# Real-Time Collaborative Scala Development with Clide[*]

Martin Ring
DFKI Bremen
martin.ring@dfki.de

Christoph Lüth
DFKI Bremen, Universität Bremen
christoph.lueth@dfki.de

## ABSTRACT

We present Clide, a real-time collaborative development environment. Clide offers a new approach to tool integration which complements the way resources are shifted to the cloud today. We achieve this by introducing the novel concept of *universal collaboration*, which drops the distinction between human and non-human participants (so-called *assistants*) and enables innovative ways of interaction.

Clide has a highly flexible and distributed architecture based on Akka. Despite the complexity of the synchronisation of distributed document states, implementing assistants is pleasantly simple. To demonstrate the versatility and usability of the platform we implement a simple wrapper turning the Scala compiler into a collaborator, offering content assistance to other developers and tools.

## Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Interactive Environments; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work

## Keywords

Computer-supported cooperative work, Interactive Programming Environments, Distributed Programming Environments, Universal Collaboration

## 1. INTRODUCTION

Social platforms for software development gain in importance as development gets more distributed, and teams get more flexible. Popular offerings like *GitHub* or *BitBucket* have vastly improved the way physically distributed developers communicate and coordinate their work. Most of these platforms are based on distributed version control systems. These are asynchronous by design, as developers will often prefer to work on their own before sharing with others. However, the asynchronous work flow prevents the kind of closer interaction that comes with physical colocation, and disrupts common tasks like code review or pair programming, resulting in our view in a poorer experience for developers.

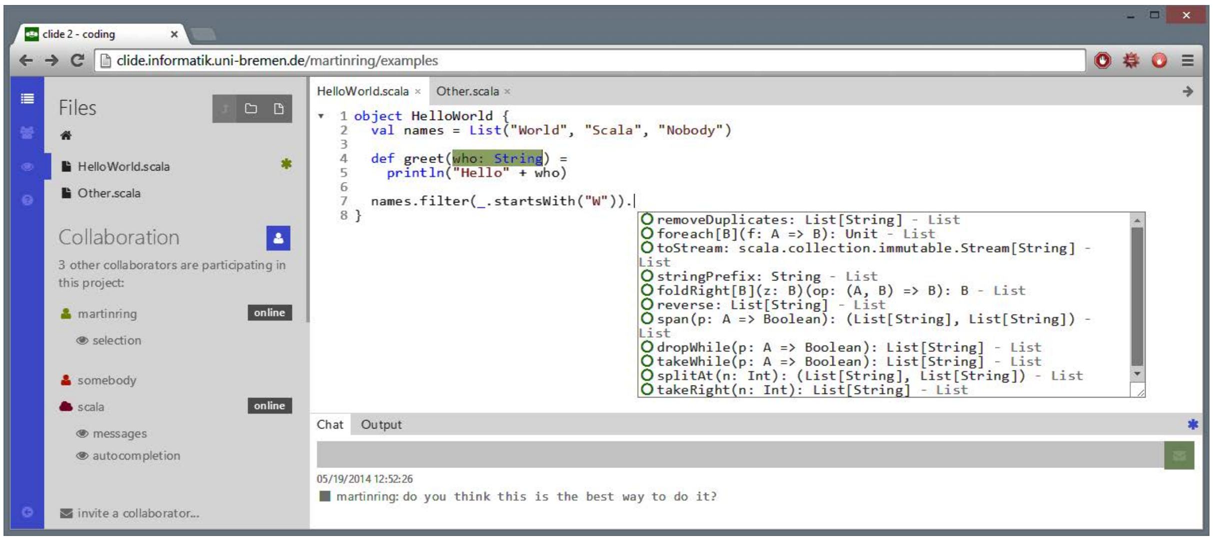
Apart from distributed development teams there is a second emerging trend of the last years. Resources are shifted towards the cloud, and computing power is increasingly distributed. Software developers already take advantage of this in many areas of the development process (utilising build servers, online documentations etc.) but the potential has as of yet not been broadly exploited at the core of the development process; development environments are still mainly running local. We believe that moving code assistance to the cloud can also bring developers closer together.

