifstopped()` Returns TRUE if the child is currently stopped.
- `pcntl_wstopsig()` Returns the signal which caused the child to stop.
- `pcntl_wtermsig()` Returns the signal which caused the child to terminate.

ext/shmop

- Written by Ilia Alshanetsky & Slava Poliakov
- Works on Win32 and *nix
- Better interoperability with the non-PHP world.
- Since PHP 4.0.3. The current naming is since 4.0.4 .
- shmop_open()
- shmop_close()
- shmop_delete()
- shmop_read()
- shmop_write()
- shmop_size()

Working with semaphores (procedural way) (0)

```
<?php
$key = ftok(__FILE__, 'R'); var_dump($key);
$sem_id = sem_get($key, 1, 0666);
if ($argc == 2 && $argv[1] == "clear") {
    echo "Clearing semaphore : $key\n";
    sem_remove($sem_id); exit;
}
$res = sem_acquire($sem_id);
var_dump(strftime('TIME : %H:%M:%S', (int)microtime(1)));
sleep(5);
$res = sem_release($sem_id);
var_dump(strftime('TIME : %H:%M:%S', (int)microtime(1)));
?>
```

```
andrey@poohie:~/test/php> php sem_create.php
int(1376196715)
string(16) "TIME : 16:12:05"
string(16) "TIME : 16:12:10"
```

```
andrey@poohie:~/test/php> php sem_create.php
int(1376196715)
string(16) "TIME : 16:12:10"
string(16) "TIME : 16:12:15"
```

Working with shared memory (procedural way) (0)

```
<?php
$key = ftok(__FILE__, 'R'); var_dump($key);
$shm_id = shm_attach($key, 1024, 0666);
if ($argc == 2 && $argv[1] == "clear") {
    echo "Clearing shared memory : $key\n";
    shm_detach($shm_id); exit;
}
$var = strftime('TIME : %H:%M:%S', (int)microtime(1));
echo "Putting into SHM : ";
var_dump($var);
$res = shm_put_var($shm_id, 1/*var_key*/, $var);
var_dump($res);
$var2 = shm_get_var($shm_id, 1/*var_key*/);
echo "Read from SHM : ";
var_dump($var2);
shm_remove_var($shm_id, 1);

?>
```

```
andrey@poohie:~/test/php> php shm_create.php
int(1386196715)
Putting into SHM : string(16) "TIME : 17:17:54"
bool(true)
Read from SHM : string(16) "TIME : 17:17:54"
```

Working with shared memory (procedural way) (1)

```
<?php
$key = ftok('/home/andrey/test/phpconf/shm_create2.php', 'R');
$shm_id = shm_attach($key, 1024, 0666);
$var = strftime('TIME : %H:%M:%S', (int)microtime(1));
echo "Putting into SHM : ";
var_dump($var);
$res = shm_put_var($shm_id, 1/*var_key*/, $var);
var_dump($res);
sleep(10);
$var2 = shm_get_var($shm_id, 1/*var_key*/);
echo "Read from SHM : ";
var_dump($var2);
?>
```

```
andrey@poohie:~/test/php> php shm_create2.php
Putting into SHM : string(16) "TIME : 17:54:47"
bool(true)
Read from SHM : string(16) "TIME : 17:54:51"
```

```
andrey@poohie:~/test/php> php shm_create2.php
Putting into SHM : string(16) "TIME : 17:54:51"
bool(true)
Read from SHM : string(16) "TIME : 17:54:51"
```

