

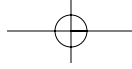


# 3

## Application Design: A Real-Life Example

*Prevent trouble before it arises.  
Put things in order before they exist.  
The giant pine tree  
grows from a tiny sprout.  
The journey of a thousand miles  
starts from beneath your feet.*

**A**PPPLICATION DESIGN IS A TOPIC SO BROAD that a whole book couldn't fully cover it. The term *application design* contains merely every single part of development, from data structure layout, flow charts, and entity-relationship diagrams to code layout, documentation, and anything in between. Because it is so important, however, we decided not to exclude it from this book, but instead to tackle a discussion of application design by restricting the topics covered to a “hands-on” example, namely phpChat. This chapter will give you an in-depth view of this real-time chat server application implemented in PHP, similar to an extended software case study. We hope that you can extract useful information and methods to use when designing your next application.



## 90 Chapter 3 Application Design: A Real-Life Example

Many of the boxed notes in this chapter contain remarks about techniques common to application design that you should memorize and try to use directly on the suggested example (or phpChat in general), and indirectly on your next project. *Note:* Another, more theoretical but shorter discussion about application design can be found in Chapter 7, “Cutting-Edge Applications.”

### Project Overview

When designing an application, you start with the idea of what the application is supposed to do. In the case of phpChat, the application is supposed to provide a browser-based chat service.

The chat should have the following features:

- **Real-time chat.** No deferred relaying of messages and no refreshes.
- **No client-side programming.** The browser should be confronted only with pure HTML (and eventually some JavaScript).
- **Networkable.** It should be possible to link chat boxes.
- **Generic.** Make as few assumptions about the target systems as possible and introduce as few requirements as possible.
- **No design enforcements.** Separation of code and page layout.
- **Easy to use and administer.**
- **Unlimited number of clients and chat rooms.**

Once you’ve gotten this far and know what your application is supposed to do, you have to evaluate the concept and create a more detailed overview of how the application should be laid out.

Take the time to write down all the requirements. It helps a lot, especially as a reminder later on.

When designing an application with a customer, this step is called *creating the specifications* (or just *specs*). At this point, the customer can still influence the layout of the application. This is very important because the application must meet the requirements listed in this step, or it won't be approved by the customer.

#### **The Customer Is Always Right, Even When He's Wrong**

Customers who contract you for an application often do not have enough expertise to design such an application by themselves, which is why they hire you. When discussing requirements with customers, guide them when they're suggesting bad solutions. For example, if the customer says, "I want a chat that displays full-screen images of every chatter, refreshed at least every second," you might make this counter-suggestion: "Wouldn't it be better to try to stick to thumbnail views next to each line? Most of your chatters won't have enough bandwidth to display full-screen pictures at all."

But be careful; never insist on your point of view (except when customers want you to implement unrealistic features). After all, customers pay you to implement *their* vision. To avoid losing a contract, you may have to accept temporarily implementing a bad solution (when you see that you can't talk the customer into doing it the right way), and then later on change it when the customer sees that it won't work out using their strategy.

For this project, you will take the role of the project manager and the authors will be your customers. Since we're nice customers, we won't keep insisting on nailing down further details of this application; we'll leave the rest of the design to you. Whenever this chapter hits a point where a choice or decision can be made, sit back and try to make your own choices. Closely evaluate all facts and then compare your results with the conclusions discussed in the book.

## **Comparing Technologies**

Before even starting to think about code layout, there's a phase we don't know what else to call but "getting things together." This is the intermediate step between the idea and the specs/code layout stage—figuring out the inner workings and on what to base them.

To make it clearer, let's go back to the very beginning:

- What do we want to create?
- How are we going to create it?
- Are there any existing implementations of our idea already?

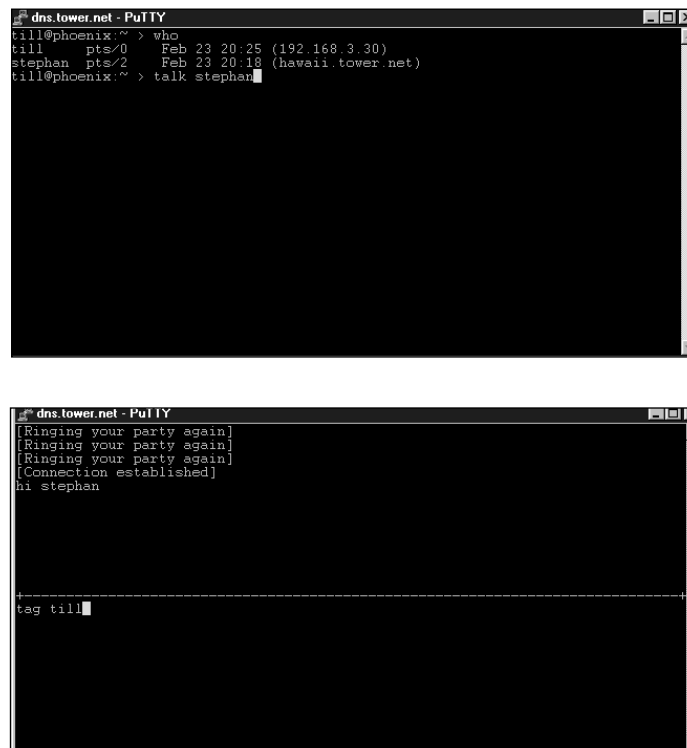
## 92 Chapter 3 Application Design: A Real-Life Example

- Do similar systems exist that perform almost the same task?
- If so, can we reuse anything from that design?
- Can we reuse foreign techniques, maybe add up to our system with them?

Questions over questions.

The first is easy to answer. We want to create a chat system. How? Well, with PHP, and somehow server-side—we don't know much more about it at this time.

Are there already any chat systems or something similar out there? Indeed there are. It starts right at your shell—the “talk” command allows you to chat with other people that you can reach via a valid network link (or local link), as shown in Figure 3.1.



```
dns.tower.net - PuTTY
till@phoenix:~ > who
till pts/0 Feb 23 20:25 (192.168.3.30)
stephan pts/2 Feb 23 20:18 (hawaii.tower.net)
till@phoenix:~ > talk stephan

dns.tower.net - PuTTY
[ringing your party again]
[ringing your party again]
[ringing your party again]
[Connection established]
hi stephan

tag till
```

Figure 3.1 The traditional “talk” command.

Of course, this isn't as powerful as we'd want it to be, but it's a start. Next, we could surf the Web to look for pages that have one of the (nowadays almost obligatory) chat links. Although they differ widely in look and feel/implementation, most of them can be boiled down to the following:

- Java for fancy interfaces, although some use plain HTML.
- A proprietary protocol with a single server (or simply database-backed).
- Few predefined rooms.
- Few predefined commands.