With the experimental platform Clide we want to provide the possibility to explore such new concepts of interaction in a distributed development environment. Clide offers real-time collaborative code editing and annotating both for humans and assistants (Clide's notion of plug-ins). Assistants and the user interface connect to the IDE through a unified TCP interface. In this unified interface, all interaction is performed by collaborative edit operations, such as text edits or adding annotations; we call this approach *universal collaboration*. We claim that by offering the same possibilities to humans and computers, humans can benefit in two ways. On one hand interaction between users becomes richer, while on the other hand interaction with machines becomes more natural. Apart from these advantages, the implementation of assistants becomes very easy due to the document-centric approach. We handle all concurrency issues at the core, allowing assistants to focus on the document and making their implementation easy.

Our contribution to the Scala community here is twofold. First, we exhibit a prototype of a collaborative real-time development environment for Scala. Second, we show a modern web application developed in Scala taking advantage of the Typesafe Reactive Platform, in particular of the Akka framework. We furthermore use the functional aspect of Scala for increased confidence in correctness.

This paper is structured as follows. We first introduce Clide from the user's perspective. We then outline the system architecture to explain the inner workings of Clide, and show how Clide can be extended to provide a basic implementation of a Scala assistant. We finish with a discussion, summarising the possibilities, advantages and shortcomings of Clide and related and future work.

[*]Research supported by BMBF grant 01IW13001 (SPECifIC).


clide 2 - coding

clide.informatik.uni-bremen.de/martinring/examples

Files

HelloWorld.scala

Other.scala

Collaboration

3 other collaborators are participating in this project:

martinring online

selection

somebody

scala online

messages

autocompletion

invite a collaborator...

HelloWorld.scala × Other.scala ×

```
object HelloWorld {
  val names = List("World", "Scala", "Nobody")

  def greet(who: String) =
    println("Hello" + who)

  names.filter(_.startsWith("W")).
}
```

```
removeDuplicates: List[String] - List
foreach[B](f: A => B): Unit - List
toStream: scala.collection.immutable.Stream[String] -
List
stringPrefix: String - List
foldRight[B](z: B)(op: (A, B) => B): B - List
reverse: List[String] - List
span(p: A => Boolean): (List[String], List[String]) -
List
dropWhile(p: A => Boolean): List[String] - List
takeWhile(p: A => Boolean): List[String] - List
splitAt(n: Int): (List[String], List[String]) - List
takeRight(n: Int): List[String] - List
```

Chat Output

05/19/2014 12:52:26

martinring: do you think this is the best way to do it?

**Figure 1: The IDE of the Clide web interface. The sidebar on the left and the chat area at the bottom are extended. The collaboration panel is located at the the second section of the sidebar. A collaborator selected the text "who: String". There is an an active help request, which has been answered by the Scala presentation compiler.**

## 2. WORKING WITH CLIDE

From the user's perspective, Clide appears as a web application divided into three layers: The public interface, the backstage area and the actual IDE. The public interface allows new users to sign up, and existing users to log in. The backstage area is the entry point after logging in. From here, users can manage their projects and start the IDE on any of the projects they own or collaborate in. (User and project management are deliberately kept simple to focus on real-time interaction, the central aspect of Clide.) The IDE is divided into a tabbed editor at the centre, a sidebar on the left and a chat and output area at the bottom. Both the sidebar as well as the bottom area may be hidden (Fig. 1).

The file management and most of the options in the sidebar should be familiar to users of other IDEs, apart from one unique feature: the *collaboration panel* allows to invite other users into the project. A notable characteristic is that inviting collaborators is not restricted to humans, but is the way all of the IDE functionality is provided. When working with Scala, the user should invite the Scala presentation compiler (see Sect. 4) which can provide the user with semantic information about the code.

