DisCo user's guide

Florian Grabbe Philip Müller

6. Februar 2015

Inhaltsverzeichnis

Zusammenfassung

DisCo steht für "distributed contest" und ist ein Framework, welches zur Durchführung von Programmierwettbewerben mit Live-Shootout (1.2) genutzt werden kann. Dieser *user's guide* beschreibt die Funktionalität und die äußeren Schnittstellen des DisCo-Frameworks sowie die zur Durchführung eines Wettbewerbes notwendigen Anpassungen.

1 Allgemeines

1.1 Einleitung

Dieser *user's guide* soll jedem ermöglichen, einen Wettbewerb mit DisCo durchzuführen. Wenn Sie den Erläuterungen einmal nicht folgen können, werfen Sie einfach einen Blick auf die mit

dem Framework ausgelieferte Beispielkonfiguration (siehe 2.3.2) oder wenden Sie sich an die Autoren.

1.2 Art der Wettbewerbe

DisCo ist für Wettbewerbe entwickelt worden, bei denen mehrere Teilnehmerprogramme[1], potenziell in einem Live-Event mit Zuschauern und Begleitprogramm, gleichzeitig gegeneinander antreten und innerhalb einer vorgegebenen Zeitspanne alle dasselbe Problem bestmöglich lösen. Wenn die konkrete Problemstellung es erlaubt, kann die Attraktivität für die Zuschauer durch eine Live-Visualisierung der Ergebnisse erhöht werden.

Der Fairness halber läuft jedes Teilnehmerprogramm auf einem eigenen Rechner. Hierbei kann jedes Teilnehmerprogramm beliebig viele Lösungsvorschläge abgeben, von denen nach Ablauf der Zeit der letzte Vorschlag gewertet wird.

Die Anzahl der zu spielenden Runden[2] und die Art und Weise, nach der die Gewinner ermittelt werden (z.B. nach dem K.O.-Prinzip oder Sieg durch Punkte) ist frei wählbar.

1.2.1 Anforderungen an die Problemstellung

Für das im Wettbewerb zu lösende Problem sollte es mehrere gültige, aber nur eine optimale Lösung geben. Diese optimale Lösung sollte aber bei entsprechender Parametrisierung schwierig genug zu finden sein, um den Vorgang für die Zuschauer interessant zu machen (z.B. 15-30 Sekunden Rechenzeit). Die verschiedenen Lösungen sollten vergleichbar sein und mit Punkten bewertet werden können.

Üblicherweise empfiehlt sich hier ein NP-vollständiges Problem, bei dem allerdings eine eingereichte Lösung auch in annehmbarer Zeit überprüft werden kann. Die benötigte Zeit zur Überprüfung einer Lösung sollte möglichst im niedrigen Millisekundenbereich liegen, da teils viele Antworten auf einmal verarbeitet werden müssen und bei einem Stau an der Bewertungskomponente eine Verzögerung in der Anzeige entsteht.

Das Problem sollte zudem parametrisierbar sein, um mehrmals mit steigendem Schwierigkeitsgrad gespielt werden zu können. Indem diese Parameter im Vorfeld nicht bekannt gegeben werden, kann man ausschließen, daß Lösungsvorschläge vorberechnet werden können.

Ein Beispiel für eine mögliche Problemstellung ist ein vereinfachtes Beladungsproblem: Es ist ein Quadrat gegeben und eine Menge kleinerer Quadrate. Nun sind Quadrate aus der Menge so in das große Quadrat hinein zu legen, dass sie sich nicht überschneiden und die bedeckte Fläche maximal ist. Je größer das zu belegende Quadrat ist und je mehr kleinere Quadrate gegeben sind, desto schwieriger wird es, in der zur Verfügung stehenden Zeit die optimale Lösung zu finden. Ein gutes Maß für die Qualität einer Lösung wäre die bedeckte Fläche.

1.2.2 Technische Voraussetzungen an die Teilnehmerprogramme

Die Teilnehmerprogramme benötigen keine grafische Oberfläche und müssen auch keine Netzwerkkommunikation unterstützen, es genügen einfache Kommandozeilenanwendungen. Diese bekommen alle benötigten Daten (die Parameter, die das zu lösende Problem beschreiben)

[1] Teilnehmerprogramme sind die von den Teilnehmern des Wettbewerbs eingereichten Programme.

[2] Runden sind in diesem Kontext zeitlich begrenzte Durchläufe mit gleicher oder unterschiedlicher Parametrisierung des Problems.

