Research Article

Stefan Friese* and Kristian Rother

A mixed-paradigm component architecture for implementing web-based game servers

DOI 10.1515/comp-2016-0004
Received February 28, 2014; accepted October 01, 2015

Abstract: Games are a well-suited testbed for the development and evaluation of concepts and tools in artificial intelligence (AI). In our paper we outline the architecture of a web-based game server that was developed to support the teaching of artificial intelligence at the university level as well as research in the domain of AI and rule-based language development. The server combines a Prolog-based declarative approach with Java-based server technologies. The architecture consists of multiple, strictly separated components. Declarative components comprise the game-independent core engine as well as game-dependent logic and visualization descriptions. General operations (e.g. user management, load balancing, match maintenance etc.) and rendering of the visualization descriptions with actual GUI techniques are implemented imperatively. External interfaces are provided to integrate remote developers and learners to facilitate usage beyond the borders of a single university.

Keywords: component architecture; artificial intelligence; computer science education; gaming

1 Introduction

When certain concepts of AI are to be developed, evaluated or taught, games are often chosen as a field of application (cf. [2, 3] for examples of games in teaching). Many real-world problems have similarities to situations in games, especially in economics where game theory plays an important role. The advantage of games as an AI testbed comes from their (usually) clearly defined set of rules and their representation of a situation with a limited (which does not imply low) level of complexity. At our university, games serve as a teaching tool for AI concepts (focussing on rule-based agent development) and form the basis of our research. Important research topics in this field include domain-specific language (DSL) development (cf. [4] for a Bridge-specific language or [5] and [6] for general information on DSLs), knowledge reasoning and analysis of human behavior to create improved AI agents, e.g. using case-based reasoning or data mining (see [7] for an example and a review of other CBR approaches in gaming AI). The focus is on turn-based games, which includes card games most importantly, but also a variety of board games.

To facilitate the development of AI artifacts in the application domain of games, we developed a game server. This paper describes the architecture and implementation of the server (for an overview of teaching AI with the game-server see [8]). Different programming paradigms are integrated within the server: The game-independent core engine and game-dependent logic and visualization descriptions are implemented in a declarative way, while general operations (e.g. user management, load balancing, match maintenance etc.) and rendering of the visualization descriptions with actual GUI techniques are implemented imperatively. This way, a clean separation of reusable components is possible, which is recognized as an advantageous approach in game development (cf. [9] or [10]), and each part of the whole system can be implemented in a well-suited programming paradigm.

Game (server) architectures have been discussed by several authors before, but the scope is mostly different from ours, as the field of online gaming is quite large. For example, architectures have been developed by [10, 11] and [12] specifically for real-time strategy games. Another type of game server architecture can be found in [13–18] and [19]. These authors present architectures focusing on massively multiplayer games, where different issues are relevant, e.g. bandwidth consumption and synchronization.

*Corresponding Author: Stefan Friese: University of Duisburg-Essen, Information Systems for Production and Operations Management, Universitätsstr. 9, 45141 Essen, Germany, E-mail: stefan.friese@icb.uni-due.de
Kristian Rother: University of Duisburg-Essen, Information Systems for Production and Operations Management, Universitätsstr. 9, 45141 Essen, Germany, E-mail: kristian.rother@icb.uni-due.de

1 This article is based on the conference paper [1]. It provides an additional section on the method used, an additional table that presents tournament results as well as minor improvements.
2 Method

Following the paradigm of design-oriented information systems research as outlined in [20] the game server is considered an artifact, more specifically an instantiation ([21], p. 77), that needs to provide a benefit to its stakeholders ([21], p. 82). Since the general goal of our project was to develop a game server that supports teaching and research of AI, the relevant stakeholders in this case are researchers, teachers and students. The envisioned benefits or objective of the solution as Peffers et al. ([22], p. 55) call it are mostly qualitative in nature. The game server should allow flexible game development to support research and improve the overall quality of student results. The expected benefits from a pedagogical point of view are outlined in [8]. The improvements of the student results and other criteria are shown in section 7 (Evaluation). According to ([20], p. 9) apart from the benefits, the artifact has to satisfy the scientific principles of abstraction, originality and justification. While the server itself is considered an original contribution thus satisfying the principle of originality, this paper satisfies the principle of justification and the communication activity or guideline according to Peffers et al. ([22], p. 56) and Hevner et al. ([21], p. 90) respectively.