When an assistant is connected to the system and invited into a project it can augment the files with additional semantic information through annotations. These may include syntax highlighting, type information, error and warning messages, evaluation results as well as more advanced annotations. The annotations are distributed to all connected collaborators, such that both humans and other assistants may use the information provided by one single assistant.

When users move their cursor around or select text, they implicitly create annotations which can be observed by all other collaborators. That way, users can highlight text about which they can talk in the chat, or get information about specific parts of the documents from a machine assistant. Users can also create more advanced annotations, *e.g.* the help request activated by pressing `Ctrl+Space`. This triggers the in-line display of a little drop down box filled with possible completions at the current context, but in the collaborative context all collaborating users can see the completion box and contribute completions.

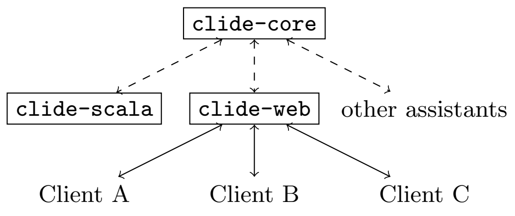


**Figure 2: Example setup of the clide infrastructure: Dashed lines represent TCP Connections via Akka remoting, solid lines are WebSocket connections. There are three web clients connected to `clide-web` where their requests are converted into internal messages and passed on to `clide-core`. Assistants, here `clide-scala`, connect to the core in the same way**

## 3. SYSTEM ARCHITECTURE

Clide has a highly flexible modular architecture built on top of the Typesafe Stack. We use Slick for data persistence and the Play Framework for the web interface. However, the most influential design decision was to use the Akka middleware. The actor paradigm is a natural fit for our distributed collaborative application, and Akka remoting allowed us to elegantly abstract over the TCP connections established between the modules. Additionally, the WebSockets used for fast bidirectional communication between the web server and the browsers fit smoothly into the actor model as they have queued message passing semantics. The infrastructure is modular (Fig. 2). The modules are very loosely coupled and only communicate via message passing.

Thanks to the loose coupling, modules do not have to be started in any particular order, and failures of individual modules do not propagate.

### *Operational Transformation.*

Internally the synchronisation of distributed, concurrently edited and annotated documents is achieved by an operational transformation algorithm inspired by the Google Wave approach [3]. In operational transformation, all conflicts are resolved automatically. This is achieved by considering operations instead of the documents, thereby gaining more information about the users' intentions. Operations are sequences of basic actions, and traverse a document sequentially, thereby transforming it. The basic actions include retaining, inserting, or deleting a character.

The core algorithm is the function *transform* which for any two operations $f$, $g$ applicable to the same document gives two operations $g'$, $f'$. When the transformed operations $g'$ and $f'$ are applied after $f$ and $g$ respectively, we get the same document again. The operations can be represented as case classes, and *transform* can be given as a functional program. We have formalised the algorithm in this way in the theorem prover Isabelle, proven the correctness, and generated functional Scala code using Isabelle's code generation facility [5] (see [2] for a full exposition). We have optimised the code by adding a simple compression which replaces sequences of contiguous retain, insert or delete actions by a single, parameterised action. The clide core server acts as an arbiter which integrates the concurrent operations from all clients and decides on their ordering. The operations are transformed accordingly and distributed to all clients. Any client can only send one operation at a time and will have to wait for an acknowledgement from the server, indicating that the edit has been integrated. During that time all concurrent operations from other clients have to be transformed against the pending edit as well as possible buffered operations. Further, since Clide is a web application, we additionally have to reimplement the client side part of the algorithm in JavaScript. To increase our confidence in the system, the *scala.js* compiler [4] could be used in future to run the *same* implementation on all clients and the server.

### *Annotations.*

An annotation is an arbitrary markup for a given text span; how it is rendered depends on the front-end (e.g. the web-frontend uses style sheets). Annotations are not deleted explicitly; they have a unique identifier, and remain until they are overridden by a subsequent annotation with the same identifier. Further, annotations behave like retain-only operations with side effects, which eases the integration into the operational transformation algorithm.