Apart from these chat setups, there are chat applications and networks such as Mirabilis' ICQ or the diverse Instant Messaging Systems—systems that don't always provide real-time services and generally require additional proprietary client software to be installed on every participating system.

However, one system stands out from the list. *IRC (Internet Relay Chat)* is a widely-known and long-used chat protocol used by many networks, some of which carry hundreds of thousands of users simultaneously. The IRC protocol is text-based—a drawback when operating under high load (long string commands generate much more traffic than single binary characters), but this also makes it significantly easier to process. Most current IRC servers support compressed backbone links, which greatly reduce traffic.

Although IRC requires special client software on every participating system, we can “tweak” this requirement to our advantage: Why not provide the client software ourselves server-side, and abstract it by using an HTML interface and allowing each user access to the network through an HTML client? This would give us control over what the user can do (each user is required to use our HTML client). Additionally, we have all the advantages of an existing network system: reliable client software, proven concept, hundreds of tools, etc. We could even allow users to use their own client software—an option to be avoided in most cases, however, as we want to create a “closed” chat network. On a closed network, you know every way that each client can access your network. By limiting the access points to specific setups, you greatly reduce the risk of being attacked.

This directly leads to the question, do we need a real protocol such as IRC? Or would it be sufficient to simply use a database-driven protocol, with a remote synchronization feature to provide the requested networking abilities?

Questions such as this will arise every time you plan an application, and they'll arise *often*. Make sure that you've got all of them covered, and make sure that no questions will arise at a later stage during development. *This* is the point where you can still address these questions; *later on* you might be unable to resolve them (and eventually get your project kicked into the trash). A good project is a project without doubts, without uncertainties, without inconsistencies, and without unforeseen eventualities. Make sure that after your planning phase you can assure a stable, fully evaluated situation!

**94 Chapter 3 Application Design: A Real-Life Example**

So let's get back to answering the question: Do we need an open (and perhaps too complex) protocol such as IRC, or should we stick with a conventional database approach? The simplest method to find an answer is also the most logical one—compare pros and cons and choose the option with the best results.

Implementing IRC as a protocol into the chat system will introduce a significant amount of complication because of protocol processing—processing network protocols requires nonlinear coding, something that isn't really supported by PHP. (To react to network messages, we need an event-based system.) On top of that problem, we'd need a way to handle message exchange efficiently; that is, dealing with messages *from* a user and *for* a user (which, unfortunately, may not always be handled in the same way). This problem exists in the database-backed solution, too, of course, but the database-backed solution doesn't require protocol handling. A lot of databases are supported natively by PHP, and those that aren't are most likely supported indirectly by ODBC. To gain the ability of networkable chat boxes, we'd only need to create a tool that can synchronize between chat boxes. (Unless you only want to run one central database server that's accessed by all boxes simultaneously.)

What would you choose?

Spoiler: phpChat is based on IRC, and this is why:

- Using a database, we'd introduce some kind of "proprietary," private protocol that wouldn't be able to interface to other standard systems. In times of interoperability and interconnectivity, this is a bad thing.
- An IRC library that functions well (namely phpIRC, see [www.phpwizard.net/phpIRC](http://www.phpwizard.net/phpIRC)) abstracts access to IRC networks into a set of easy-to-use API functions—and makes IRC handling equal to database handling in terms of code complexity.
- Existing IRC server software handles all the itchy-bitsy teenie-weenie problems of user management, reliable traffic forwarding, routing, etc., across networks. The software has been around for a long time and is proven to work, plus it's available for all types of systems.
- IRC is extremely scalable. If you run into load problems on server A at peak times or due to unforeseen events, simply fire up server B and dynamically establish a server connection into the existing chat (IRC allows you to do so, and is fully automated)—and you now have another server with enough free capacity for additional users.

## IRC Network Basics

Having chosen a communication standard for the chat, we should take a look at how exactly IRC networks are built.

Ideally, you should have evaluated the IRC network basics discussed in this section prior to choosing IRC—since it’s a bad thing to find out that IRC introduces a complicated structure after already having made the decision to use it. To this point, however, we’ve been working with “common knowledge” about using IRC networks, just for the sake of application planning. Now that we’ve led you to the “right” method to use for the application (IRC), this section provides the details you need to execute that plan.

IRC networks distinguish between clients and servers. Users can participate on the network only by using a special client software that establishes a client link to a server. All servers on the network are interconnected using special server links. Current implementations of IRC servers only support hierarchical structures, meaning that there must not be redundant ways to reach a server. This forms the net into a tree-like structure prone to network splits, but also greatly simplifies routing: All servers simply have to send all incoming data to all other links, without fearing to send redundant information to a server.

Each server can have a number of clients; the maximum number depends on the number of connections the server is willing to accept (of course, limits also exist in terms of network capacity and server load). As shown in Figure 3.2, each server can reach every other server across more or fewer server hops, so each server simply sends all incoming data to all outgoing links. For example, Server C and Server F might carry clients participating in the same channel (*channels* are IRC’s chat rooms—places where people can “meet” and “talk”). In this example, Server C would send the data via the only link it has: Server B. Server B then distributes the data to its other links, namely Server A and Server D. Server A doesn’t have any other links, so it won’t do anything, but Server D would pass on the data to Server E, and Server E in turn to Server F. Pretty easy to implement, but with one drawback: If Server A doesn’t have any clients connected to it participating in the channels to which Server C sends data, all data for Server A targeted at this channel would simply waste bandwidth.

### RFC for IRC

Similar to all open standards on the Internet, the basics of the IRC protocol have been specified in an RFC (Request For Comments). The RFC for IRC is RFC 1459, which can be retrieved, for example, at [www.irchelp.org](http://www.irchelp.org), a site that carries a lot of information on IRC.

## 96 Chapter 3 Application Design: A Real-Life Example

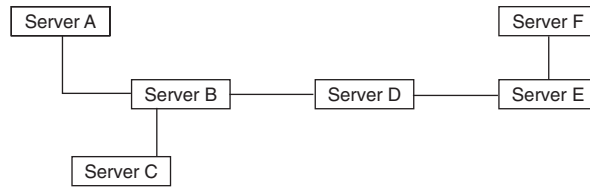


Figure 3.2 A sample IRC network structure.

This is one of the main problems of this “limited by design” network: All public traffic has to go to all servers. But will this problem really arise under the conditions in which we intend to implement our IRC network? Surely not, as the number of clients we intend to handle will never be so large as to be harmful, given a standard server-hosting situation. In internal networks, this problem shouldn’t arise at all.