The characteristics of the server that satisfy the principles of abstraction (other than the fact that it is to be used in both teaching and research) are indicated in the following text. The next section describes the requirements or what would be considered the ‘to be’ conception ([20], p. 8) in design-oriented terms.

3 Requirements

Since our research covers a variety of games the server should not be bound to a single game but rather support different games. These games should be playable by human players against other human players, by AI players against AI players, and any mixed set of human and AI players. It should be possible to call the AI players remotely, so that an AI host can be located anywhere on the Internet, making the server usable for any AI technology. All of these requirements satisfy the principle of abstraction. Furthermore, it should be possible to integrate the server with our programming environment for students and also with learning environments to teach game playing (thus providing a concrete benefit to both teachers and students).

Apart from the general functional goals, there were several non-functional design goals for the server and its architecture. First of all, the development of games should be as flexible as possible. This means that it should be easy to add new games to the server based on a description of their rules. Also, the visualization of the games should not cause an implementation overhead. Hence, a generalized concept to describe the visualization on top of the game rules description was needed to avoid redundant implementation efforts for different games. The technical demands should be as low as possible on the client side. The reason for this requirement is that it is advantageous for the analysis of human players to attract as many players as possible. Therefore additional technical requirements such as platform specific components that require installation or specific browser plugins should be avoided. This is especially important as some of the games that we use such as the card game Rok (also known as Shelem) have their main distribution area in less developed countries. Another side effect of the large target audience is that scalability and load balancing should be respected in the system design. The requirements could be elaborated in more detail, but the above description provides an overview sufficient for the scope of this paper.

4 Technology choices

We use (SICStus) Prolog both in teaching and in research and the existing programming environment for students is also Prolog-based, which implied that the game server should be compatible with Prolog. With this (hard) constraint in mind we considered both writing the entire game server in Prolog and combining Prolog with other technology. The requirement of low technological demands on the client side and wanting to reach a broad audience led to a web-based approach due to the ubiquitous availability of browsers. Unfortunately we were unable to find a mature, Prolog-based full stack web framework. At one end of the spectrum, The PiLLoW library ([23]) provides low level support for web related activities such as HTTP requests. Further up there are also ways to directly embed Prolog into web pages². Prosper ([24]) is a good step in the right direction and comes closest to fulfilling our needs but ultimately all investigated solutions required, in our estimation, writing a nontrivial amount of maintenance code.

This led us to consider a hybrid approach. SICStus Prolog provides interfaces to C/C++, Tcl/Tk (primarily for Tk-

² See http://sourceforge.net/projects/prolog-ws/ for a web service oriented approach that extends this idea but is still in alpha
GUIs), Java and .NET ([25]) as well as third party support for other languages such as Python³. We decided to stick to one of the options bundled with SICStus and eliminated all third party options from our set of candidates. Because of its strong platform independence which fulfills the requirement of reaching a large audience Java emerged as the first candidate. Our choice was reinforced because Java could be considered the most natural fit for Prolog integration (most Prologs come with a Java interface, cf. [26] for a Java interface for multiple Prologs). The choice was further supported by the requirement of scalability, which led to Apache Tomcat as the web server of choice due to its powerful and mature load balancing features and flexibility. From a technology point of view the chosen separation makes sense because it plays to the strength of both Prolog (rule-based game engine, game descriptions and agents) and Java (strong enterprise level web technology).

5 Architecture overview

The requirements led to an approach which combines different programming paradigms. Fundamentally, the architecture of the game server can be separated into declarative (Prolog) and imperative components (Java). The main imperative part is concerned with the general operations of the server. This includes the registration of users, login/logout, session management and the maintenance of matches (creation of new matches, running matches and finished matches). The second imperative part of the architecture is the generation of a visualization in the web browser and handling user actions therein.

The declarative component of the server handles both the abstract definition of the match scene to be visualized and the game logic. The logic part is split into game-independent general game logic and game-specific logic. Each game needs a specification of its rules and a specification of the visual representation (called a view) of a match situation for the participating players. These views are an abstract description which is independent of the actual visualization technology. This way it is possible to replace the imperative rendering component by a different one, without any impact on the declarative implementation of the game or on the resulting rapid development of new games. The server architecture consists of multiple, strictly separated components which represent different aspects of the system. These components will be explained in the following sections. The basic architecture is depicted in Figure 1.