Working with shared memory (procedural way) (2)

```
<?php
$key = ftok(__FILE__, 'R');
$shm_id = shm_attach($key, 1024, 0666);
$sem_id = sem_get($key, 1, 0666);
if ($argc == 2 && $argv[1] == "clear") {
    echo "Clearing shared memory / semaphore : $key\n";
    shm_detach($shm_id);sem_remove($sem_id);exit;
}
$var = strftime('TIME : %H:%M:%S', (int)microtime(1));
echo "Putting into SHM : ";var_dump($var);
sem_acquire($sem_id);
printf("Acquired access @%s:\n", strftime('TIME : %H:%M:%S', (int)microtime(1)));
$res = shm_put_var($shm_id, 1/*var_key*/, $var);
sleep(10); // sleep 10 seconds
$var2 = shm_get_var($shm_id, 1/*var_key*/);
sem_release($sem_id);
echo "Read from SHM : ";var_dump($var2);
?>
```

```
andrey@poohie:~/test/php> php shm_create3.php
Putting into SHM : string(16) "TIME : 18:03:23"
Acquired access @TIME : 18:03:23:
Read from SHM : string(16) "TIME : 18:03:23"
```

```
andrey@poohie:~/test/php> php shm_create3.php
Putting into SHM : string(16) "TIME : 18:03:25"
Acquired access @TIME : 18:03:33:
Read from SHM : string(16) "TIME : 18:03:25"
```

Working with a message queue

Server :

```
<?php
$key = ftok("/home/andrey/test/phpconf/mqueue.php", 'R');
$queue = msg_get_queue($key, 0666);
$message = NULL;
$error = NULL;
msg_receive($queue, 1/*desired*/, $real_type, 16384, $message, 1/*ser*/,0,$error)
echo "Received : ";var_dump($message);
echo "Error code : ";var_dump($error);
?>
```

Client :

```
<?php
$key = ftok("/home/andrey/test/phpconf/mqueue.php", 'R');
$queue = msg_get_queue($key, 0666);
$message = "Hello World!";
$error = 0;
echo "Sending :";var_dump($message);
msg_send($queue, 1/*msg_type*/, $message, 1/*ser*/, TRUE,$error);
var_dump($error);
?>
```

```
andrey@poohie:~/test/php> php mqueue_client.php
Sending :string(12) "Hello World!"
int(0)
```

```
andrey@poohie:~/test/php> php mqueue.php
Received : string(12) "Hello World!"
Error code : int(0)
```

ext/shmop example (0)

```
<?php
// Create 100 byte shared memory block
$shm_id = shmop_open(ftok(__FILE__, 'R'), "c"/*mode*/, 0644/*rights*/,
100/*size*/);
if (!$shm_id) {
    echo "Couldn't create shared memory segment\n";exit;
}
// Get shared memory block's size
$shm_size = shmop_size($shm_id);
echo "SHM Block Size: " . $shm_size . " has been created.\n";
// Lets write a test string into shared memory
$shm_bytes_written = shmop_write($shm_id, "my shared memory block", 0);
if ($shm_bytes_written != strlen("my shared memory block")) {
    echo "Couldn't write the entire length of data\n";exit;
}
// Now lets read the string back
$my_string = shmop_read($shm_id, 0/*start*/, $shm_size/*count*/);
if (!$my_string) { echo "Couldn't read from shared memory block\n"; exit; }
echo "The data inside shared memory was: " . $my_string . "\n";
//Now lets delete the block and close the shared memory segment
if (!shmop_delete($shm_id)) {
    echo "Couldn't mark shared memory block for deletion.";exit;
}
shmop_close($shm_id);
?>
```

```
andrey@poohie:~/test/php> php shmop.php
SHM Block Size: 100 has been created.
The data inside shared memory was: my shared memory block
```

ext/shmop example (1) : semaphore emulation

```
<?php
$key = ftok(__FILE__, "K");
error_reporting(0);
do {
    $sem_id = shmop_open($key, "n", 0644, 10);
    usleep(100);
} while ($sem_id === false);
error_reporting(E_ALL);
echo "Entered critical section at :".strftime("%H:%M:%S\n", time());
sleep(5);
shmop_delete($sem_id);
echo "Exited critical section at :".strftime("%H:%M:%S\n", time());
?>
```

```
andrey@poohie:~/test/phpconf> php shmop2.php
Entered critical section at :16:50:41
Exited critical section at :16:50:46
```

```
andrey@poohie:~/test/phpconf> php shmop2.php
Entered critical section at :16:50:46
Exited critical section at :16:50:51
```