## 4. EXTENDING CLIDE

The concept of universal collaboration allows rich plug-ins as assistants that can augment the document with annotations, contribute content, and communicate with users and other assistants through a unified interface. Assistants can pick a specific state of the document to analyse and annotate; the transformation of the annotations takes place on the core server and is distributed to other clients. Implementors of assistants need not concern themselves with delayed computation results, network delay or concurrent editing operations by other users. The integration is optimistic in the sense that assistants may report back their results which in the meantime have been rendered meaningless, if for example other users keep on editing the document. This may result in inconsistent annotations being displayed. However, this consideration has to be made for any asynchronous development environment. The resulting behaviour is well-known from IDEs, thus conforming to users' expectations. In practice, many assistants wait for a certain amount of time with no activity before they start processing the document.

We have implemented a lightweight interface for assistants which abstracts away some of the more low level aspects of the communication. When developing a plug- in all we have to do is to implement a couple of callback methods. If a computation takes longer, a simple back pressure mechanism will automatically conflate subsequent messages in the background to prevent overwhelming the assistants. This is made possible by the compositional nature of operations and annotations.

### *The Scala Presentation Compiler as a Collaborator.*

As a proof of concept we implemented basic Scala assistance to demonstrate how Clide assistants work. For this purpose we wrote a tiny wrapper around the Scala presentation compiler that reports all compiler messages back to the server as annotations on the documents. The entry point for the implementation of new assistants is the abstract class `AssistantServer` which takes a constructor as an argument that creates an instance of `AssistBehavior` from a passed `AssistantControl` (Fig. 3).

The `AssistBehavior` has a number of abstract callback methods, which can be implemented to be informed about events in the project. `AssistantControl` on the other hand offers an interface of thread safe methods that can be called to annotate or edit documents, set the status or chat with users. Document states are identified via unique revision numbers and must be provided to all document related methods of the `AssistantControl`. This way, Clide always knows about which state of the document the assistant is talking and can then transform all actions accordingly to match the current server state of the document. The client side transformations necessary at the assistant are abstracted away from the interface. The `AssistBehavior` is local to a specific project. That means, AssistantServer takes care of joining projects to which the assistant has been invited. All other aspects of the assistant may be configured in the `application.conf` file.

To interface with the Scala presentation compiler, we create an instance of an interactive compiler (`scala.tools.nsc.interactive.Global`) per project. When we get the indication that a file is watched by a collaborator (`fileActivated`) we trigger the compiler to look into that file and save the state of the document for further reference. We have implemented a custom instance of `scala.tools.nsc.reporters.Reporter`, that passes compiler messages as annotations to the server. When files are edited we pass this information on to the compiler.

To enable code completion we need to track help requests from the users. These are reported back to the assistant through the callback method `helpRequested` in the `AssistBehavior`. When a user requests help, that method is triggered. We can then use the provided code position to deter-

```
object Scala extends
  AssistantServer(ScalaBehavior)

case class ScalaBehavior(
  control: AssistantControl) extends
  AssistBehavior {
  val mimeTypes = Set("text/x-scala")
  def start(project: ProjectInfo) = {...}
  def fileOpened(file: OpenedFile) = {...}
  def fileActivated(file: OpenedFile) = {...}
  def collaboratorJoined(who: SessionInfo) =
    {...}
  def fileChanged(file: OpenedFile,
    delta: Operation,
    cursors: Seq[Cursor]) = {...}
  def receiveChatMessage(from: SessionInfo,
    msg: String) = {...}
  def annotationsRequested(file: OpenedFile,
    name: String) = { ... }
  def helpRequested(from: SessionInfo,
    file: OpenedFile,
    pos: Int, id: String,
    request: String) = {...}
  def cursorMoved(cursor: Cursor) = {...}
  ...
}
```

**Figure 3: Implementing Scala Assistance for Clide. All methods in `AssistBehavior` are of type `Future[Unit]` to implement back pressure.**

mine the available completions in the current scope and pass these as so called response annotations on to the server.