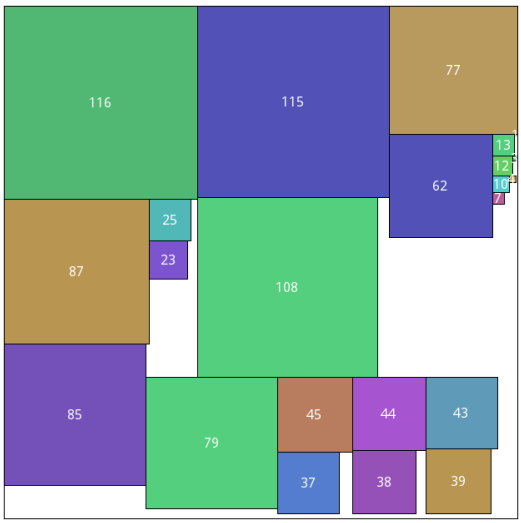

Abbildung 1: Beispiel für eine Visualisierung des Beladungsproblems

über die Standardeingabe, nachdem sie gestartet wurden[3]. Das Format ist hierbei frei wählbar und sollte so gewählt werden, dass es für die Teilnehmer möglichst einfach zu parsen ist. Es ist zu beachten, dass die Standardeingabe anschließend vom Framework nicht geschlossen wird (es folgt kein `EOF`, Teilnehmerprogramme sollten auch nicht auf ein solches warten).

Die Bearbeitungszeit läuft ab der Übergabe der Problemparameter. In dieser Zeit können die Teilnehmerprogramme beliebig viele Vorschläge machen. Pro Vorschlag ist genau eine Zeile auf die Standardausgabe zu schreiben (das Format ist auch hier frei wählbar). Alle Lösungsvorschläge werden zur Weiterverarbeitung an die "Bewertungskomponente" (siehe 2.1.2) gesendet.

1.3 Was macht das Framework dabei?

Das Framework steuert den grundlegenden Ablauf des Wettbewerbs. Dies beinhaltet das Starten und sichere Beenden sowie die Kommunikation mit den Teilnehmerprogrammen und den Komponenten (siehe 2.1).

Jedem Teilnehmerprogramm kann für die Ausführung ein eigener PC zugewiesen werden (siehe 2.3), damit allen zur Lösung des Problems dieselbe Rechenleistung zur Verfügung steht, obwohl sie gleichzeitig laufen. Der Aufbau des Clusters wird dabei ebenfalls vom Framework übernommen.

Jede Problemstellung kann über mehrere Runden gespielt werden. Hierfür speichert das Framework für die aktuell ausgewählte Problemstellung einen Zustand (siehe 2.3). Der Zustand ist dabei eine beliebige Zeichenkette, die durch die ins Framework eingehängten Komponenten manipuliert werden kann. So kann der Zustand durch anwenden einzelner Teilnehmerlösungen verändert werden, um beispielsweise in der nächsten Runde das Problem mit einer leicht veränderten Ausgangssituation zu Lösen.

Dieser Zustand könnte z.B. bei dem bereits erwähnten Beladungsproblem verwendet werden. So könnte man als Zustand die Menge der zur Verfügung stehenden Quadrate definieren. Jetzt lässt man eine Runde laufen und nimmt dann die in der besten Lösung verwendeten Quadrate aus der Menge (modifizierter Zustand). Damit ist diese Lösung in der nächsten Runde ausgeschlossen.

[3]Zwischen dem Programmaufruf und der Übergabe der Daten ist genügend Zeit um beispielsweise einen Interpreter oder eine VM zu laden, damit alle Teilnehmer die gleiche Zeit zur Verfügung haben. Unabhängig von der Implementierungssprache.

Eine andere interessante Verwendung des Zustandes wäre in einem Scrabble-basierten Problem möglich. Hier könnte man die Anfangsbelegung des Spielfelds als Zustand nutzen und die beste Lösung jeweils der Belegung hinzufügen, so dass sie ebenfalls in der nächsten Runde nicht wieder als gültige Lösung eingereicht werden kann.

1.4 Architektur

1.4.1 Überblick

Um das Framework auf einen Wettbewerb anpassen zu können ist die Logik, die abhängig von der konkreten Problemstellung ist, in externe Komponenten ausgelagert. Diese Komponenten können in einer beliebigen Sprache implementiert werden. Die Kommunikation mit dem Framework erfolgt in JSON über die Standard-Ein-/Ausgabe (siehe 2.1). Je nach konkreter Problemstellung können für einige Funktionen Standardkomponenten verwendet werden, um Entwicklungsaufwand zu sparen. Die Standard-Komponente für das "user interface" zum Beispiel ist sehr allgemein gestaltet und sollte mit jeder Problemstellung funktionieren, sofern keine Visualisierung gewünscht ist.