To reduce the total number of critical links, the network can be laid out to follow its physical topology. If one server is connected with higher capacity than the rest, for example, it can take more leaf nodes than others (connecting lots of leaf nodes to a server with a small backbone wouldn’t even make sense). Another option is to set up the routing to fit the network. For example, U.S. servers are homed in the States, German servers are homed in Germany, and so on. Frankfurt has an overseas link to New York; thus, the IRC server in Frankfurt should link to New York’s server (following the network’s physical layout). It could also be done in another way: Frankfurt could link to, say, Poland. But if Poland doesn’t have its own overseas link, the traffic routed from Frankfurt to Poland would need to find some other way to cross the ocean—it would be routed to some other country (or even two or three countries) until it finds a free overseas link. This additional routing wastes a lot of bandwidth; thus, attempts are being made to adapt the IRC network structure to best fit the underlying physical network structure.

These design problems are present only in the biggest networks, carrying tens of thousands of users. These networks really need to find reliable links for their backbones. Typical Web-based chat rooms or networks are unlikely to carry more than 1,000 clients at once, so you shouldn’t run into serious problems at first. To avoid complications, however, it’s a good idea to plan around these sorts of problems that may arise eventually.

From a server’s point of view, the network looks like Figure 3.3.



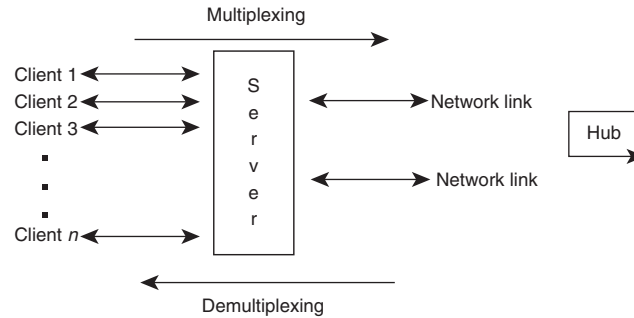


Figure 3.3 Network structure from a server's point of view.

The structure implemented here is similar to a mixture of a multiplexor, demultiplexor, and a hub. In the direction client to network, the server compresses all data from the clients and sends it to the network links. In the other direction, it determines which information from the network is important for which client and sends it to the appropriate link. All incoming data from the network that has to be passed on to the other network links is sent on directly.

Basically, this is the setup we'd need for our own chat system. Now take a minute and try to imagine how we can achieve our goal. We need a working server environment that fits the following description:

- Accepts IRC network links
- Accepts IRC client links
- Provides a Web-based user interface
- Is as easy to implement as possible

## Fitting the Application into the Network

If you came up with a plan to develop your own server in PHP (or something similar), rethink a bit. You might have gotten a bit confused with the idea that implementing a chat server means implementing a network server. This is indeed something we wanted to lead you to, but don't want you to *do*, as this is simply unnecessary—there's already a well-written server software available for all systems. So how about using one of the existing servers and representing our server to the network as a client? The only thing we'd have to do is to add another layer of abstraction to the network, as shown in Figure 3.4.

## 98 Chapter 3 Application Design: A Real-Life Example

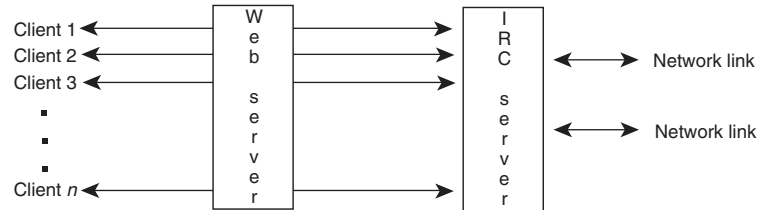


Figure 3.4 phpChat as an abstraction layer to the server.

The Web server will run the PHP chat server. For each client connection it accepts, it will create a client connection to the IRC server. This way, we can make sure that all data we get for this client is meant only for this client—and nobody else. Each chat process will carry a single user, and doesn't have to worry about other users. User coordination, traffic control, and so on can be done by the IRC server, for which we'll simply take one of the freely available servers.

This technique also has the advantage that this chat server application can be used as a safe gateway to IRC networks (see Figure 3.5). A lot of corporate and private networks are behind firewalls that filter IRC ports. Since this chat is only communicating via HTTP to its clients (which is not filtered), only the chat server itself needs an open connection to an IRC server.

Therefore, the only thing we're going to do is to implement the client software that would otherwise be required on the user's side on our Web server. IRC knows all the commands that are required to set up a powerful chat, and the networking issues can all be solved by using standard "off the shelf" server software that's already available. Thus, if our interface supports all features of IRC in a convenient way, we're done with our work.

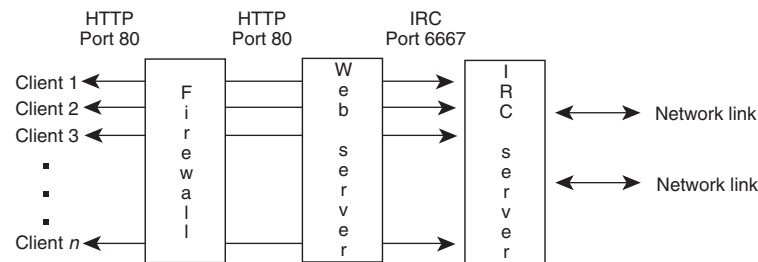


Figure 3.5 phpChat as a safe IRC gateway.

## Interfacing the Network

As we mentioned earlier, IRC requires some processing overhead. Hacking a complete protocol handler for interfacing with IRC is a bit of a complex task, but we favored IRC instead of the database-backed solution because an API already exists that does this work for us.

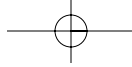
Know the market! It's essential for every programming project to know which parts have already been done by other people and which still need to be done. Never reinvent the wheel! Especially for commercial projects, it can pay off tremendously to buy foreign bulletproof solutions for specific tasks, rather than design and develop one yourself. The latter is sometimes more expensive and much more time-consuming. On top of that, external solutions are usually constantly being improved—a process that's totally independent of the progress of your own project. By receiving an upgrade from an external company, you simply replace a part of your application with a newer version. This way, you can upgrade certain parts of your application without having to put your own work into the changes. Plus, when using existing libraries, you automatically agree to build your project on common, standardized APIs, which is always a great benefit.

On the other hand, binding yourself to foreign products can prove to be a negative decision if the producer fails to improve the product or keep it up to date, as well as if bugs in it aren't corrected.

In our experience, Open Source products have been the most successful external parts to be integrated. Open Source products are being improved and extended extremely rapidly and are usually oriented at common and open high-potential standards.

### Exercise for the Reader

Search for applications/libraries written in PHP that make use of IRC and compare them in terms of design, flexibility, and ease of use. Of course, the implementation is also interesting (but shouldn't be your main focus). The design is always the most crucial part of development; after the design is finished, the actual implementation is usually straightforward and easy to do (even though a lot of programmers think differently).



## 100 Chapter 3 Application Design: A Real-Life Example

The library we've chosen for this project is phpIRC ([www.phpwizard.net/phpIRC](http://www.phpwizard.net/phpIRC)), for these reasons:

- It's easy to use.
- It's a powerful, complete API.
- It uses event-based processing.