ext/pcntl example

```
<?php
declare(ticks = 1); //since 4.3.0
$seconds = 2;
function logit($msg){ echo getmypid()." ".$msg." at ". strftime("%H:%M:%S", time())."\n";}
function signal_handler($signal) {
    switch($signal) {
        case SIGTERM:logit("Caught SIGTERM");exit;
        case SIGINT :logit("Caught SIGINT");exit;
    }
}
function alarm_handler($signal) {
    logit("Got SIGALRM");
    pcntl_alarm($GLOBALS['seconds']);
}
pcntl_signal(SIGTERM, "signal_handler");
pcntl_signal(SIGINT, "signal_handler");
pcntl_signal(SIGALRM, "alarm_handler");
pcntl_alarm($seconds);
while ($i++ < 3) {
    sleep(5);
    logit("After sleep");
}
pcntl_signal(SIGALRM, SIG_IGN); //ignore, reenabled in child
switch ($pid = pcntl_fork()) {
    case -1:die('Error');
    case 0:pcntl_signal(SIGALRM, "alarm_handler");
        pcntl_alarm($seconds);while (1) sleep(5);break;
    default:logit("Sleeping 5s");sleep(5);logit("Killing $pid");
        posix_kill($pid, SIGTERM);
        $st = pcntl_wait($pid); //kill and get status
        echo "status  :";var_dump($st);
        echo "exited  :";var_dump(pcntl_wifexited($st));
        echo "signaled:";var_dump(pcntl_wifsignaled($st));
}
}??>
```

```
andrey@vivaldi:~> php signals.php
653 Got SIGALRM at 20:08:45
653 After sleep at 20:08:45
653 Got SIGALRM at 20:08:47
653 After sleep at 20:08:47
653 Got SIGALRM at 20:08:49
653 After sleep at 20:08:49
653 Sleeping 5s at 20:08:49
658 Got SIGALRM at 20:08:51
658 Got SIGALRM at 20:08:53
653 Killing 658 at 20:08:54
658 Caught SIGTERM at 20:08:54
status  :int(658)
exited  :bool(false)
signaled:bool(true)
andrey@vivaldi:~> php signals.php
678 Got SIGALRM at 20:08:59
678 After sleep at 20:08:59
678 Caught SIGINT at 20:08:59
```

Using semaphores/shared memory in a OO way

- Procedural way is makes the code larger and more error prone.
- With OO semaphore, one has only the P and V operations visible to the programmer :
 - *acquire()*
 - *release()*.
- With OO shared memory, one uses only
 - *getVar()*
 - *putVar()*
- The parameters are passed during instantiation.

OO semaphores : example

```
<?php
require 'shm.php';
$sem = new Shm_Semaphore("test", 2); // semaphore's name is "test"
        // max_acquire is 2. All processes can use
        // the semaphore
$sem->acquire();
printf("Entered section at : %s\n", strftime("%D %T", time()));
sleep(5);
printf("Exited section at : %s\n", strftime("%D %T", time()));
$sem->release();
?>
```

```
andrey@poohie:~/test/phpconf> php sem1.php
Entered section at : 09/26/04 18:20:32
Exited section at : 09/26/04 18:20:37
```

```
andrey@poohie:~/test/phpconf> php sem1.php
Entered section at : 09/26/04 18:20:33
Exited section at : 09/26/04 18:20:38
```

OO shared memory : example

```
<?php
require 'shm.php';
//allocating 1024 bytes for the variable
$shm_var = new Shm_Var("acm3", 1024, "666");
if (!($str = $shm_var->getVar())) {
    $str = strftime("First script started at : %H:%M:%S\n", time());
    $shm_var->putVar($str);
}
echo "From memory : ".$str;
echo "Current time: ".strftime("%H:%M:%S\n", time());
?>
```

```
andrey@poohie:~/test/phpconf> php shm1.php
From memory : First script started at : 17:23:58
Current time: 17:23:58
```

```
andrey@poohie:~/test/phpconf> php shm1.php
From memory : First script started at : 17:23:58
Current time: 17:24:01
```