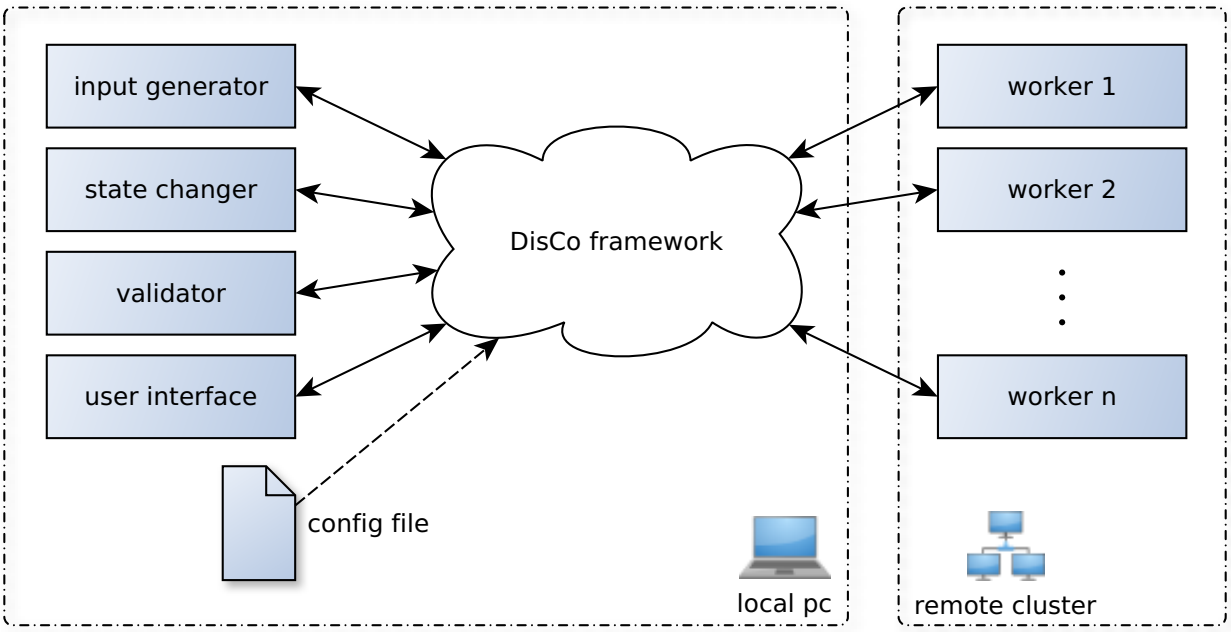

Abbildung 2: Aufbau des Frameworks

1.4.2 Komponenten

Die genauen Protokolle, über die die Komponenten mit dem Framework kommunizieren, sind unter 2.1 aufgeführt.

input generator Erzeugt die problembeschreibenden Daten, die alle Teilnehmerprogramme über die Standardeingabe bekommen. Für die Erzeugung dieser Daten bekommt der "input generator" die Problemspezifikation für das aktuell ausgewählte Problem sowie den aktuellen Zustand. Die Problemspezifikation und der Initialzustand können in der Konfigurationsdatei definiert werden.

Im einfachsten Fall wird kein rundenübergreifender Zustand benötigt (wenn jedes Problem nur einmal gespielt werden soll) und die Problemspezifikationen in der Konfigurationsdatei können unverändert als Problembeschreibung an die Teilnehmerprogramme gesendet werden.

Der "input generator" kann aber auch auf Basis der Problemspezifikation eine Zufallskomponente ins Spiel bringen. Im Fall eines Scrabble-Problems könnten z.B. ein Wörterbuch in der Problemspezifikation und die Belegung des Spielfelds als Initialzustand definiert werden. Die Auswahl der zur Verfügung stehenden Buchstaben erfolgt dann aber zufällig, z.B. basierend auf den im Wörterbuch vorkommenden Buchstaben.

state changer Verändert den aktuellen Zustand des Problems durch Anwendung einer Teilnehmerlösung. Nicht jeder Wettbewerb benötigt einen Zustand (z.B. wenn die Probleme nicht über mehrere Runden gespielt werden sollen), in diesem Fall muss der "state changer" nicht ausgetauscht werden. Die Standardkomponente für den "state changer" gibt den Zustand einfach unverändert zurück.

Ein Beispiel für einen Wettbewerb mit veränderbarem Zustand wäre das Beladungsproblem, wenn hier alle Quadrate, die in der Lösung eines bestimmten Teilnehmers verwendet wurden, aus der Liste der verfügbaren Quadrate für die nächste Runde entfernt werden sollen.

validator Die Bewertungskomponente bekommt den von einem Teilnehmerprogramm abgegebenen Vorschlag (zusammen mit der zugehörigen Problembeschreibung) und überprüft diesen auf Gültigkeit und Qualität. Wird ein Vorschlag von der "validator"-Komponente als ungültig erkannt, wird das Teilnehmerprogramm vom Framework deaktiviert. Andernfalls erhält es die vom "validator" bestimmte Punktzahl. Dabei liegt es im Ermessen des Implementierers, wann und ob überhaupt Vorschläge als ungültig erklärt werden sollen.

user interface Ist für die Steuerung des Ablaufs (Auswahl eines Problems, starten/beenden einer Runde, ...) und die Darstellung zuständig. Üblicherweise stellt das "user interface" eine Tabelle dar, in der die Teilnehmer, ihre Punkte und der letzte Vorschlag angezeigt werden.

Im Gegensatz zu den anderen Komponenten kann es mehrere (unterschiedliche) "user interface"-Komponenten geben. Dies ist hilfreich für Problemstellungen bei denen die Visualisierung der Lösungen mehr Platz in Anspruch nimmt und beispielsweise auf einem separaten Monitor angezeigt werden soll.

Außerdem kann so z.B. die Standardkomponenten für die Listenansicht verwendet werden und zusätzlich eine getrennte Komponente für die problemspezifische Visualisierung.

1.5 Ablauf

Die Durchführung eins Wettbewerbs mit dem Framework erfolgt nach folgendem Ablauf:

1. Starten des Frameworks

   → Die Konfigurationsdatei wird gelesen.