The use of event-based processing is particularly interesting here. This is a technique usually implemented in traditional applications; for example, all Windows programs are event-based. *Event-based programs* run in an endless loop, waiting for something (an event) to happen. Events can include user input, mouse movements, network events (incoming packets), etc. As soon as an event is signaled, the program breaks out of its main loop and searches for a procedure that handles this event. All procedures that want to handle the event that just occurred are called with the specific parameters of the event (for example, packet data of incoming network traffic).

Concretely, using “traditional” programming, an incoming ping would be handled as shown in Listing 3.1:

Listing 3.1 **Pseudocode for handling a ping.**

```
again:

wait_for_network_data();

if(incoming_data == ping)
{
    send_pong();
    update_traffic_counter();
}

goto again;
```

This code waits until it receives data from the network, then tries to find out whether the data was a ping. If so, the code sends a pong back and updates a traffic counter for statistical reasons. After that, it just jumps back to where it began. Imagine this with hundreds of events, some of which might depend on others, some not, some only under certain circumstances...A pain!

However, event-based programming makes it significantly easier, as shown in Listing 3.2:

Listing 3.2 **Event-based pseudocode for handling a ping.**

```
event_handler ping()
{

    send_pong();
```

```
}  
  
event_handler incoming_data()  
{  
  
    update_traffic_counter();  
  
    case of ping: handle_event(ping);  
}  
  
while(not_done())  
{  
  
    wait_for_event();  
  
    case of network_data: handle_event(incoming_data);  
}
```

The code looks bigger, but also much clearer. The main loop waits for an event to happen. If it finds that an event happened and that it was triggered due to incoming network data, it dispatches this event using the central procedure `handle_event()`. This function then determines a handler for the event and calls it. The handler in turn updates the traffic counter and launches another event if the first event was a ping. After dispatching the event using `handle_event()` again, a pong is sent.

Alternatively, both `ping()` and `incoming_data()` could register themselves to the event "incoming\_data". However, creating two different events gives a greater variety of events and thus allows for much more detailed, target-oriented processing.

It's a bit strange at first getting used to event-based processing of information (it works similarly to a finite-state machine), but it has many advantages:

- A modular structure is forced on the application. Each module works independently of the other modules and can easily be changed, exchanged, or extended.
- Any part of the program can trigger any kind of event and thus enforce any type of reaction in the application (in other words, you can control any part of your code from any other part of your code).

## 102 Chapter 3 Application Design: A Real-Life Example

- From one central point of the program, all data can be dispatched to all recipients transparently. You don't have to worry about manually copying and transforming structures; each event handler takes care of receiving its data on its own.
- New code can be plugged into the application extremely easily, just by creating a procedure that registers itself to the appropriate event.

Thus, once the main event-dispatcher framework is created, the whole application can be created by writing handlers, handlers, and more handlers.

Get familiar with the techniques used to implement finite-state machines. These are elemental in programming and information processing in general.

Luckily, the event-dispatcher framework is already contained in phpIRC, so we won't need to do that programming for this project.

### Interface Structure

phpIRC forms the IRC client part of the application and is responsible for all network communication. This means that it also needs to be in control all the time to be able to react to network messages in a timely manner. If phpIRC's message-processing functions were activated only occasionally, safe, secure, and speedy communication couldn't be guaranteed. For this reason, phpIRC forces a special program layout, as shown in Figure 3.6.

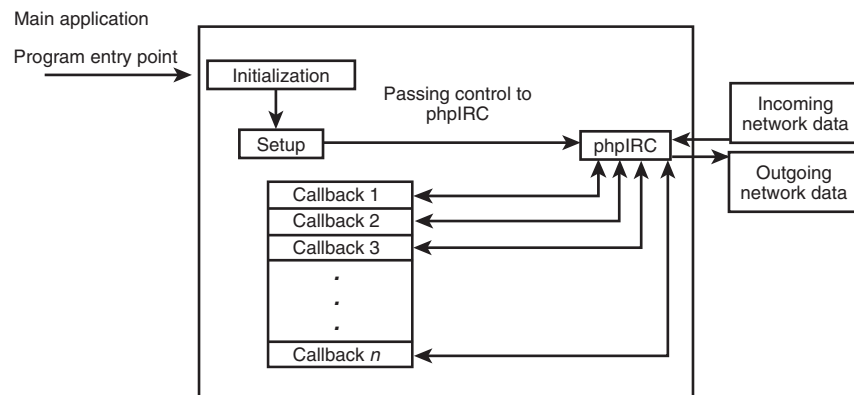


Figure 3.6 phpIRC's forced application layout.

After doing initialization and setup, the application has to surrender control to phpIRC. phpIRC then enters its main event loop and waits for something to happen. During setup, the application has to register callbacks for each event it wants to process (for example, incoming private messages, incoming server messages, and so on). These callbacks are the only possibility for the application to regain control. phpIRC then dispatches all events to all functions that have registered themselves with the library. These functions can in turn enter another idle loop in phpIRC to wait for another event to happen, or they can use phpIRC's API to perform certain actions on the network (send private messages, join/leave channels, and so on).

This very basic layout already allows for downstream communication, which means that phpIRC is able to receive messages from other users. People could actually “talk” to your script.

*Note:* *Downstream* means from the network to the user. *Upstream* is the opposite, from the user to the network.

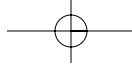
**Exercise for the Reader**

Structure a downstream interface that makes use of phpIRC's features. Implement it on paper to become familiar with phpIRC's API. Then build a simple downstream interface that logs onto IRC and displays all messages from a specified channel.

**Downstream Communication**

Since chatting is a real-time task, meaning that it happens as you do it and causes instant replies, we don't want to introduce latency into the interface. *Latency* describes the reaction time of the interface; for example, the time from the point when the reader presses the Enter key to submit a message until it shows up in the chat window. Even though a latency of less than a second might objectively be a very short wait, it *seems* extremely long and annoying to the user. Ergo, incoming messages must be displayed at once (or at least as soon as possible). HTTP is a stateless protocol, however, and doesn't allow instant updates of pages without reloading a complete document. Of course, there are multipart documents and automatic refreshes, but these options introduce a very nasty flicker each time the page loads again, require database buffering for output, and introduce lag because of constant reconnects and data transfer from the Web server.

One solution is “streaming HTML,” something that's not officially supported anywhere, but works nevertheless: The script that does the interface output simply idles in an endless loop and doesn't terminate the HTML page the browser is receiving. When something has to be sent to the user, it's printed and immediately flushed from the server's buffers. This way, the browser is constantly rendering and always displays the most up-to-date data. One problem persists in this approach, however; no complex HTML entities can be rendered on the fly. For example, you can't output the rows of a table one by one, because the browser requires all rows and



## 104 Chapter 3 Application Design: A Real-Life Example

columns of the table to be present completely to determine the final size of the table. As long as you restrict yourself to outputting text lines one after another, and only use tables when you can print them all at once, everything works fine.