Another useful feature to implement is to provide type information. For that purpose we watch the cursors of the users (callback `cursorMoved`) and then annotate the document at the entity under the cursor with type information.

### *Other Instances.*

An early instance of Clide provides a web interface for the Isabelle theorem prover [1, 2]; the Isabelle assistant provides slightly more functionality than the Scala assistant, such as mathematical notation. We have also implemented a simple Haskell assistant, which is essentially just a wrapper around the Glasgow Haskell compiler; it is more of a proof-of-concept, showing how programming languages with fast batch compilers (e.g. C) can be integrated into Clide easily.

As an example for some more complex interaction between collaborators, the Isabelle assistant can be used to generate Scala files which in turn will be considered by the Scala assistant. In practice that means, if one collaborator is working on the formalisation while another uses it in some other Scala file, any (valid) change to the formalisation will instantly propagate. This might for example result in additional completion options for the other collaborators.

## 5. DISCUSSION

While Clide is not quite ready for production use, it demonstrates that the web is ready for rich, distributed development environments without sacrifice in user experience and productivity. Our thesis is that closer interaction of humans and machines in the appropriate situations can enhance the productivity of developers and increase quality of resulting software, e.g. by enabling pair programming or real-time code reviews in physically distributed development teams, and our prototype will allow us to validate that thesis. We do *not* suggest that Clide replaces collaborative tools like *GitHub*, but instead complements them.

The system architecture of Clide shows that Scala and the Typesafe Reactive Platform are well suited to implement novel web-based applications. Moreover, the functional aspect of Scala allowed us to derive our core algorithm, namely the operational transformation, from a machine-proven formalisation, which increases confidence in the correctness of the implementation (in particular, converging documents across all clients).

### *Related and Future Work.*

While there are other examples of social WebIDEs like *Koding.com* or *Cloud9* (the latter of which even provides real-time collaboration), none of these are truly distributed. Users are provided with workspaces located on a single virtual machine, essentially just moving their local way of development onto that server. Clide takes a very different approach by moving the collaborative aspect at the centre of the architecture. Plug-ins and tools can run on multiple servers and are synchronised by the system, with the added advantage that they no longer have to be installed locally.

We have concentrated our efforts on the Clide core to make it as stable and easy to implement assistants for as possible. To turn the prototype Scala instantiation presented here into a production IDE, several improvements are required. First, it has to be integrated with some version control system. Second, the Scala assistant currently lacks project and build management; the user has no influence on the libraries available in the classpath of the presentation compiler, which is an essential requirement for production use.

### *Conclusion.*

The concept of universal collaboration turns out to be very powerful and opens an endless amount of possibilities to developers of IDE plug-ins. We see a strong potential in Clide as a heterogeneous social coding platform. As Clide is implemented in Scala, we believe that the Scala community can particularly benefit as the implementation of assistants happens in a familiar environment. We have a public demo instance of clide running at `http://clide.informatik.uni-bremen.de`, and invite you to give it a try!

## 6. REFERENCES

[1] Ring, M., Lüth C.: Collaborative Interactive Theorem Proving with Clide. Interactive Theorem Proving ITP 2014, LNAI 8588 Springer (2014) 467– 482.

[2] Lüth, C., Ring, M.: A web interface for Isabelle: The next generation. Conf. Intelligent Computer Mathematics 2013. LNAI 7961 Springer (2013) 326– 329

[3] Wang, D., Mah, A., Lassen, S.: Google Wave operational transformation. `http://tinyurl.com/q6xwdu7` (Accessed: 30.01.2014).

[4] Doeraene, S.: Scala.js website. `http://www.scala-js.org` (Accessed: 14.05.2014).

[5] Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. Theorem Proving in Higher Order Logics (TPHOLs 2007), Emerging Trends Proceedings. Dept. of Comp. Sci, U Kaiserslautern (2007) 128– 143