   → Die Komponenten werden gestartet.

   → Das Rechner-Cluster wird aufgebaut und die Teilnehmerprogramme werden auf den entfernten Rechnern gestartet. Sie bekommen aber noch keine Problembeschreibung.

2. Auswahl einer Problemstellung über die Benutzeroberfläche.

   → Der "input generator" erzeugt die Problembeschreibung für die Teilnehmerprogramme.

3. Starten einer Runde des ausgewählten Problems über die Benutzeroberfläche.

   → Die Teilnehmerprogramme bekommen die Problembeschreibung und können Lösungsvorschläge einreichen.

4. Jeder eingereichte Lösungsvorschlag wird vom "validator" bewertet.

   → In der Benutzeroberfläche wird der letzte Vorschlag (inkl. Bewertung) jedes Teilnehmers angezeigt.

5. Nach Ablauf der Rundenzeit werden alle Teilnehmerprogramme beendet und direkt ohne Input neugestartet, damit sie für die nächste Runde bereit sind. Das Framework wartet noch auf die Validierung eventuell in der Warteschlange vorhandener Vorschläge und erklärt dann die Runde für beendet.

6. Über die Benutzeroberfläche kann nun

   - ein anderes Problem ausgewählt werden (weiter bei Schritt 2.), oder
   - eine weitere Runde mit dem aktuellen Problem gestartet werden (weiter bei Schritt 3.). In diesem Fall sollte der Zustand vorher verändert werden (durch Anwenden einer Teilnehmerlösung über die Benutzeroberfläche).

2 Benutzung

2.1 Implementierung der Komponenten

Die vier Komponenten, die die wettbewerbsspezifische Logik implementieren, werden beim Starten des Frameworks mitgestartet und laufen solange, bis das Framework wieder beendet wird. Sollte eine Komponente mal abstürzen, wird sie automatisch neugestartet. Auf diese Weise stehen die Komponenten immer für Anfragen bereit. Der Verzeichnis- und Dateiname der einzelnen Komponenten kann in der Konfigurationsdatei (siehe 2.3) eingestellt werden. Die Kommunikation mit dem Framework erfolgt über die Standard-Ein/Ausgabe.

Beispielkomponenten finden Sie unter den in 2.3.2 angegebenen Pfaden.

2.1.1 Eingabeerzeuger (`input generator`)

Der "input generator" bekommt vor dem Start einer neuen Runde eine JSON Nachricht mit folgenden Elementen:

Name	Typ	Beschreibung
`problem`	`string`	die Problemspezifikation aus der Konfigurationsdatei
`state`	`string`	der aktuelle Zustand

Beispiel: `{ "problem" : "[1,2,3,4]", "state" : "42" }`

Die Antwort muss ebenfalls in JSON erfolgen und den folgenden Wert enthalten:

Name	Typ	Beschreibung
`worker input`	`[string]`	Die Eingabe, die die Teilnehmerprogramme anschließend als Problembeschreibung erhalten. Die Strings dürfen keine Zeilenumbrüche enthalten, da sie zeilenweise an die Teilnehmerprogramme ausgegeben werden (Ein Wert pro Zeile).

Beispiel: `{ "worker input" : ["[1,2,3,4]", "42"] }`

2.1.2 Bewertungseinheit (`validator`)

Der "validator" ist für die Bewertung der Lösungsvorschläge zuständig. Für jeden Vorschlag, den eines der Teilnehmerprogramme einreicht, bekommt der "validator" vom Framework eine JSON-Nachricht mit folgenden Elementen:

Name	Typ	Beschreibung
`input`	`[string]`	Beschreibung des Problems, die auch das Teilnehmerprogramm bekommen hat.
`output`	`string`	Der eingereichte Lösungsvorschlag des Teilnehmerprogramms.

Beispiel: `{ "input" : ["[1,2,3]", "6"], "output" : "[(0,0,3)]" }`

Die Antwort muss ebenfalls in JSON erfolgen und die folgenden Werte enthalten:

Name	Typ	Beschreibung
`score`	`number`	Qualität des Vorschlags. Negative Werte stehen für eine ungültige Lösung.
`caption`	`string`	Kurzer Beschreibungstext zur Anzeige in der grafischen Oberfläche. Dies kann entweder direkt der Vorschlag sein oder eine Umschreibung, mit der Menschen mehr anfangen können, siehe Beispiel.

Beispiele: `{ "score" : 9, "caption" : "25% coverage" }`
`{ "score" : -1, "caption" : "invalid solution ..." }`

Die Kommunikation zwischen Framework und Komponente ist synchron, der "validator" bekommt also erst den nächsten Vorschlag, nachdem der aktuelle validiert wurde.

Wird eine Lösung als ungültig bewertet (negative Punktzahl), so wird das Teilnehmerprogramm vom Framework beendet und deaktiviert (dies kann über das "user interface" rückgängig gemacht werden).

2.1.3 Zustandsveränderer (`state changer`)

Der "state changer" bekommt eine JSON Nachricht mit folgenden Elementen:

Name	Typ	Beschreibung
`proposition`	`string`	Die Lösung des ausgewählten Teilnehmerprogramms.
`state`	`string`	Der aktuelle Zustand.