5.1 Operations

The general operations are an imperative component of the server. It represents the entry point to the system and communicates with the database, the engine, the chat and the user. Its main functions are user maintenance, match maintenance and view visualization. User maintenance means the registration of new users, login and logout, and changing user profiles. The maintenance of matches contains all match maintenance activities from a user perspective (configuring and starting matches, joining matches, deleting matches). Hence, the operations component must provide a login / registration page and a lobby. The actual creation of a new match (the initial state of the match) is handled by the engine. The view visualization is the actual user interaction inside a match. The operations component receives a view description from the game engine and presents it to the user. The user generates events in his browser which are again sent to the game engine by the operations component. These events are clicks on elements, drag and drop actions, and actions related to forms. For the communication between operations and engine, three basic calls can be distinguished: Creating a new match, joining an existing match, and performing actions inside a match.

³ http://sicstus.sics.se/thirdparty.html
The interface of these calls looks like this (+ and - indicate input and output parameters, respectively):

\[
\text{initMatch}(+\text{MatchID}, +\text{MatchOptions}, +\text{PlayerList}, +\text{PlayerDetails}, -\text{State}, -\text{Views}, -\text{Round}, -\text{Phase})
\]

\[
\text{joinMatch}(+\text{MatchID}, +\text{PlayerDetails}, +\text{OldState}, -\text{NewState}, -\text{Views}, -\text{Result}, -\text{Round}, -\text{Phase})
\]

\[
\text{play}(+\text{Player}, +\text{GUIAction}, +\text{MatchID}, +\text{OldState}, -\text{NewState}, -\text{Views}, -\text{Result}, -\text{Round}, -\text{Phase})
\]

PlayerDetails contains information about the player types (human, local or remote AI). Result, Round and Phase contain meta information for the operations component like the match result (if it is finished) and the current state of the match (e.g. a board number in Bridge).

### 5.2 Chat

The chat component is used for the communication between users. The chat is a separate component which must handle multiple chat instances: There should be one chat instance for the lobby, and one chat instance for each match. The chat must be able to handle messages visible only to a subset of users (e.g. private messages between members of a team in a match). The chat is called by the operations component.

### 5.3 Data Base

The data base ensures persistence of all data used on the server. This includes data related to the users (user names, passwords, configuration of AI players), games (which games are available, how many players do they have, which options may be configured etc.) as well as data on matches (waiting matches, currently running matches and finished matches, including their history which is used for replay and for analysis purposes). Finally, it contains data on the positions of elements for match visualization. As the visualization description is independent of an actual GUI technology, there must be a mapping which describes where exactly to display the visual elements (forms, cards etc.) of each game. An excerpt of the ER diagram which shows the relations relevant for games is depicted in Figure 2.

### 5.4 Engine

The engine is a declarative component which covers the game logic and the abstract visualization of components of the server. Generally speaking, the engine describes the relationship between an old match state, an action performed by a user, the resulting new match state and a visualization of the new match state. This complex task is further divided to support the separation of concerns and reusability. Firstly, each game provides a set of rules which transforms an old match state into a new one based on the user action. Secondly, each game implementation includes rules to describe the visualization of a situation (which is called a view). The declarative part also contains some game-independent components. These are common parts which are shared between the rules and visualizations of games (often used general rules like the transformation of representations of card lists etc.), and mechanisms to access AI players both locally and remotely.