OO protected shared memory: example

```
<?php
require 'shm.php';

$guarded_var = new Shm_Protected_Var("acm4", 1024); //allocating 1024 bytes
$guarded_var->startSection();
$s = sprintf("putVal() at : %s\n",microtime());
echo $s;
sleep(5);
$guarded_var->setVal($s);
sleep(5);
echo $guarded_var->getVal();
$guarded_var->endSection();
?>
```

```
andrey@poohie:~/test/phpconf> php shm2.php
putVal() at : 0.99202900 1086906968
putVal() at : 0.99202900 1086906968
```

```
andrey@poohie:~/test/phpconf> php shm2.php
putVal() at : 0.99968900 1086906978
putVal() at : 0.99968900 1086906978
```

OO memory queue (1)

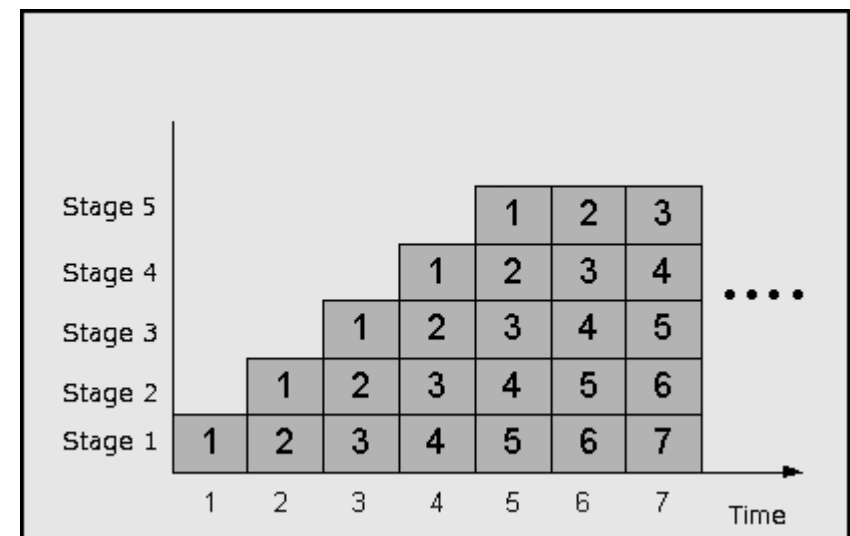
- Less details exposed to the programmer.
- Still quite powerful since all things that can be done in a procedural way are possible by using the OO memory queue.
- Less and more structured code which is easier to read and less error-prone.

OO memory queue - API

- `::__construct($shm_name, $size=16384, $perm='666')`
- `::setReceiveOptions($options)`
- `::setSendOptions($options)`
- `::send($message, $options = array())`
- `::receive($options = array())`
- `::getRecvMsg()`
- `::getRecvMsgType()`
- `::getErrorCode()`

Pipeline

- “In pipelining, a stream of "data items" is processed serially by an ordered set of threads. Each thread performs a specific operation on each item in sequence, passing the data on to the next thread in the pipeline. ” [PTProg]
- All processors, nowadays, use pipelining for executing instructions. The deeper the pipeline the better since more microinstructions are executed in parallel.



Pipeline - example

```
<?php
function pipeline_stagel() {
    $shm_var = new Shm_Var("pip1",512000,"666");

    $mq = new Shm_Message_Queue("pip1");
    $mq->setSendOptions(array('message_type'=> 12));
    $mq->setReceiveOptions(array
        ('desired_message_type' => 21));

    while ($i++ < ITERATIONS) {
        $ar =& generate_matrix();
        echo "ST1] Matrix generated\n";
        do {
            if (!$mq->receive()) die("ST1] Error\n");
        } while ($mq->getRecvMsg() != "2");
        $shm_var->putVar($ar);
        $mq->send("1");
        echo "ST1] Matrix sent. Iteration $i\n";
    }
    echo "ST1] Finished work!\n";
}

function print_matrix($ar) {
    for ($i=0 ;$i < DIMENSION; ++$i)
        echo implode(' ', $ar[$i]), "\n";
}

function &generate_matrix() {
    static $ar = NULL;
    if (!$ar) $ar = array_fill(0, DIMENSION,
        array_fill(0, DIMENSION, 0));
    for ($i = DIMENSION-1; $i >= 0; --$i)
        for ($j = DIMENSION-1; $j >= 0; --$j)
            $ar[$i][$j] = rand(0, 9);
    return $ar;
}
```

```
function pipeline_stage2() {
    $shm_var = new Shm_Var("pip1", 512000, "666");

    $mq = new Shm_Message_Queue("pip1");
    $mq->setSendOptions(array('message_type'=>21));
    $mq->setReceiveOptions(array
        ('desired_message_type' => 12));
    $mq->send("2");
    while ($i++ < ITERATIONS) {
        $mq->send("2");
        do {
            if (!$mq->receive()) die("ST2] Error\n");
        } while ($mq->getRecvMsg() != "1");
        $matrix = $shm_var->getVar();
        $mq->send("2");// let stage 1 work again
        $sum_matrix = sum_matrix_by_rows($matrix);
        echo "ST2] SUM=";
        var_dump(array_sum($sum_matrix));
        print_matrix($matrix);
        echo "ST2] iteration $i\n";
    }
    echo "ST2] Finished work!\n";
}

require 'shm.php';
define('DIMENSION', 3);define('ITERATIONS', 2);

$pid = pcntl_fork();
switch ($pid) {
    case 0 : pipeline_stage2();    exit;
    case -1 : die("fork() error\n"); break;
    default : pipeline_stagel();
        pcntl_wait($pid);break;
}??
```