Beispiel: `{ "proposition" : "(1+2+3+4)", "state" : "200" }`

Die Antwort muss ebenfalls in JSON erfolgen und den folgenden Wert enthalten:

Name	Typ	Beschreibung
`state`	`string`	Der neue Zustand.

Beispiel: `{ "state" : "210" }`

Die Kommunikation zwischen Framework und Komponente ist synchron.

2.1.4 Benutzeroberfläche (`user interface`)

Über die Benutzeroberfläche kann der Ablauf des Wettbewerbs gesteuert werden. Dies geschieht über einfache JSON Nachrichten, die an das Framework gesendet werden können:

`{ "action":` *Aktion*`,` *Parametername*`:` *Parametertyp* `}`

Wobei folgende Aktionen möglich sind (der Parameter wird nicht bei allen Aktionen benötigt):

Aktion	Parameter		Beschreibung
	Name	Typ	
`block worker`	`worker id`	`string`	blockiert den Teilnehmer mit der angegebenen *worker id*
`unblock worker`	`worker id`	`string`	reaktiviert den Teilnehmer mit der angegebenen *worker id*
`choose problem`	`problem idx`	`number`	wählt das angegebene Problem aus
`start round`	— *ohne Parameter* —		startet die Runde, indem allen Teilnehmerprogrammen die Problembeschreibung gesendet wird
`kill all workers`	— *ohne Parameter* —		beendet alle Teilnehmerprogramme sofort (sie werden anschließend neugestartet) und beendet damit auch die aktuelle Runde
`apply proposition`	`worker id`	`string`	übernimmt die Antwort des angegebenen Teilnehmerprogramms in den aktuellen Zustand
`save game state`	`file path`	`string`	speichert den aktuellen Zustand des Wettbewerbs
`load game state`	`file path`	`string`	lädt einen gespeicherten Wettbewerbszustand
`add scores`	— *ohne Parameter* —		addiert die Punkte der aktuellen Runde auf die Gesamtpunkte
`quit program`	— *ohne Parameter* —		beendet das gesamte Framework
`get all data`	— *ohne Parameter* —		Alle Daten erfragen: Workerinfos, Zustand, Problemliste, ...

Beispiele:
```
{ "action" : "block worker", "worker id" : "pwb01" }
{ "action" : "choose problem", "problem idx" : 2 }
{ "action" : "start round" }
```

Nach jeder Änderung informiert das Framework die Benutzeroberflächen (es kann mehrere geben) über die aktualisierten Daten mit Hilfe einer asynchronen JSON Nachricht. Diese Nachrichten sind also im allgemeinen keine Antworten auf die obenstehenden "actions", sondern können zu jedem Zeitpunkt getriggert werden. Format der "event"-Nachrichten:

`{ "event":` *Event*`,` *Parameter...* `}`

Folgende *Event*s sind dabei möglich (je nach Eventtyp sind weitere Parameter vorhanden):

`worker updated`
: Dieses Event wird gesendet wenn sich an den teilnehmerspezifischen Daten etwas geändert hat. Z.B. wenn ein Teilnehmerprogramm einen Lösungsvorschlag sendet oder wenn in der Benutzeroberfläche ein Teilnehmer geblockt/reaktiviert wird. Die Daten werden in einer Liste mit genau acht Elementen und dem Namen `"worker data"` übertragen:

  1. Die ID des Teilnehmers. Typ: `string`
  2. Der zuletzt gesendete Lösungsvorschlag des Teilnehmerprogramms, oder `null`, wenn noch keiner gesendet wurde. Typ: `string | null`
  3. Die vom "validator" bestimmte Kurzbeschreibung des Lösungsvorschlags. Typ: `string`
  4. Die vom "validator" bestimmte Punktzahl des Lösungsvorschlags. Typ: `number`
  5. Die normierte Punktzahl des Lösungsvorschlags. Diese wird erst nach dem Rundenende berechnet, da die Lösung des besten Teilnehmers als Maßstab genommen wird. Typ: `number`
  6. Die Summe der erreichten Punkte über mehrere Runden (für das aktuelle Problem). Typ: `number`
  7. Ob das Teilnehmerprogramm geblockt ist. Aktive Teilnehmer sind mit dem Wert `"no"` gekennzeichnet, ausgeschiedene Teilnehmer mit einem JSON Objekt welches angibt, als wievielter dieser deaktiviert wurde. Das ist entscheidend, wenn man den Gewinner im K.O.-Prinzip ermittelt. Typ: `string | { "idx": number }`
  8. Ob das Teilnehmerprogramm gerade rechnet (`true`) oder nicht (`false`). Typ: `boolean`

  Beispiele:
  ```
  { "event": "worker updated",
    "worker data": ["pwb01", "((1 + 2) + (3 + 4))", "1+2+3+4",
                    200, 0, 2000, "no", true] }
  { "event": "worker updated",
    "worker data": ["pwb02", "foo", "Invalid syntax",
                    0, 0, 2000, {"idx": 0}, false] }
  ```

`round started`
: Dieses Event wird ausgelöst, sobald eine Runde durch die Aktion `"start round"` (siehe oben) gestartet wird. Der Parameter `"round number"` (Typ: `number`) gibt an, die wievielte Runde gestartet wurde.