Quirks such as streaming HTML are common tricks that you should know. Always keep yourself informed about such things.

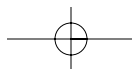
Streaming HTML also has one implication that some see as drawback and some as advantage: Since the client connection stays open, there must always be one server process handling it. This means that every client requires at least one Web server process to be running only for that client. The advantage is that no overhead “per hit” occurs. Usually, when the client requests a document, a new process has to be spawned; the script generating that document has to be loaded, parsed, and executed; and finally, the data has to be sent. Since the server process now remains in memory, however, spawning, script loading, and interpretation only have to be done once per client. On sites that would otherwise have hundreds of hits per second, this might be a definite advantage. However, each process now stays resident in memory and demands RAM for itself—on Intel x86 systems equipped with Linux, Apache, and PHP 4.0, such processes tend to be as big as 2MB each. Consequently, a small server with minimal RAM on board would soon start to run from swap—and that means death.

*Note:* *Swap memory* is virtual memory that’s meant to extend the RAM—the physical amount of memory on a computer. Swap memory is stored on a hard disk, which is extremely slow. When physical memory is all used up, modern operating systems start allocating new memory in the slow swap memory. If a chat server gets hit by a lot of clients at once, which eat up all physical memory and start running in swap memory, the operating system will constantly have to exchange parts of the RAM with parts of the swap memory (since programs can’t be executed from swap memory), and this starts a “cycle of death”: The operating system notices that a process in swap needs to be run and loads it into RAM, but has to put another running process from RAM into swap. It executes the process in RAM but finds the old process (now residing in swap) has to be run, so it swaps it back into RAM, and so on and so on. You can quickly kill a server this way, forcing it to be reset or taken off the net. By the way, this is also a common “denial of service” attack, a bit similar to the ones that Yahoo! and others were exposed to earlier this year.

Would you have thought about the implications of resident processes? If not, make sure you do next time! Keep evaluating every situation fully.

### Upstream Communication

Upstream communication—that is, accepting user input and sending it to the network—is the next stage to consider.





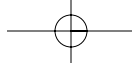
Here's the hard part: We can't send data to the IRC network from just any process. Why not? Because IRC is a state-sensitive protocol, communication is bound to a specific client connection. PHP doesn't allow taking over foreign sockets from other processes; thus, the main process that also handles downstream communication (the process that acts as IRC client) runs isolated from all other processes. The question now is how we can open a door to pass data into the main client.

How would you implement upstream communication? Make at least a theoretical approach. Draw the dataflow on paper. If you haven't done so already, write down at least three possibilities for runtime data exchange.

The downstream process must keep running and may not be terminated. We can't simply reinvoke it using a POST or a GET for passing data, since that would mean launching another process, with the need to re-login, re-setup, etc. Using such an approach would result in constant login/quit sequences that would be extremely disturbing in a chat. And it would result in data loss, since during the time between a logout and a login, lots of messages could be transmitted (which would be invisible to the newly logging-in client).

The chat could be based on a single bot that stayed online all the time and recorded all messages for all users into a database. The user interface would then only need to extract all meaningful data from the database. However, two problems stand against this possibility: a) The chat would be mainly database-backed (something we wanted to avoid); and b) It wouldn't make the clients visible to other IRC clients, as the bot would be the only "real" client on the network. This would make usage of the IRC network ridiculous.

Thus, we need at least two independent processes: one that handles the IRC communication and can't be interrupted, and another to accept incoming messages from the user. Some sort of "container" must then be used to interface between the two processes. Figure 3.7 illustrates this problem.



## 106 Chapter 3 Application Design: A Real-Life Example

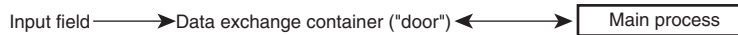


Figure 3.7 Upstream communication.

The situation can be compared to a car race. The driver racing on the track is the “main client” and the racing team in the pit is the user input field. The driver is bound to the race he’s in; he can’t just leave the track and stop to see what’s going on. Whenever the racing team flags him in for a pit stop, they “interface” to him—giving him a signal to make a break after the next lap.

What’s being done is (leaving radio communication aside) to signal every time the driver passes the finish line. This signal works as the “interface” to the driver. Basically this is what we need to do, too—signal to our main process. Since the main process is event-based, we frequently get the chance to take control over the application and do what we want to do. This means that we can install a handler that “looks” frequently for a signal from the outside. The method to periodically stop and check for incoming data is called *polling* and will be the preferred method for phpChat. phpIRC features idle callbacks, which get invoked every time phpIRC has nothing to do and simply waits for something to happen on the network. Tagging a handler to this event enables us to watch out for a signal. Now, how are we going to signal something? This is actually pretty easy, using one of the following methods:

- Set a flag in a database.
- Create a lockfile in the file system.
- Use semaphores.
- Set a flag in shared memory.

These are basically the methods that we have with PHP to “leave a message.” The following sections describe each method.

Pipes can’t be used for interprocess communication here, because a pipe requires two processes to be running at the same time. Our situation requires interfacing one constantly running process from other, short-term processes.

*Note:* Of course, more exotic methods are available, such as sending emails between processes. We’ve seen people doing this, but we won’t go into that option here, as the disadvantages should be clear to the reader.

### Setting a Flag in a Database

Setting a flag in a database is probably the *de facto* standard method for PHP users: Connect to a database, leave some data in it, let it be processed further by other processes. This method is extremely easy to implement and is available on all systems, but has a disadvantage. Can you tell what the disadvantage is?

The disadvantage doesn't come from the database *stuffer* (the process that inserts user messages) but rather from the database *reader* (the main process that retrieves all user messages from the database). To achieve a good "chat feeling," we need as little latency as possible—and thus a very good response time. The response time is crucial for Web-based chat, as this is how the user will actually feel "integrated" into the action. When the messages come slower and slower, users quickly become frustrated and quit. Our testing showed that a latency of more than a second is too much. To stay below this value, the poll frequency in which the main process has to read messages from the database must be very short; the default value in phpChat is 0.5 seconds (two checks within a second). Now, as soon as a lot of clients have to be handled by the chat system, the database gets quite busy and takes up more and more resources. At about 40–50 queries per second, our test server spent about one third of its processing time simply executing database queries. Even if this was a disqualifying benchmark for the database system (it should have been able to process many more queries), some optimization is obviously necessary, and this isn't the ideal setup.

### Creating Lockfiles

Our next idea was that, if the database took up too many resources when handling interprocess communication, a file system might be more efficient.

But the file system clearly lost the race. Again, the stuffer wasn't the problem—creation of the lockfiles worked smoothly. To detect whether a lock was set, however, lots of calls to `clearstatcache()` had to be done in order to correctly determine whether a lockfile had been deleted or was still present. `clearstatcache()` had such a hard impact on the system performance that we didn't try to look further into this option; the chat only worked at a quarter of the performance it reached using the database-backed approach.