#### 5.4.1 Game Logic

The game logic provides a generic interface to run a game. Firstly, it is able to generate a new initial state when the operations component requests to start a new match. The operations component sends a list of configuration options to the game logic. These options are game dependent (for example, the number of rounds or a specific scoring rule) and stored in the data base. Secondly, and most importantly, the game logic transforms an old match state into a new one based on the action the user performed and the rules of the game. As the engine is stateless, the operations component provides the old match state for each such state transformation. Some games require further operations, e.g. if the game rules allow players to leave or enter an already started match.
5.4.2 Game View Generation

The second part of the definition of a game is a set of rules for its visualization. These rules describe the views of each player of a match based on the current state. A view is a technology-independent description of elements to be presented to the users. There is one view for each player in a match and possibly additional ones for spectators. Defining the available abstractions of a view is a particularly important part of the architecture, as no visualization requirement of a specific game can be fulfilled if it cannot be expressed by the means of a view, but the structure of a view should still be simple and generic to facilitate development. Each view consists of the following elements:

- **Player**: The name of the corresponding player
- **ViewItems**: A list of all items the player can see (e.g. cards, including those face-down) and their positions. The positions are logical positions which are independent of the actual rendering in the operations component.
- **Animations**: A list of animations of items (an item moving from one position to the other or an animated update of an item)
- **AllowedActions**: A list of actions the player may perform in the current state. This includes clicking on items as well as drag-and-drop actions.
- **Forms**: A list of forms visible to the user. These forms can display information (e.g. a score table) or they may be interactive, e.g. to choose a bid value. The forms are also represented in an abstract format which can be rendered in different GUI technologies by the operations component.
- **WidgetItems**: A list of items with special behavior. This makes it possible to develop advanced widgets which exceed the possibilities of forms and animations, e.g. opening a previous trick in a pop-up.
- **UndoData**: This component represents the possibility to undo an action of a player. It consists of the number of match states to go back and a list of players which have to be asked for their agreement. It is then a task of the operations component to handle undo requests and negotiations.
- **Chat**: A list of possible recipients for the integrated chat. A recipient is a named set of players. In many games it is either required to privately talk with a subset of players, or some players are not allowed to communicate (e.g. in Bridge).

An example view for a scene from a simple game like Tic-Tac-Toe could look like this:

```javascript
view(
    playerName, 
    [1- tile (blank), 2- tile (x), 3- tile (o),
     4- tile (blank), 5- tile (blank),
     6- tile (x), 7- tile (blank),
     8- tile (blank), 9- tile (o)],
    [],
    [1-[ click ], 4-[ click ], 5-[ click ],
     8-[ click ]],
    [ form (1,[ text (‘{i18nItsYourTurn }’) ]),
     update (7, tile (o))],
    undo (2, playerName),
    [ recipient (‘{i18nOpponent }’,
     [ playerName ])]
)
```

Based on these views and positioning information in the database, the operations component is able to render the scene in the browser. This rendering is currently implemented with using HTML, CSS and JavaScript, but due to the generic description, other UI mechanisms could be used on the architecture.

5.5 AI communication

An important part of the architecture is the integration of AI players, as the goal of development was to create an AI testing system. On first sight, it might appear reasonable to call AI players from the operations component, because this component already interacts with human players. However, a different approach was chosen for the game server architecture due to several reasons. One reason is that the operations component has no understanding of the actual match state. It renders a view for a user, but has no clue about the meaning of the displayed items - for example, it does not know that the cards which should be rendered at the bottom of the screen are the player’s hand cards. An AI agent, however, is much simpler to implement if it is provided with a functional perspective to the match state rather than a visual one. Another reason is that an efficiency gain can be expected if the communication between the declarative engine and the imperative operations level is bypassed. This is not very important in interactive games, but useful for the simulation of large tournaments between AI players. As a consequence, all AI players are directly accessed by the engine. Apart from that decision, there are still multiple possible ways to realize the communication. For efficiency reasons, it is attractive to call AI players locally, but the server should also be able
to access remote AIs. Hence, both a local and a remote interface are included.

5.5.1 Remote AI interfaces

The engine sends the player’s perspective of the current match state (meta data and game-specific data) and a knowledge base for the agent. The agent responds with an action and an updated version of its knowledge base. The knowledge base is a private data store for the agent which is sent back to it in the next call. This store is necessary as the agent is usually stateless and could possibly take part in multiple matches at the same time. However, an agent may maintain a local state if necessary (e.g. for learning agents). The players’ match state perspective does contain information on the identity of the match as part of the meta data.

In our implementation, there are two different remote interfaces. The first one is a basic interface using sockets. Each developer who implements an AI agent must install a server which listens for incoming socket connections on a specified port.