Beispiel: `{ "event": "round started", "round number": 1 }`

`round ended`
: Dieses Event wird ausgelöst, sobald die für dieses Problem eingestellte Zeit abgelaufen ist, alle Teilnehmerprogramme beendet wurden und auch alle Lösungsvorschläge bewertet wurden. Der Parameter `"round number"` (Typ: `number`) gibt an, die wievielte Runde beendet wurde.

  Beispiel: `{ "event": "round ended", "round number": 1 }`

`worker input changed`
: Diese Nachricht wird gesendet, wenn sich durch die Auswahl eines anderen Problems oder durch Aktualisierung des Zustands die Problemspezifikation für die Teilnehmerprogramme ändert. Der Parameter `"worker input"` (Typ: `[string]` gibt dabei die neue Problembeschreibung an. Die Strings in der Liste sind die Zeilen, die auch die Teilnehmerprogramme beim Starten der Runde erhalten.

  Beispiel:
  ```
  { "event": "worker input changed",
    "worker input": ["[1,2,3,4]", "42"] }
  ```

`save game state`
: Als Reaktion auf eine `"save game state"`–Aktion gibt diese Nachricht mit einem Parameter `"result"` (Typ: `"string"`) an, ob das Speichern erfolgreich war. Mögliche Rückgabewerte sind:

`ok`	Speichern der Datei war erfolgreich
`eacces`	Berechtigungen fehlen
`eisdir`	Unter diesem Namen existiert ein Verzeichnis
`enoent`	Pfad existiert nicht
`enospc`	Speicherplatz nicht ausreichend
`enotdir`	Pfad enthält Datei, wo Verzeichnis sein sollte

  Beispiel: `{ "event": "save game state", "result": "ok" }`

`load game state`
: Als Reaktion auf eine `"load game state"`–Aktion gibt diese Nachricht mit einem Parameter `"result"` (Typ: `"string"`) an, ob das Laden erfolgreich war. Mögliche Rückgabewerte sind:

`ok`	Laden der Datei war erfolgreich
`eacces`	Berechtigungen fehlen
`eformat`	Dateiinhalt hat das falsche Format
`eisdir`	Unter diesem Namen existiert ein Verzeichnis
`enoent`	Pfad existiert nicht
`enotdir`	Pfad enthält Datei, wo Verzeichnis sein sollte

  Beispiel: `{ "event": "load game state", "result": "ok" }`

`problem chosen`
: Dieses Event wird ausgelöst, wenn durch die Aktion `"choose problem"` (siehe oben) eine andere Problemstellung ausgewählt wurde. Der Parameter `"problem idx"` (Typ: `number`) gibt dabei den Index des ausgewählten Problems an. Mit diesem kann aus der Nachricht `"all data"` (siehe unten) die genaue Problembeschreibung ermittelt werden.

Beispiel: `{ "event": "problem chosen", "problem idx": 2 }`

`problem state changed`

Wird ein Problem über mehrere Runden gespielt, liefert diese Nachricht mit dem Parameter `"problem state"` den aktuellen Zustand, sobald sich dieser durch die Aktion `"apply proposition"` (siehe oben) ändert.

Beispiel:
```
{ "event": "problem state changed",
  "problem state": "neue Zustand"}
```

`all data`

Dieses Event wird automatisch nach dem Start der Benutzeroberfläche gesendet oder wenn mit der Aktion `"get all data"` danach gefragt wird. Es enthält folgende Elemente:

`running` gibt an, ob aktuell eine Runde läuft — Typ: `boolean`

`workers` Liste aller Teilnehmer und ihrer Daten. — Typ: `[TeilnehmerDaten]`

Die `TeilnehmerDaten` werden jeweils in einer Liste mit genau neun Elementen angegeben (ähnlich wie im `"worker updated"`–Event):

1. Die ID des Teilnehmers. — Typ: `string`
2. Der Name des Teilnehmerprogramms. — Typ: `string`
3. Die Gruppe, in der der Teilnehmer antritt (laut Konfigurationsdatei, siehe 2.3). — Typ: `string`

4.–10. genau wie die Punkte 2.–8. im `"worker updated"`–Event (siehe oben): letzter Vorschlag, Kurzbeschreibung, Punktzahl, normierte Punktzahl, Punktsumme, der geblockt-Status und ob das Teilnehmerprogramm gerade rechnet.