Create your own benchmarks. Make test scripts accessing the database and the file system at high frequency. Write down your results and compare them. This is always a good idea when evaluating data-exchange methods—never trust theoretical descriptions of what the systems *can* be capable of! In practice, most things will look different.

### Using Semaphores

Of course, the reasons for the poor performance of the former approaches are easily recognized.

## 108 Chapter 3 Application Design: A Real-Life Example

What are the reasons? Try to find and write them down. Try to find the critical points—this is crucial when having to optimize later on. “A chain is only as strong as its weakest link,” and software is only as fast as its slowest inner loop. The process of finding these bottlenecks is called *profiling* and is extremely important.

When using a database, the bottleneck is the database: the time required to invoke the database, let it execute the (relatively small) query, retrieve the result, and determine what to do next (called the *overhead*) is pretty long compared to the result we’re getting. In other words, we’re using a huge software system designed for complex data storage to exchange simple, Boolean values—if there’s something a database was *not* designed for, it’s this. No wonder it didn’t perform optimally; the bottleneck is the overhead, the time required for setup and deinitialization.

The file system performed badly because it was not designed for this usage, and because of other limitations: PHP doesn’t include optimal file-system access methods. Determining the existence of a file requires constant cache invalidations and recaching—again, large overhead for a trivial task.

So why not use something completely different? We’re surely not the first people having to deal with interprocess communication; others must have come up with good solutions for this already. And so we reach the next possibility: semaphores.

*Semaphores* do exactly what we want to do: They work as signals. Semaphores are counters stored in shared memory. You can “acquire” a semaphore and thus increase its counter, and you can “release” a semaphore, decreasing its counter. Additionally, there’s the possibility of waiting for a semaphore to become free, meaning that its counter falls back to zero. This option has one drawback, however: Semaphores were meant to lock resources, to create some kind of scheduling mechanism allowing many processes to wait for available time on a device, or something similar. Whenever you’re waiting for a semaphore to become free, the process that’s waiting is put to sleep and cannot perform other tasks. If the main process was waiting for the user-input field to signal a new message, it would sleep and couldn’t process the incoming network traffic.

No reason to give up yet; people have come up with still other solutions.

### Setting Flags in Shared Memory

*Shared memory* is similar to semaphores, but a bit more versatile; shared memory is memory that’s available to every process in a system. Multitasking systems are usually designed in such a way that each process is running completely isolated from other processes for security reasons. Different processes can share data by setting up and connecting to special memory blocks, namely *shared memory blocks*. These blocks can then contain variables (or any other kind of data, but PHP only supports storage of concrete variables).

This is exactly what we want: the ability to store a Boolean value in a place in memory where every process can look at it. Since shared memory works (as the name suggests) only in RAM, it's extremely fast and requires almost no overhead. With this option, every chat process looks for its own variable in shared memory and only issues a query to the database whenever it finds that variable set by the user-input field.

Why is the data exchange still based on a database at the very end? Try to find some answers.

The database is still being used for one main reason. Shared memory is not supported by default in PHP; you need to specifically compile support for it into PHP. However, many people with access to a PHP-enabled server don't have the option of recompiling PHP because they only rented space on the server, because they don't have sufficient rights, or maybe because others depend on a certain setup of PHP. Leaving the database in as the final data-exchange path makes use of shared memory as an optional optimization. People who can't use it can simply disable it and still have a fully working version of the chat server—operating at suboptimal performance, but operating.

When creating an application designed for widespread distribution, keep in mind that not everyone will have the same setup as you—and probably not the possibility of re-creating your very special setup. Even though PHP is 99% system-independent, some things do depend on the system. Carefully calculate whether enforcing certain circumstances is worth a potentially huge loss of customers.

### Interface to the User

Now that we moved all the tricky parts with the data exchange out of the way, the actual HTML interface to the user is trivial. We know how to accept input from the user and how to deal with network communication. The last “problem” is packaging the generated output for the user in a convenient way. HTML offers only one way to have different windows act independently in one browser view: framesets. The interface typically consists of the user-input field; the chat output field; a *nickname list* (or just *nick list*), which shows other participating clients in the same room; and an action panel to allow one-click control over the chat for actions such as nickname changes, joins, parts, quits, and so on. These activities can all be handled by single processes whose output will be integrated into a frameset.

## 110 Chapter 3 Application Design: A Real-Life Example

The main process, also responsible for the chat-output streaming, will keep state information updated in a database that all other interface components can access, retrieve, and display in a suitable fashion (see Figure 3.8).

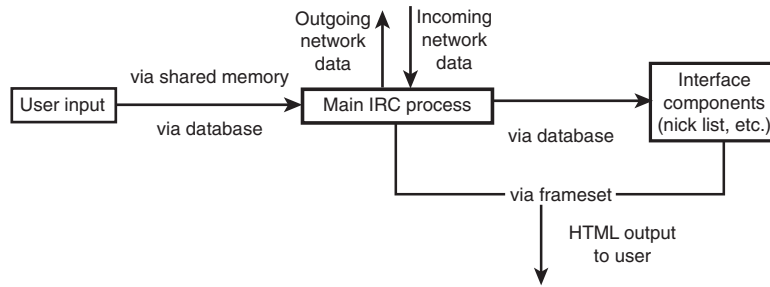


Figure 3.8 The final application layout.

### Interface to the Developer

An interface for developers? What does this have to do with chat? And how is it supposed to work? Typically, most applications suffer from the disability of being “solid,” meaning being either completely unmodifiable or difficult to modify by foreign developers. In terms of end-user-oriented software (for example, desktop environments such as Windows, KDE, MacOS, etc.), hardly anyone will ever find the ideal solution. Similar to a chat system, most people who download it say, “Hey, great, but it lacks this and that,” or “Cool, but I don’t like the way it does *xyz*.”

Without an easy, clearly exposed path for modification by anyone using it, most applications end up in the trash. Most people won’t even try to work on a program they didn’t develop themselves if the ease of doing so doesn’t hit them right in the face.

This means that for the chat application to consistently enforce independence of code and interface layout (allowing an interface to HTML developers) and to consistently enforce independence of data-processing steps, we need to create a solid *application core* (the part of the application that nobody should ever need to change) which interfaces to a distributed set of *plug-ins* (the part of the application that most people will want to change somehow).

Think again about the importance of these enforcements. Would you like an application to be designed like this? Would you even need it? Think about how this could be realized.

## Interface to HTML Developers

In terms of the HTML interface, abstraction of code and layout is done using templates. This is the easiest possibility for tweaking an application to your needs, yet it's also the most powerful. Within seconds, you can change the look and feel—without having to modify a single line of code. Everyone with basic HTML knowledge could completely restructure the way the application would show itself to a user. As this method is discussed elsewhere in this book, we won't go deeper into it here. To find more details about using templates, please read Chapter 5, "Basic Web Application Strategies."