Many programming languages and frameworks support the development of web services. A web service encapsulates a reusable piece of functionality and provides it over a network, typically HTTP (cf. [27]). To simplify the development of remote AI agents an RPC-based web service interface has been added. A developer using this interface only needs to implement the web service according to the WSDL specification and does not have to implement the socket communication.

5.5.2 Local AI interface

The local AI interface is an interface to access AI agents which are located on the same server as the game server. This interface allows a very efficient access as there is no network overhead at all. The disadvantages are close coupling to the implementation of the engine and potential security risks if outside developers are allowed to run their agents on the server. That is why this interface is merely used for local development.

6 Additional features

The students who participate in our AI courses implement their agents in a specifically designed learning and programming environment (EPPU, cf. [28]). One of the development goals was to integrate this system with the game server, so students can directly use their agents on the server and evaluate it in matches against other AI agents or human players. This integration has been implemented based on the local AI interface. An integration on this level was possible because our game engines and the agents of our students are both implemented in Prolog. Each time a student creates a new version of an agent in the programming environment, this agent is transported to the game server. Furthermore, the game server contains a mechanism to fetch all available agents from EPPU, to make sure it is possible to use the most recent student agents on arbitrary installations of the game server.

To enable the integration of the game server into learning environments to teach game playing, a template-based anonymous game mode has been implemented. For a developer it is possible to generate a template from a match. This template can be instantiated automatically when accessed via a specific URL from a learning environment. This mechanism supports creating templates in any state of the match, which means a template may either be just a match in its initial state, or a scenario somewhere during a match, because technically it is based on the concept of a match state. When the learning environment requests an instance of the template, the operations component creates a match from the template and inserts an anonymous player replacing the human player in the template.

To facilitate the development of learning agents, further callbacks from the engine have been implemented. These additional calls of the AI agents inform them about match results. This is important for the agent to judge the success of his actions with respect to the result of the whole match. Without this mechanism, the result of the match would not be visible for the agent, because no action is required from it after the final state of the match.

7 Evaluation

In this paper, we have presented a game server which combines the advantages of declarative descriptions of game logic with powerful imperative server techniques. The strict separation and loose coupling of components enables flexibility and rapid development of games and allows a high amount of reuse. The chosen abstractions proved to be suitable when different games were implemented.

The game server has been applied for teaching in an AI project with master students developing agents for a card
game. These agents were evaluated in tournaments where all agents (including those of earlier semesters) played against each other in sets of 4000 matches to rule out random effects (for a comparison, the Annual Computer Poker Competition\textsuperscript{4} uses 3000 hands to determine the winner of a poker match). The results of these tournaments allowed us to compare the old agents developed before and the new ones developed after the introduction of the game server as a learning and evaluation tool. All match sets between old and new agents were won by the new agents with the exception of one extraordinary good old agent. Nevertheless we need more empirical evidence to claim an effect of the game server on the quality of the developed agents because the number of participants in the project is low (41 in six semesters) and isolating the effect is difficult. In Table 1, the results of a limited size tournament is shown, where only the best agents of each year participated. best2013 is the winner of the first year where the gameserver was used, which is only beaten by the best2010 agent.

The game server has proven to be a useful tool in research as well because it allowed us to quickly prototype new games and to prototype and test AIs for different games (especially Rok, Bridge and Poker). The ability to play against AI players as a human player has proven particularly valuable for prototyping and debugging agents, especially in complex scenarios with cooperating agents like in the game of Bridge.

### 8 Outlook

So far most of the development has been focused on card games and local AIs for said games at a single university. The existing AIs have been written directly in Prolog but infrastructure for describing the agents in game specific DSLs and automatically generating them from that description is currently being developed. There are also plans to extend this DSL-based approach to the formulation of the game rules themselves, possibly even using AI techniques to extract the rules from a digitized rulebook. The potential addition of boardgames, especially purely cooperative ones for the study of multi-agent systems, provides another interesting direction. Making the server available to the public will open some new and interesting research opportunities like analyzing human play and generating agents that mimic winning players. Furthermore it will hopefully lead to the contribution of agents for different games from beyond the borders of our university.

### References

\textsuperscript{4}http://www.computerpokercompetition.org/index.php/competitions/rules/96-2014-rules

\[1\] S. Friese, K. Rother. A mixed-paradigm approach for implementing a web-based game server. In V. Novitzká and Š. Hudák