`problems` Liste der Problemstellungen und ihrer Daten. — Typ: `[ProblemDaten]`

Die `ProblemDaten` werden jeweils in einer Liste mit genau fünf Elementen angegeben:

1. Der Index der Problemstellung, so wie er auch in der Aktion `"choose problem"` und im Event `"problem chosen"` verwendet wird. — Typ: `number`
2. Die Kurzbeschreibung des Problems aus der Konfigurationsdatei. — Typ: `string`
3. Die Spezifikation des Problems aus der Konfigurationsdatei, die auch der "input generator" bekommt. — Typ: `string`
4. Die für eine Runde zur Verfügung stehende Zeit in Millisekunden. — Typ: `number`
5. Der Initialzustand des Problems aus der Konfigurationsdatei. — Typ: `string`

`problem idx` Der Index des aktuell ausgewählten Problems. — Typ: `number`

`round` Die Nummer der aktuell laufenden Runde bzw. der zuletzt beendeten Runde, je nachdem, ob gerade eine Runde läuft (siehe `running`). — Typ: `number`

`worker input` Die Definition des ausgewählten bzw. laufenden Problems, so, wie sie auch die Teilnehmerprogramme bekommen werden bzw. haben. — Typ: `[ string ]`

`state` Der Zustand des aktuellen Problems. — Typ: `string`

Beispiel:
```
{ "event": "all data",
  "running": false,
  "workers": [
    ["pwb01", "hello", "", "2 + 4", "2+4", 6, 0, 0, "no", false],
    ["pwb02", "foo",   "", null,    "",    0, 0, 0, "no", false],
    ["pwb03", "blub",  "", "3 + 3", "3+3", 9, 0, 0, "no", false]
  ],
```

```
"problems": [ [0, "erstes Problem", "[1,2,3,4]", 5000, "42"],
              [1, "zweites Problem", "[2,3,4]", 2000, "42"] ],
"problem idx": 1,
"round": 1,
"worker input": ["[1,2,3,4]", "42"],
"state": "Zustand" }
```

2.2 Vorbedingungen

Damit das Framework richtig funktioniert, müssen ein paar Voraussetzungen erfüllt sein.

2.2.1 Zugriff per SSH

Der Aufbau des Rechner-Clusters, auf dem die Teilnehmerprogramme ausgeführt werden, erfolgt per SSH. Hierfür ist es erforderlich, dass ein Login auf den betroffenen Rechnern ohne Angabe von Benutzername oder Passwort erfolgen kann. Dies lässt sich über eine Public-Key-Authentifizierung erreichen. Ist der Benutzername auf den entfernten Rechnern nicht derselbe wie lokal, kann dieser in der eigenen SSH-Konfiguration (`~/.ssh/config`) angegeben werden:

```
Host 192.168.1.*
     User discoUser
```

Binärdateien des Frameworks und Teilnehmerprogramme müssen auf allen Rechnern dem per SSH eingeloggten Nutzer unter dem selben, konfigurierbaren Pfad zur Verfügung stehen.

Befinden sich diese Daten auf einem Benutzerprofil, das erst beim Login von einem Server geladen wird, so muss sichergestellt werden, dass dies vor dem Starten des Framworks passiert (z.B. indem manuell eine SSH-Verbindung mit den betroffenen Profilen aufgebaut wird).

Das Framework kann auch vollständig auf dem lokalen Rechner ausgeführt werden. Dies ist jedoch nur zu Testzwecken sinnvoll, da dann alle Teilnehmerprogramme gleichzeitig und zusätzlich zum Framework auf dem lokalen Rechner laufen. Allerdings wird so kein SSH und damit auch keine Public-Key-Authentifizierung benötigt. Erreicht werden kann dies, indem alle IP-Adressen des Clusters in der Konfigurationsdatei auf den Wert des Makros "`LOCAL_IP`" des `Makefiles` ("`127.0.0.1`") gesetzt werden.

2.2.2 Benötigte Pakete

Auf dem Steuerungsrechner, auf dem das Framework und die externen Komponenten laufen, werden folgende Pakete benötigt:

- `erlang` R16B oder 17
  Für die Programme `erl` und `escript`.
  Die Kompatibilität zu anderen Versionen ist nicht ausgeschlossen, allerdings wurde das Framework nur mit den Versionen R16B und 17 getestet. Bei abweichenden Versionen muss ggf. der Wert von `require_otp_vsn` in der Datei `rebar.config` angepasst werden.

- `make`
  Ist nicht zwingend erforderlich, erlaubt aber die Nutzung des Makefiles zum Erzeugen und Starten des Frameworks. Viele Aktionen werden an `rebar` (ein Buildtool für Erlang) weitergeleitet.

- `git`
  Das Buildtool `rebar` verwendet `git` um die verwendeten Erlang-Bibliotheken herunterzuladen (siehe auch `rebar.config`).

- ssh client (z.B. `openssh`)
  Wird benötigt, um das entfernte Cluster aufzubauen.
  Falls das Framework komplett auf dem lokalen Rechner ausgeführt werden soll, wird kein SSH benötigt.

- `ip` (optional)
  Zur automatischen Ermittlung der eigenen externen IP-Adresse für den Makro-Wert "`LOCAL_IP`" im `Makefile`.

- `rsync` (optional)
  Wird nur beim optionalen Aufruf von "`make distribute`" für die Verteilung des Erlang-Codes auf die entfernten Cluster-Rechner benötigt.

- Pakete, die von den verwendeten Komponenten zur Ausführung benötigt werden. Die wettbewerbsunabhängige Benutzeroberfläche benötigt `python` und `pyqt4-dev-tools`.