## Interface to Code Developers

Providing an interface to other developers is usually associated with the term *API* (*Application Programming Interface*). APIs are normally provided by libraries (such as phpIRC), but not by complete applications. But applications that have the capacity to be extended by a programmer are much more successful than applications that must be used "as is." Of course, in terms of PHP applications, anyone can modify the source code, but many people refrain from analyzing a complex system and applying modifications to it. Thus, the application itself needs to expose certain ways of being extended.

*Note:* We're differentiating here between *applications* and *libraries*. Libraries are meant to be used by applications, cannot be run stand-alone, and are generally much easier to extend than applications. Applications consist of a full, closed system.

Try to find out how common applications can be extended. For example, for your favorite text-processing tool, see whether the developers provided the capacity to extend the tool's functionality.

Two primary possibilities of extending applications have evolved: Either the application provides scripting capabilities (similar to macros), or the application is able to use plug-ins. As for PHP, implementing a script language in a time-critical part of a system...we don't need to think any further. On top of that, the complexity of creating a full-fledged parser is way too much to ask. But plug-ins are much easier to implement and have many advantages. A *plug-in* is a little piece of code that can register itself with the application and catch certain events from it, get access to internal data, and so forth. While integrating seamlessly with the main system, plug-ins still remain isolated files that can be detached and spread separately. They can be attached to the system without having to modify a line of code, which allows a system administrator without any knowledge of PHP to extend the application by using foreign code. Concretely, this is realized as shown in Figure 3.9.

## 112 Chapter 3 Application Design: A Real-Life Example

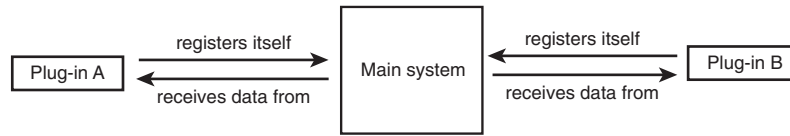


Figure 3.9 Chat system with plug-ins.

Design your own plug-in-system, at least theoretically. Create a minimal application that's able to register plug-ins with itself and execute them.

When starting up, phpChat includes an include file, which in turn includes all wanted plug-ins. Listing 3.3 shows how this include file works:

Listing 3.3 The plug-in includer.

```

////////////////////////////////////
//
// Plug-in Integrator
//
////////////////////////////////////

include("chat_plugin_out_htmlspecialchars.php3");

include("chat_plugin_out_link_transform.php3");

include("chat_plugin_out_colorcodes.php3");

include("chat_plugin_clock.php3");

include("chat_plugin_cmd_basic.php3");
include("chat_plugin_out_basic.php3");

////////////////////////////////////

```

Each of the plug-ins is built up in the same way, consisting of a main part and an event part. The main part calls two functions in phpChat, with the following names: `chat_register_plugin_init()` and `chat_register_plugin_deinit()`. Each function takes as a parameter the name of another function, which should be called for plug-in initialization and plug-in deinitialization, respectively.

phpChat adds these function names to an internal table. Upon initialization of the chat, as soon as phpChat is fully set up, it makes a run through the initialization table and calls the initialization function of every plug-in that registered itself. Similarly, upon shutdown, it runs through the deinitialization table. This method allows signaling the plug-ins to activate and deactivate themselves.



To be useful in the application, phpChat offers a set of events to which each plug-in can attach itself. During plug-in initialization, each plug-in tells phpChat to send a set of desired events. Events might include the chat being idle, the user submitting a new message, the user clicking on a nickname in the nick list, an incoming message from the network, and so on.

At runtime, the plug-ins can intercept these events and perform certain tasks. The clock plug-in, for example, registers itself to the “idle” event and checks the current system time frequently. After a predefined number of minutes, it announces the time to the user.

For most events, phpChat also sends parameters (such as the message texts for incoming messages), which the plug-ins are allowed to change. For example, the list of plug-ins in Listing 3.3 includes plug-ins named `htmlspecialchars` and `link_transform`. These plug-ins change the output of messages; `htmlspecialchars` applies a call to `htmlspecialchars()` to all printed text (for security reasons, so that no one can insert malicious HTML code into the chat), and the link transformer detects all URLs and email addresses and prefixes them with `<a href=" "></a>` or `mailto:`, respectively, so that users can click links right in the chat window (see Figure 3.10).

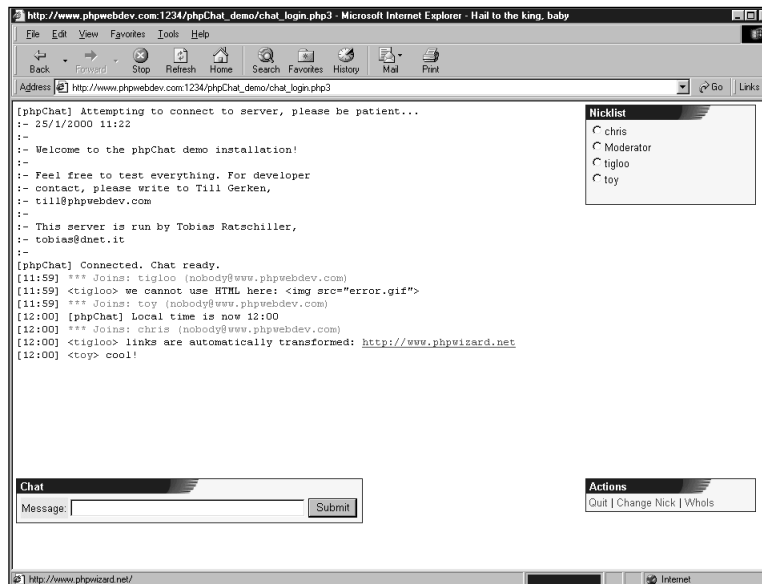


Figure 3.10 The plug-ins at work.

## 114 Chapter 3 Application Design: A Real-Life Example

As you can see, plug-ins offer an extremely powerful way of extending a complex system. Consequently, phpChat has abstracted most of its own internals into plug-ins as well. The complete command interpreter has been moved into a plug-in, as well as the complete set of text formatting/printing procedures. This means that there is only a solid kernel that doesn't have to be changed because there's simply nothing in there that would require changing—the rest can be freely modified, extended, even removed, without any impact on system performance or operability. Have you ever seen an application that doesn't complain about someone deleting its files? Using this technique, an application won't complain—and will even dynamically adapt to it.

Plug-ins can be used in many ways, not just for chat programs. For example, you could also build a portal site consisting of the traditional news page, an email interface, etc. Using plug-ins, you can design a “site kernel” that handles all basic issues such as providing page layout, database back end, sessioning, and so on. Based on the site kernel, you can then create plug-ins for displaying news, sending and receiving email, even for providing different methods of logging in. Even if it's quite an effort, we encourage you to create a plug-in-based application as an exercise. It will be worth the work.