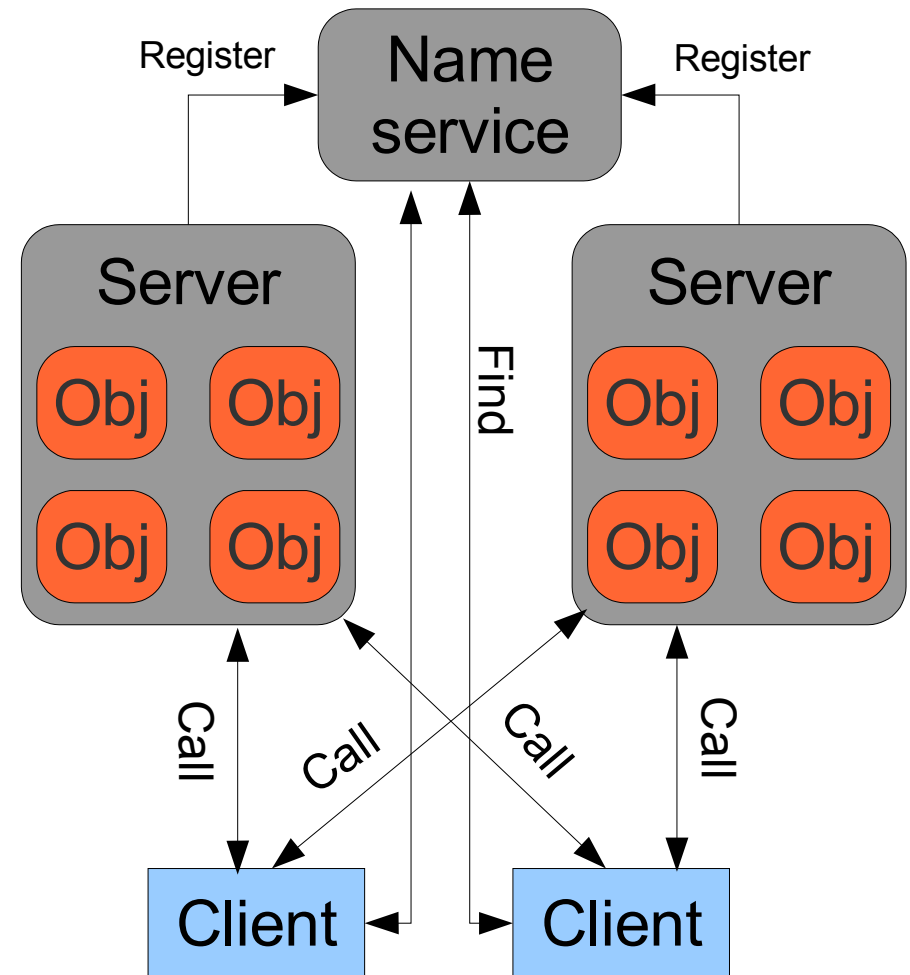
Pipeline – output

```
andrey@vivaldi:~/test/phpconf> php pipe.php
ST1] Matrix generated
ST1] Matrix sent. Iteration 1
ST1] Matrix generated
ST2] SUM=int(90)
1 7 4 1 0
3 1 4 1 5
0 2 0 7 9
5 9 8 2 6
7 3 4 0 1
ST2] iteration 1
ST1] Matrix sent. Iteration 2
ST1] Matrix generated
ST2] SUM=int(107)
8 0 4 6 5
2 1 5 9 1
7 0 5 7 1
3 1 4 9 7
3 8 7 4 0
ST2] iteration 2
ST1] Matrix sent. Iteration 3
ST1] Finished work!
ST2] SUM=int(81)
3 8 2 1 2
3 0 5 2 4
7 0 1 6 6
1 9 2 2 1
7 2 0 0 7
ST2] iteration 3
ST2] Finished work!
```

```
andrey@vivaldi:~/test/phpconf> php pipe.php
ST1] Matrix generated
ST1] Matrix sent. Iteration 1
ST1] Matrix generated
ST2] SUM=int(18)
4 1
6 7
ST2] iteration 1
ST1] Matrix sent. Iteration 2
ST1] Finished work!
ST2] SUM=int(18)
1 5
8 4
ST2] iteration 2
ST2] Finished work!
```