Auf den Rechnern des Clusters, auf denen die Teilnehmerprogramme laufen sollen, werden folgende Pakete benötigt:

- Pakete, die evtl. von den Teilnehmerprogrammen zur Ausführung benötigt werden
- `erlang` (gleiche Version wie auf dem Steuerungsrechner)
- `rsync` (wie Steuerungsrechner)
- ssh server (wie Steuerungsrechner)

2.3 Konfiguration

Die Einstellungen, die für einen Wettbewerb benötigt werden, können in einer Konfigurationsdatei angegeben werden. Dies sind unter anderem die Pfade zu den Komponenten, die die wettbewerbsspezifische Logik implementieren, die Liste der IP-Adressen des Clusters, auf dem die Teilnehmerprogramme ausgeführt werden sollen und die zu spielenden Problemstellungen.

Welche Konfigurationsdatei zu verwenden ist, wird direkt im `Makefile` definiert.

2.3.1 Syntax

Die Konfiguration erfolg in Erlang-Syntax und besteht aus einer Liste, in der für jede Erlang-OTP-Application (in diesem Fall nur eine: "disco") eine Liste von Schlüssel-Wert-Paaren in Form von Tupeln definiert wird (siehe auch: `http://www.erlang.org/doc/man/config.html`).

```
[
  {disco,
    [
      {key1, "value1"},
      {key2, 1337},
      {key3, ["hello", "world"]},
      {key4, {foo, 1, "2", []}},
    ]
  }
].
```

Hierbei ist der Schlüssel ein Atom und der Wert kann ein beliebiger Term sein. Welche Einstellungen es gibt und welche Werte dort erlaubt sind, wird anhand der folgenden Beispielkonfiguration erläutert.

2.3.2 Beispielkonfiguration

Den Quelltext der einzelnen Komponenten finden Sie an den unter "local components" angegebenen Pfaden.

```
[
  {disco,
    [
      %% --- local components ---
      %% commands relative to the current directory:
      %% (barkeeper is the "input generator" and changer the "state changer")
      {validator, "make run -sC priv/countdown/components/validator/"},
      {barkeeper, "priv/general/components/barkeeper/barkeeper.rb"},
      {changer,   "priv/general/components/changer/changer.rb"},
      {gui,       ["make run -sC priv/general/components/gui/"]},

      %% --- remote workers ---
      %% remote directory containing the 'ebin', 'deps' and 'apps' folder
      %% If this is an empty string (default), the folder structure and
      %% location is assumed to be the same on master and slave nodes.
      {remote_node_path_prefix, "/home/stud/inf/inf9404/pwb/"},

      %% worker commands relative to the remote_node_path_prefix
      %% '%id%' will be replaced by the woker id (e.g. 'pwb_01'; see workers)
      {worker_run_cmd,  "make run  -sC priv/countdown/workers/%id%/"},
      {worker_name_cmd, "make name -sC priv/countdown/workers/%id%/"},
      {workers, [% {<worker id>, <ip address>, <additional information>}
                 {pwb01, '192.168.1.101', "students"},
                 {pwb02, '192.168.1.102', "students"},
                 {pwb03, '192.168.1.103', "students"},
                 {pwb04, '192.168.1.104', "alumni"},
                 {pwb05, '192.168.1.105', "alumni"},
                 {pwb06, '192.168.1.106', "staff"},
                 {pwb07, '192.168.1.107', "staff"}
                ]
      },

      %% When the cluster is started in parallel, multiple nodes are booted
      %% at the same time. The degree of parallelism can be adjusted through
      %% the slice size which determines the number of concurrent threads
      %% to use. Make sure `MaxStartups' in the config of your ssh daemon is
      %% large enough.
      {cluster_start_mode, parallel}, % :: sequential | parallel
                                      %                  (default: sequential)
      {startup_slice_size, 10},       % :: pos_integer()  (default: 10)

      %% Score calculation mode.
      %% raw:        the solution scores are added up
      %% normalized: the rounded percentage that this score has of the maximum
      %%             percentage from the current round is added up.
      %%             Example:
      %%             The worker A has 20 points. The best worker in the
      %%             current round, B, has 50 points. So worker A gets
      %%             40 normalized points added to his total score while
```

```
        %%              worker B gets 100 normalized points added.
        %% ranked:      The ''reverse rank'' gets added to the total score.
        %%              So when there are 12 workers, the worker with the
        %%              best solution gets 12 points, the one with the worst
        %%              solution gets just 1 point. In the case of multiple
        %%              solutions with equal scores, those workers share a
        %%              rank. So if all workers have equally good solutions,
        %%              all workers get 12 points added to their total score.
        {score_mode, ranked},           % :: raw | normalized | ranked
                                        % default: ranked

        %% --- list of problems to solve ---
        %% Each problem specification can be used for an arbitrary number of
        %% rounds.
        {problems, [%{<description>, <specification>, <answer time>, <state>}
                    {"First problem", "[1,1,2,3,5,8] 42",      1000, ""     },
                    {"Prime numbers", "[2,3,5,7,11,13,17] 42", 9000, ""     },
                    {"Impossible",    "nothing",                  0, ""     }
                   ]
        }
    ]
  }
].
```