Listing 3.4 shows a plug-in template implementing a “dummy” plug-in as code base for new plug-ins.

Listing 3.4 **A plug-in template.**

```
<?
//
// Use these variables to tell the plug-in installer how you named your
// initialization and deinitialization functions. This is done to eliminate
// the need for changing the installer code, which would ask for errors.
//
$plugin_init_function = "myplugin_init";
$plugin_deinit_function = "myplugin_deinit";
//
//
//
//
// myplugin_idle_callback(int code, mixed parameter) - example callback
//
//
```

```

// This is an example for a callback function. See below on how to register
// and remove it from the call chain.
//
// $code specifies the reason for invocation, $parameter contains all callback
// information.
//
// The return value should always consist of a modified or unmodified version
// of the input parameter $parameter. The return value is used as input
// parameter for the next callback. This allows for multi-stage message
// processing and such.
//
//
//
function myplugin_idle_callback($code, $parameter)
{
    return($parameter);
}

//
// myplugin_init() - initializes this plug-in
//
//
// Put all your initialization code in here. This code will be called as soon
// as the main bot is all set up with connecting and callback installation;
// thus, you can rely on a safe environment.
//
// Although the return value is currently not used, "0" should indicate
// initialization failure and "1" initialization success. This might be used
// later on to enable plug-ins to stop the current chat session right after
// login.
//
//
// Return value:
// 0 - error
// 1 - success
//
function myplugin_init()
{
    // register callbacks here
    chat_register_callback(CHATCB_IDLE, "myplugin_idle_callback");

    return(1);
}

```

*continues*



The main code registers the initialization and deinitialization routines for this plug-in. The plug-in initializer then installs the callbacks this plug-in wants to intercept, and the deinitializer removes them.

## Administration and Security

No system is a good system if it can't be administered. The days when "Netiquette" made it a point of honor to be polite and integrate oneself into the community are long gone. Nowadays it's common to be exposed to hacks, harassment, and other forms of attacks—and unfortunately, most of them don't stay at the verbal level. There's hardly anything to say against digging for security leaks and other holes in an application or network system. Constantly exploiting them, however, is worthy of condemnation, yet a lot of people consider it "fun." This demands an external interface, running independently of the main system, which allows full control over all of the application's data and users. In terms of a chat system, this means that we need to be able to kick users, moderate their messages, and secure chat rooms.

*Note:* Not all features listed here are implemented into the code on the CD-ROM. The basic administration system is complete and fully functional, but we'd like you to exercise and extend the code base with the features you feel appropriate. If you haven't made significant extensions to larger applications, we honestly urge you to gain the experience now.

The question for our chat program is this: Where do we fit in the administration? We have a few possibilities:

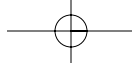
- **At network level:** We could filter users connecting to the server.
- **At PHP level:** We could prevent users from logging into the chat.
- **At database level:** We could discard messages from users from the database.
- **At IRC level:** We could use IRC's native network administration features.

### Network Level

Securing at network level only allows two possibilities: letting a connection through or not. This could be realized using a firewall or other possibilities of IP masking. This method is limited, complicated, insecure, and in general not what we want.

### PHP/Web Server Level

Securing at the Web server level basically allows connection to the server but restricts clients from logging in using password protection (or different methods of authentication). Basically it again boils down to letting a connection through or not, which is not really satisfying.



## 118 Chapter 3 Application Design: A Real-Life Example

However, this method can be used to emulate user bans. The common bans for IRC, namely K-lines and G-lines (local and global bans of users), cannot be used with a Web-based chat system, as all connections originate from the Web server. The only ban-able address would be the address of the Web server, which would completely ban the whole interface from the network. To still be able to filter out special users, connections should be evaluated at the PHP level.

### Database Level

The database level is a totally different approach. Clients are allowed to log in and chat, but their messages and session information are filtered in the database. Either an external tool or the chat code itself would check for the user to be allowed to say or do something and, based on this info, allow his/her messages to be inserted into the database—or not. But this strategy requires a very tight integration into the main chat code, is not very flexible (and kind of clumsy), and is inelegant to implement.

### IRC Level

IRC provides native administration features built into the server code and network protocol (we hope you read the RFC and are familiar with these possibilities).

Administration can even be done by regular users. Three levels are available:

- **Channel operators.** These operators have administrative control over channels. They can kick users, “mute” them, ban them, make other users into operators, and such (this level is available to all users).
- **IRC operators.** These operators have administrative control over the network (but not channels). They can kill users from the net, ban them, establish network links, and so on (this level is only available to special users).
- **Services.** Services have administrative control over channels but no control over the network, and are not able to perform like regular users. They also require a special login procedure (this level is only available to special users and is meant for automated clients).

As you can see, administration at IRC level can be done using a client running separately from the main chat system. A separate client with IRC operator and channel operator status would give the ideal combination of features that we need an administration system to have. Basically, only IRC operator status is needed initially, since as soon as the administration client has gained IRC operator status, it can gain channel operator status everywhere by killing all users from a channel. This is not a very nice method, but more effective and versatile than patching the IRC server code to give IRC operators equal rights to channel operators.

## Implementation

The implementation of the chat administration is designed to be quite similar to the main chat script. A bot is launched, which, with the help of phpIRC, logs into IRC and tries to register itself as IRC operator. Then it waits for further commands from a Web interface. These commands are issued like those in a database-backed RPC (remote procedure call). The bot will frequently query a table in the database that contains input commands for it. The commands are put into the database by the Web interface and consist of a function name, a session ID, and a parameter array. Whenever the bot finds a new command in the database, it executes it and writes the command results in an output table along with the session ID. Thus, the Web interface just has to write a command with a self-generated session ID and then only needs to wait until a result dataset with the same session ID pops up in the output table (see Figure 3.11).

This method allows flexible remote control over the bot.

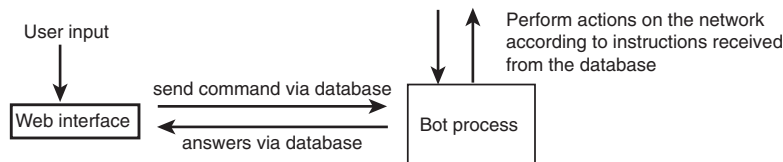


Figure 3.11 Database-backed RPC control over the administration bot.

## Summary

In this chapter, you've learned from a real-life example how to plan a development project. We've outlined the typical stages of development:

- Analyzing the requirements.
- Choosing an appropriate technology.
- Defining interfaces and APIs.
- Implementation.

You've followed us through the whole development phase, and we've drawn conclusions from our example that are applicable in most software projects. With this background, you're ready for the next part of this book, and we'll introduce you to some important concepts of Web applications in the next chapter.