Message based RPC

- MSGRPC is an example of using memory queues.
- There are 3 parts in the system :
 - Client
 - Server
 - Name Service (also a server)
- Steps (client):
 - Create an object – factory
 - Implicit lookup is made
 - Use the object
- Steps (server):
 - Create the server
 - Add servants
 - Register them



Use case : XML example

```
<?php
require_once 'shm.php';
function rec($obj) {
    $ret_val = array();
    foreach ($obj as $k => $v) {
        if (($tmp = rec($v)) === NULL) {
            list(,$vv) = each($v);
            if (array_key_exists($k, $ret_val)) {
                $ret_val[$k][0] = $ret_val[$k];
            }
            if (is_array($ret_val[$k]))
                $ret_val[$k][] = $vv;
            else
                $ret_val[$k] = $vv;
        } else {
            if (array_key_exists($k, $ret_val)
                && !(is_array($ret_val[$k])
                && array_key_exists(0, $ret_val[$k]))) {
                $ret_val[$k] = array($ret_val[$k]);
            } // if
            if (is_array($ret_val[$k]))
                $ret_val[$k][] = $tmp;
            else
                $ret_val[$k] = $tmp;
        } // if
    } // foreach
}
```

```
        if (!count($ret_val)) return NULL;
        return $ret_val;
    } // rec

    $shm =
        new Shm_Protected_Var("cff2",16384,"666");
    $shm->startSection();
    if (!(($configuration = $shm->getVal())) {
        echo "Not cached\n";
        $xml = simplexml_load_file('server.cnf');

        $configuration = rec($xml);
        $shm->setVal($configuration);
    } else {
        echo "Cached\n";
    }
    $shm->endSection();

    var_dump($configuration);

?>
```

Use case : persistent data

- Good for cases when extracting data from different resources takes time or cache access time have to be low (no disk caching).
- Example : pre-parsed XML configuration file. The config file rarely changes -> put into shared memory on first script load.
- Get the configuration from the shared memory on every next request.
Note : deserialization needs some time.
- If you cache strings then better use ext/shmop since it does not serialize the content.

Use case : XML example : outcome

- Caching in shared memory with semaphore locking was about $\sim 4x$ faster than parsing on every execution.
- Pre-caching and no locking at later stage gives more $\sim 2x$ performance boost.
- Total speed-up measured was up to $10x$ – this saves minutes/hours of CPU time on busy machines.

Questions ?

I am reachable at
andrey.hristov_100days_de or **andrey_php_net**

This presentation is available at :
http://andrey.hristov.com/projects/php_stuff/pres/

SHM/SEM classes:
http://andrey.hristov.com/projects/php_stuff/shm.tar.gz

Books:
[PTProg] "Programming with POSIX threads", D. Butenhof, Addison-Wesley 1997